# Dynamic processes, testbenches, & modelling styles

P. Marwedel*
Informatik 12, U. Dortmund

---

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

# Dynamic processes (SystemC 2.1)

Functions and methods can be used as dynamic processes.

Prerequisite:
**#define SC_INCLUDE_DYNAMIC_PROCESSES**

Functions that can be used as processes:
**void** inject(); //ordinary fct w/o args or return value
**int** count_changes(**sc_signal**<**int**>& sig)//ordinary fct
**bool** TestChan::Track(**sc_signal**<packet>& pkt); //method
**bool** TestChan::Errors(**int** maxwarn, **int** maxerr); //method

# Syntax for creating dynamic processes

Registration of Dynamic Processes without return value

```
sc_process_handle hname = sc_spawn (
/* void*/ sc_bind(&funcName, ARGS),
processName,
SpawnOptions
);
sc_process_handle hname = sc_spawn(
/*void*/ sc_bind(&methName, object, ARGS)
processName,
SpawnOptions
);
```

default: call by value

reference to calling module, e.g. **this**

stack size, don't_initialize, sensivity

# Debugging and signal tracing

SystemC has no built-in viewer for trace files (disadvantage if compared to VHDL).

Recommended to generate VCD (value dump format) files.

Waveform interchange format (WIF) also supported.

```
sc_tracefile* tracefile;
tracefile = sc_create_vcd_trace_file(tracefile_name);
if (!tracefile) cout << "There was an error" << endl;
…
sc_trace(tracefile, signal_name, "signal_name");
…
sc_start();
…
sc_close_vcd_trace_file(tracefile);
```

# Debugging and signal tracing (2)

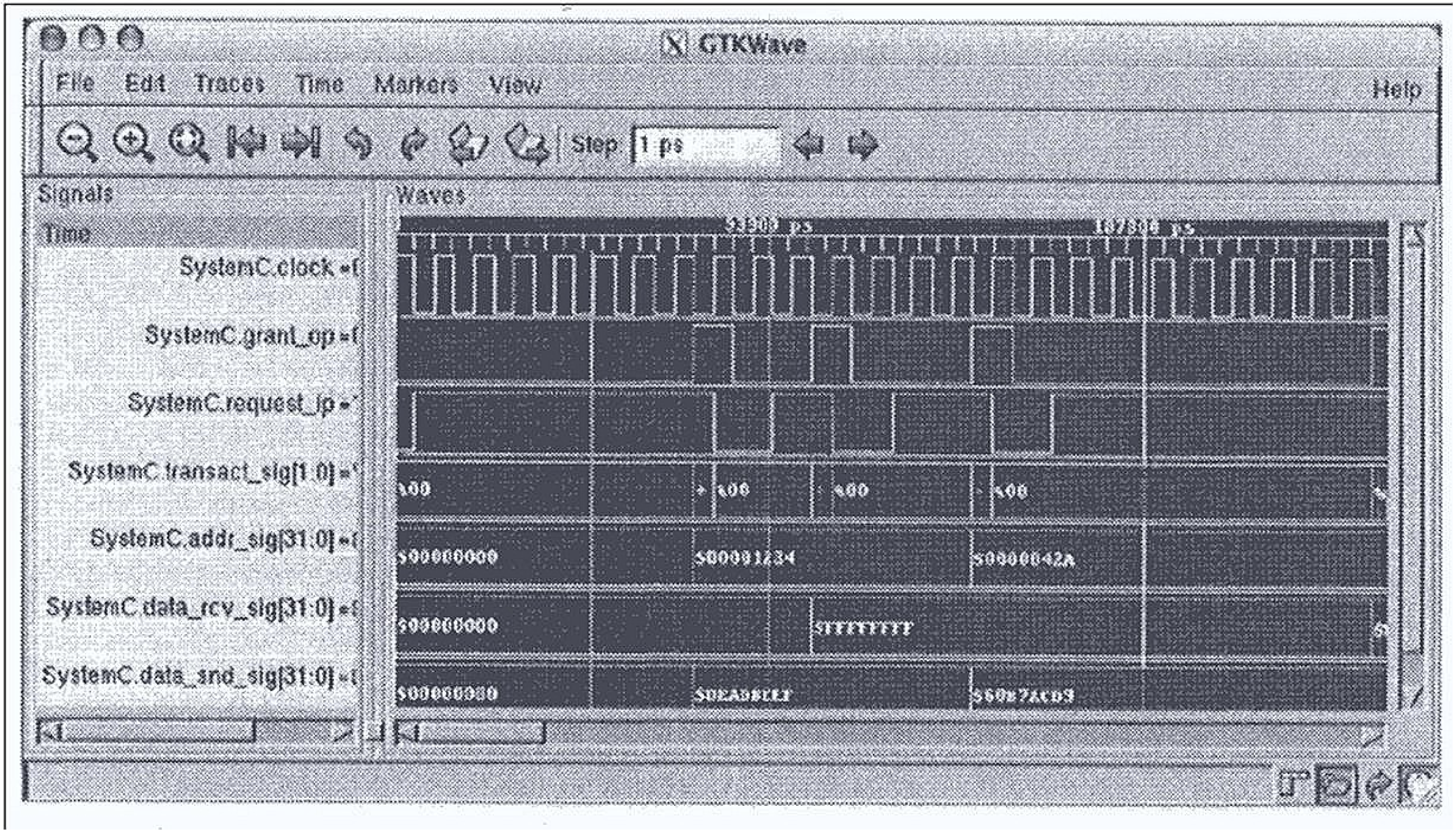
- Signal name must be declared before calling **sc_trace**.
- Hierarchical notation for signals in submodules is feasible.
- Variables, signals and ports can be traced.
- No explicit filename extensions.
- Tracing can be included in constructors of modules

# Debugging and signal tracing: Example

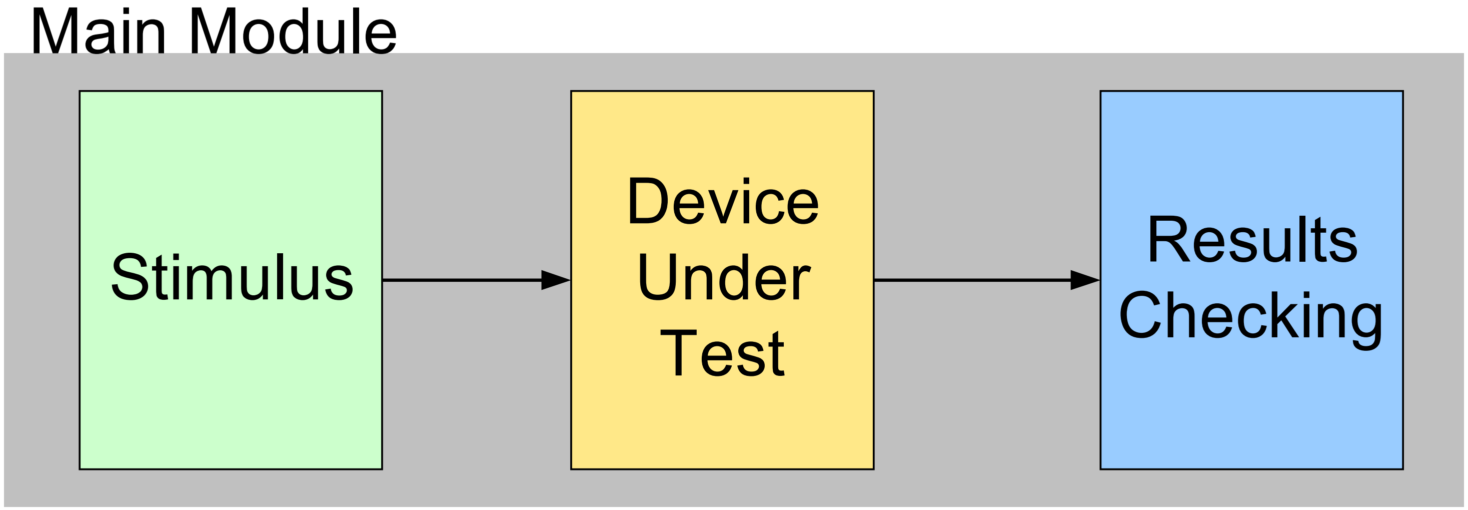
```
//file wave.h
SC_MODULE(wave) {
  sc_signal<bool> brake;
  sc_trace_file* tracefile
  double temperature;
};
```

```
//file wave.cpp
wave::wave(sc_module_name nm);// constructor
:module(nm) { …
  tracefile = sc_create_vcd_file("wave");
  sc_trace(tracefile, brake,"brake");
  sc_trace(tracefile,temperature,"temperature");
}// endconstructor
wave::~wave() {
  sc_close_vcd_file(tracefile);
  cout << "created wave.vcd" << endl;
}
```

# Sample output



# Typical TestBench



The stimulus module will provide stimulus to the Device Under Test (DUT) and the Results Checking module will look at the device output and verify the results are correct.

# TestBenches

TestBenches are used to provide stimulus to a design under test and check design results

The testbench can be implemented in a number of ways.

The stimulus can be:

- generated by one process and results checked by another.
- embedded in the main program and results checked in another process.

The checking can be:

- embedded in the main program, etc.

There is no clear "right" way to do a testbench, it is dependent on the user application.

# Typical TestBench

The stimulus module can be implemented by reading stimulus from a file, or as an **SC_THREAD** process, or an **SC_CTHREAD** process.

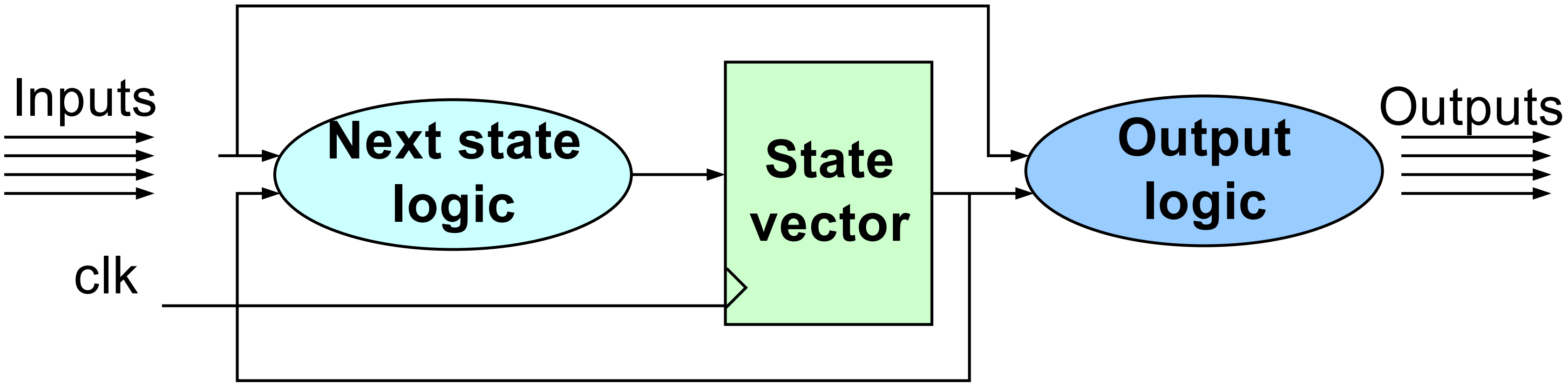
The same is true of the results checking module.

Some designers combine the stimulus and results checking modules into one module.

Also the results checking module can be left out if the designer does manual analysis of the output results.

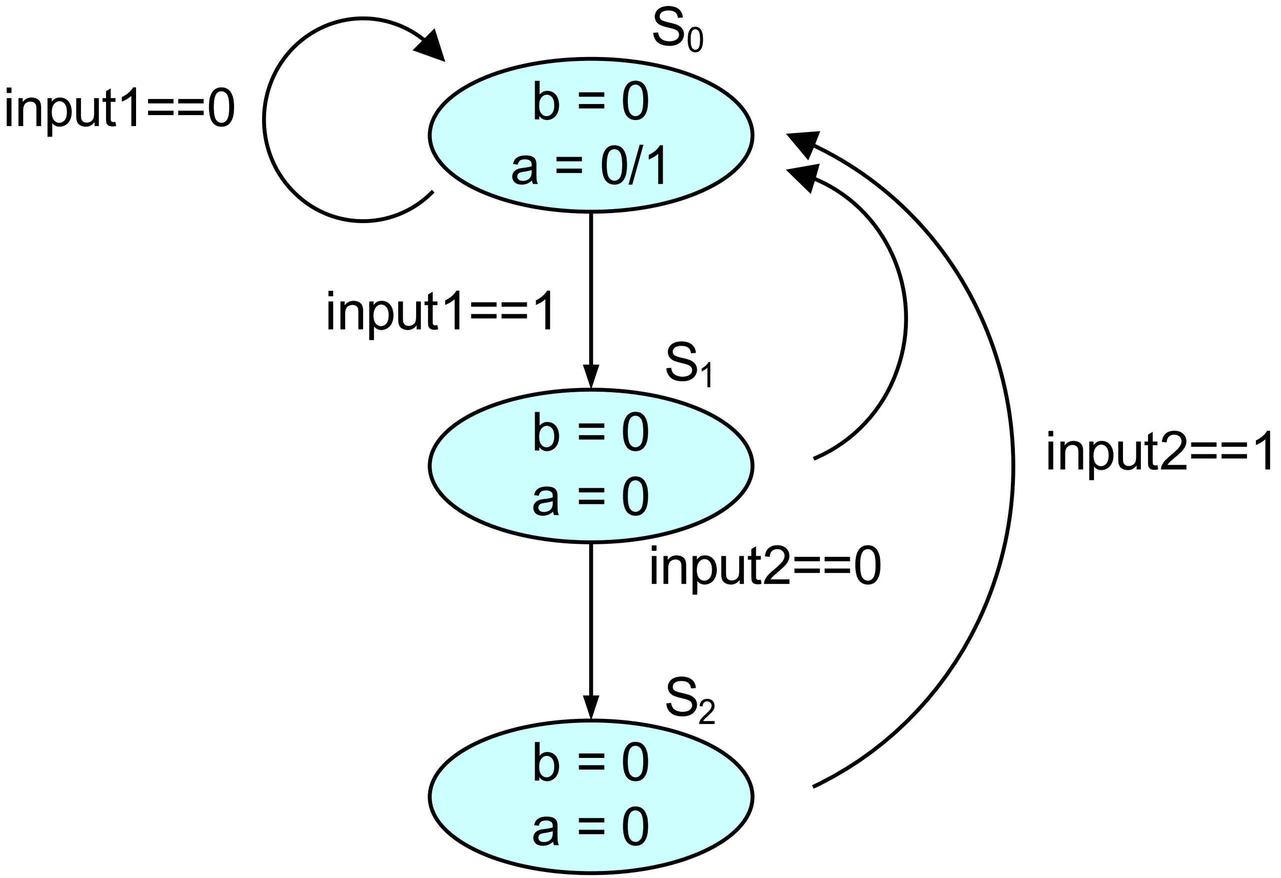
# TestBench Example: Finite State Machine

Mealy State Machine: has one sequential element -the state vector - and two combinational elements, the output logic and the next state logic.

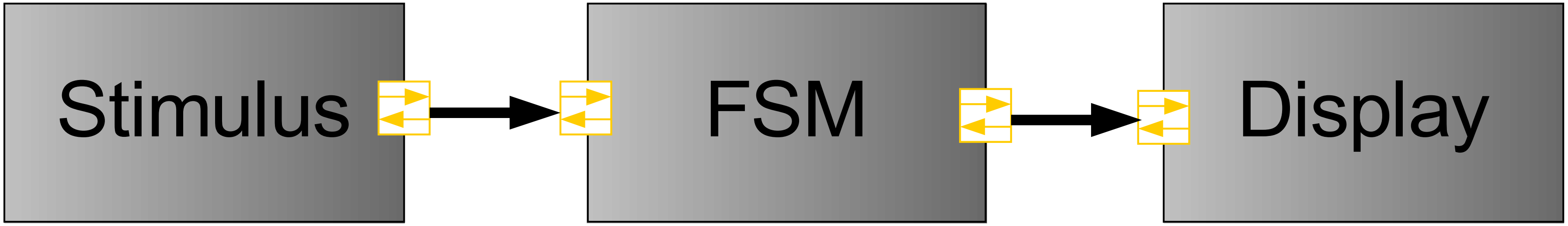


# TestBench Example: Finite State Machine

A state diagram that represents an example state machine, where a and b represent the outputs.



# TestBench Example: Finite State Machine



# TestBench Example: Finite State Machine

// ** main.cpp

```
#include <systemc.h>
#include "fsm.h"
#include "stimulus.h"
#include "display.h"
int sc_main (int argc , char *argv[])
{
        sc_clock clock;
        sc_signal<bool>  rst;
        sc_signal<bool>  in1, in2;
        sc_signal<bool>  out_a, out_b;

        stimulus
stimulus1("stimulus_block");
        stimulus1.rst(rst);
        stimulus1.input1(in1);
        stimulus1.input2(in2);
     stimulus1.clk(clock.signal());
```

SystemC 1.1?

```
fsm fsm1("MyFSM");
        fsm1.rst(rst);
        fsm1.input1(in1);
        fsm1.input2(in2);
        fsm1.a(out_a);
        fsm1.b(out_b);
        fsm1.clk(clock);

        display  display1 ("display");
        display1.a(out_a);
        display1.b(out_b);
        display1.clk(clock);

        sc_start(clock, -1); // run forever

        return 0;
}
```

# TestBench Example: Finite State Machine

**// ** fsm.h**

```
enum state_t
{ // enumerate FSM states
        S0, S1, S2
};

SC_MODULE(fsm){
        sc_in_clk clk;
        sc_in<bool> rst, input1, input2;
        sc_out<bool> a, b;
        sc_signal<state_t> state,
next_state;

        void ns_op_logic();
        void update_state();

        SC_CTOR(fsm){
                SC_METHOD(update_state);
                sensitive_pos << clk;
                SC_METHOD(ns_op_logic);
                sensitive << state << input1 <<
input2;
        }
};
```

Old style SystemC → sensitive_pos << clk;

# TestBench Example: Finite State Machine

```
#include "systemc.h"
#include "fsm.h"

void fsm::update_state() {
        if (rst.read() == true)
                state = S0;
        else
                state = next_state;
}
void fsm::ns_op_logic() {
        switch(state) {
             case S0:   b.write(0);
                        if (input1.read() ||
input2.read())
                                a.write(1);
                        else
                                a.write(0);
                        if (input1.read() == 1)
                                next_state = S1;
                        else
                                next_state = S0;
                break;
```

**// ** fsm.cpp**

```
case S1:   a.write(0);
           b.write(1);
           if (input2.read() == 1)
                   next_state = S2;
           else
                   next_state = S0;
           break;

        case S2:   a.write(0);
           b.write(0);
           next_state = S0;
           break;

        default:   a.write(0);
           b.write(0);
           next_state = S0;
           break;
        }
}
```

# TestBench Example: Finite State Machine

**stimulus.h**

```
    SC_MODULE(stimulus) {

 sc_out<bool> rst;
 sc_out<bool> input1, input2;
 sc_in<bool>  clk;

 unsigned cycle;

 SC_CTOR(stimulus)
 {
     SC_METHOD(entry);
     dont_initialize();
     sensitive_pos(clk);
     cycle      = 0;
 }
 void entry();
};
```

Old style SystemC

# TestBench Example: Finite State Machine

**stimulus.cpp**

```
#include <systemc.h>
#include "stimulus.h“
void stimulus::entry() {

  cycle++;

  if (cycle<4) {
    rst.write(true);
        input1.write(false);
        input2.write(false);
  } else {
    rst.write(false);
        if (cycle%4==0) {
                input1.write(true);
                input2.write(true);
```

```
        } else if(cycle%4==1) {
                input1.write(true);
                input2.write(false);
        } else if(cycle%4==2) {
                input1.write(false);
                input2.write(true);
        } else if(cycle%4==3) {
                input1.write(false);
                input2.write(false);
        }
  }
```

# TestBench Example: Finite State Machine

**display.h**

```
SC_MODULE(display) {

  sc_in<bool> a, b;
  sc_in<bool>  clk;

  unsigned i;


  SC_CTOR(display)
   {
     SC_METHOD(entry);
     dont_initialize();
     sensitive_pos(clk);
     i = 0;
   }

  void entry();
};
```

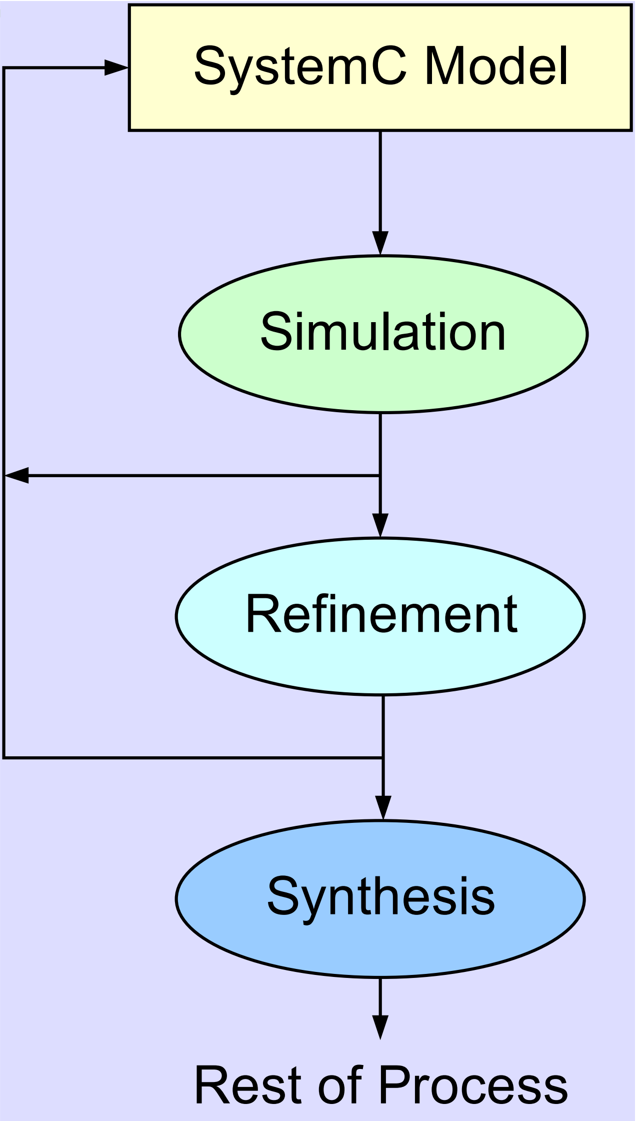
← Old style SystemC

# TestBench Example: Finite State Machine

**display.cpp**

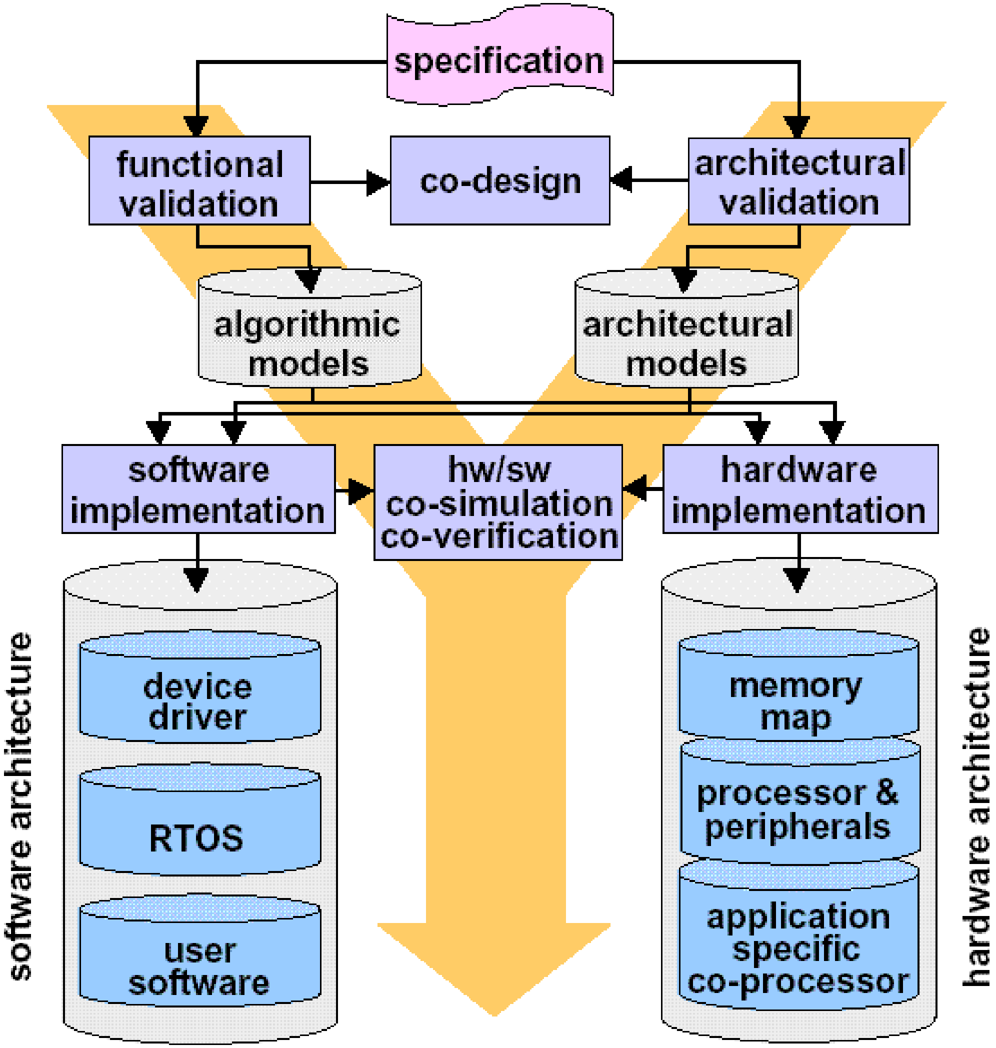
```
#include <systemc.h>
#include "display.h"

void display::entry(){

  cout<< "Display : " << "a=" << (int)a.read()
          << " b=" << (int)b.read() << " at time "
          << sc_simulation_time() << endl;
  i++;
  if(i == 24) {
    cout << "Simulation of " << i << " clock period"
                 << " at time " << sc_simulation_time() << endl;
    sc_stop();
  };
}
```

# Possible SystemC Design Flow



# Possible SystemC Design Flow (2)

Single specification should be partitioned into HW- & SW-parts. HW & SW-submodels should be generated.



# Transaction level modeling

Models in other HW-description languages typically comprise clocking, and simulation is said to be **cycle-true**, referring to clock cycles of circuits described at the **register-transfer level** (RTL).

The gas station example does not use any clocks, it is based on the communication of **events**.

Such models are called **transaction-level** models (TLM).

TLM models typically simulate significantly **faster** than RTL models, due to the abstraction used.

Consequently, industry is moving towards TLM for complex circuits.

# Functional Modeling &TLM

***Untimed functional models:*** used to describe & analyze DSP functionality such as filters, decoders, or demodulators. Effective model of computation: *dataflow modeling*.

***Timed functional models:***

- required if either modeling temporal relationships needed to analyze the system behavior
- or if the functional model used as placeholder for its to-be-designed implementation.

Introducing time into untimed functional models can be done by adding **wait(sc_time)** statements

# Untimed Functional Model

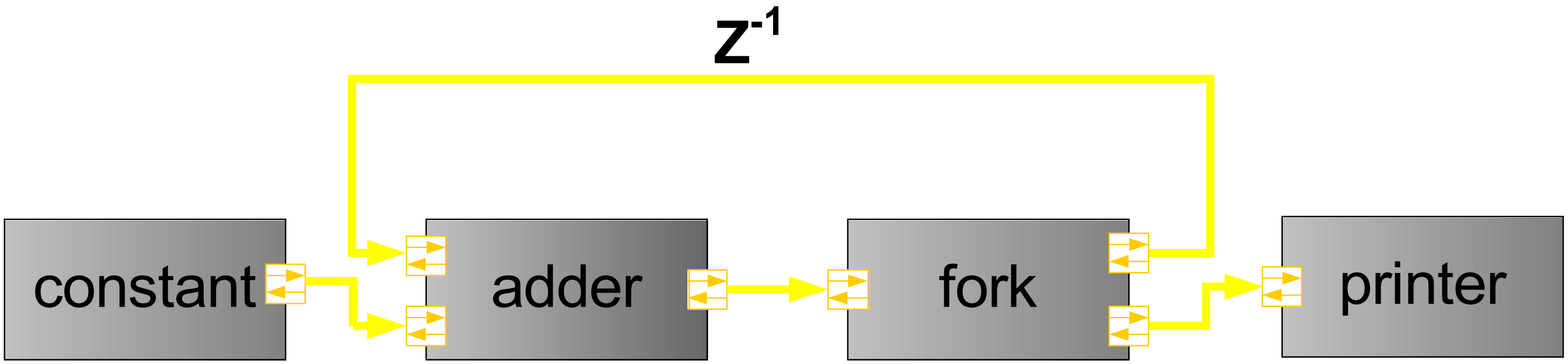
- Design is specified in terms of its functional components
- Functional components specify *possible* structural boundaries of final implementation
- Some functional components might be merged while other might become discrete cores

# Untimed Functional Model

Dataflow MoC is the most common form of specification

- Alternatively Kahn process networks, Multi-rate dataflow
- Communication between components is through directed point-to-point FIFO
- FIFO are bounded
- Blocking read, blocking write
- Time delays might exist in the model at the boundaries
- Processes in which the model interacts with the environment
- Time delays denote constraints for the system

# Functional Modeling - Example



```
template <class T>
SC_MODULE(DF_Const){
sc_fifo_out<T> output;
T constant_;
void process() { while(1)
output.write(constant_); }
SC_HAS_PROCESS(DF_Const);
DF_Const(sc_module_name N, const T& C):
sc_module(N), constant_(C) {
SC_THREAD(process); }
};
```

# Functional Modeling - Example

```
template <class T>
SC_MODULE(DF_Adder){
        sc_fifo_in<T> input1, input2;
        sc_fifo_out<T> output;
        sc_fifo_out<T> output2;
        void process() {  while(1){
           output.write(input1.read() +
input2.read()); } }
        SC_CTOR(DF_Fork) {
SC_THREAD(process); }
};
```

```
template <class T> SC_MODULE(DF_Fork){
sc_fifo_in<T> input;
sc_fifo_out<T> output1;
sc_fifo_out<T> output2;
void process() {
while(1) {
T value = input.read();
output1.write(value);
output2.write(value);
}
}
SC_CTOR(DF_Fork) {
SC_THREAD(process); }
};
```

# Timed Functional Model - Example

```
template <class T> SC_MODULE(DF_Const){
sc_fifo_out<T> output;
T constant_;
void process() { while(1)
        { wait(200, SC_NS); output.write(constant_);} }
SC_HAS_PROCESS(DF_Const);
DF_Const(sc_module_name N, const T& C):
sc_module(N), constant_(C) { SC_THREAD(process); }
};
```

```
template <class T> SC_MODULE(DF_Adder){
sc_fifo_in<T> input1, input2;
sc_fifo_out<T> output;
sc_fifo_out<T> output2;
void process() {  while(1) {
        output.write(input1.read() + input2.read()); wait(200,
SC_NS);}  }
SC_CTOR(DF_Fork) { SC_THREAD(process); }
};
```

# Register-transfer level modeling

**"Register-transfer level“:**
synchronous transfer between functional units, such as multipliers and ALUs, and register files, synchronized by a **clock.**

- For processors, functional units, register files, and the connection between them are called **data-path**.
- Transfers in data-path are coordinated by a controller.

# Register-transfer level modeling (2)

Characterized by the kind of channels, ports, and processes

RTL describes cycle-by-cycle behavior

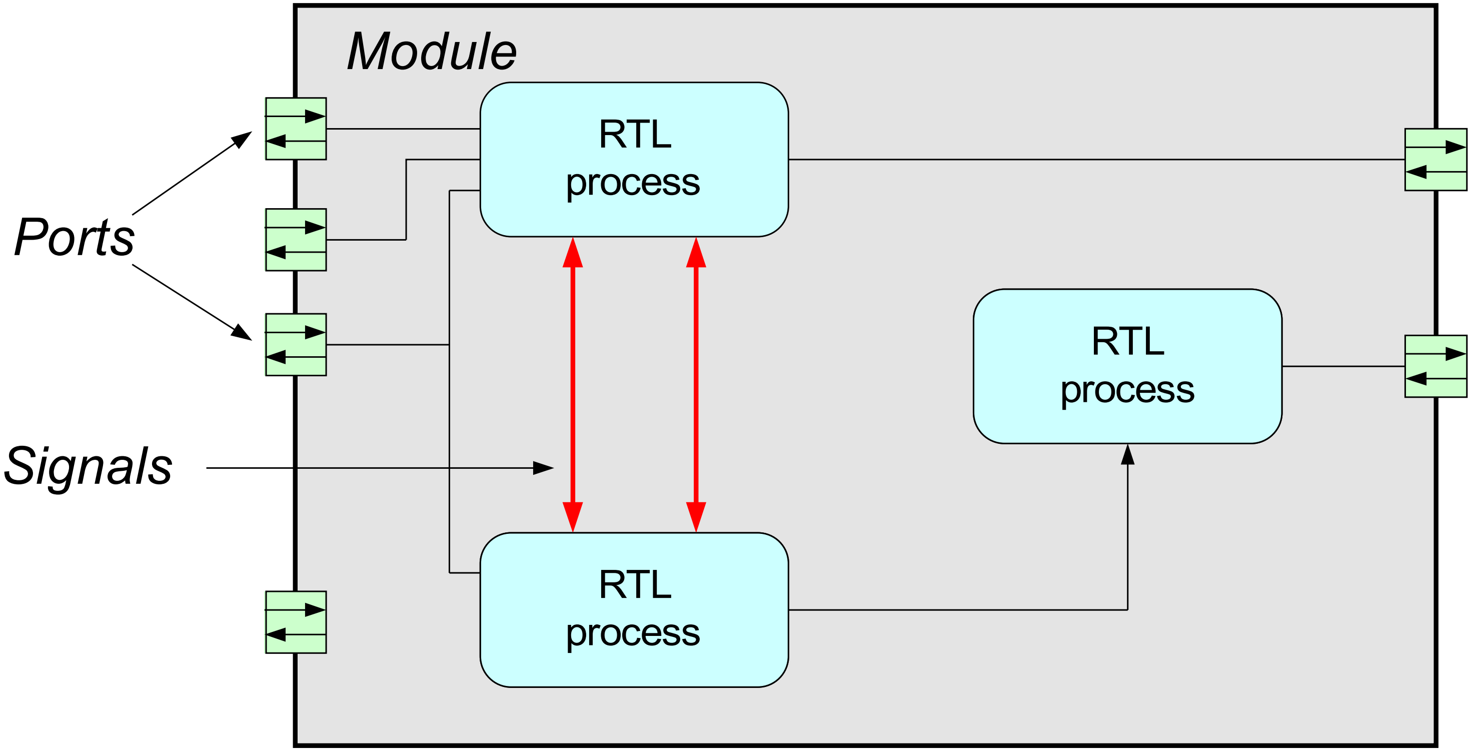
→ *channels* restricted to plain and resolved signals,
no complex temporal behavior.

Only *port* types **sc_in**, **sc_out**, and **sc_inout**
(corresponding to **sc_signal**'s interfaces),
and resolved port types.

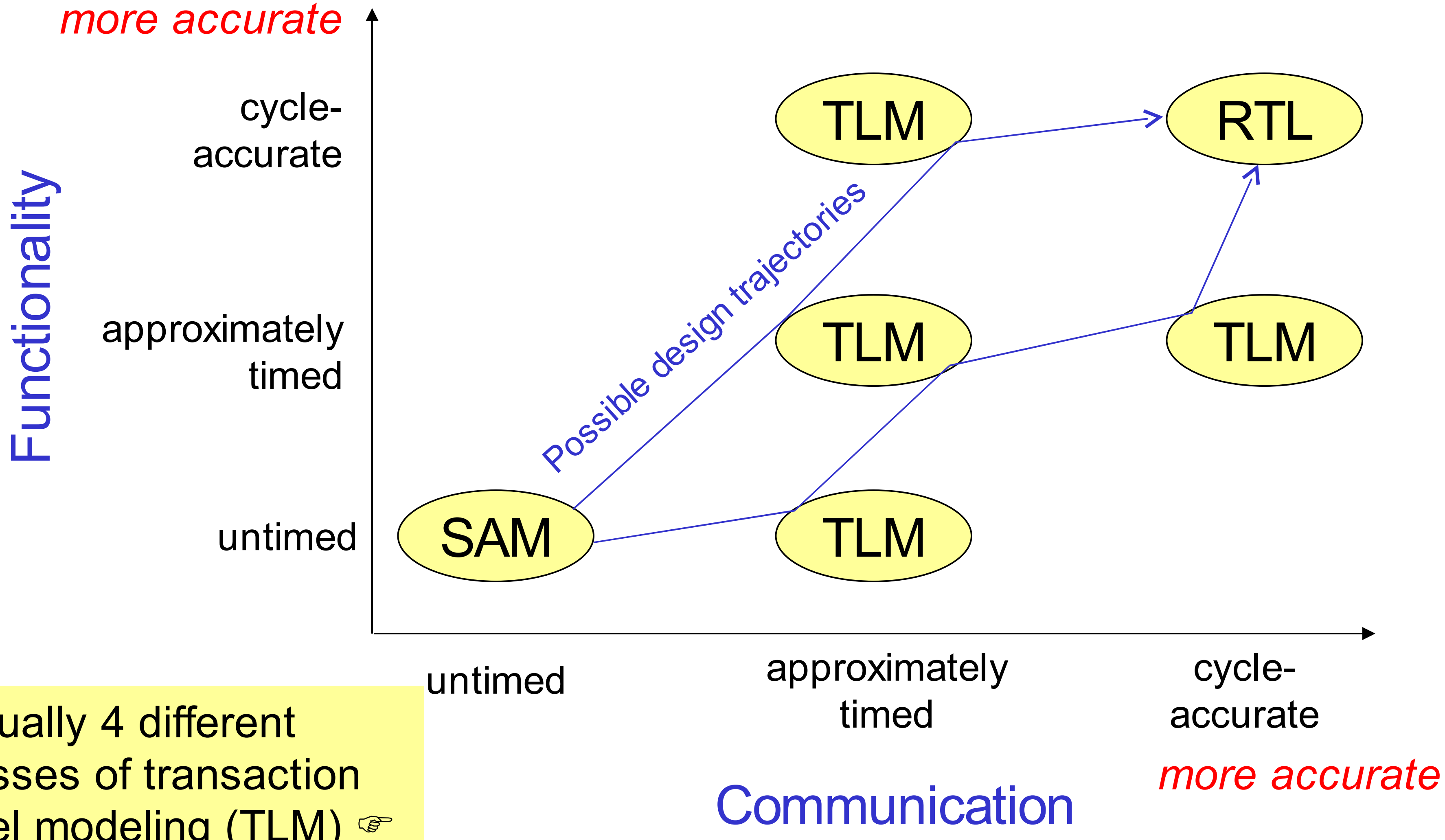
Each RTL *process* is captured by an **SC_METHOD**.

# General structure of an RTL module

- ≥1 RTL processes ∈ module.
- Process ≅ typical RT component.
- All sequential components are clocked.
- Communication is via signals (intern.) or ports (extern.).



# Modeling levels: classification according to Gajski (Codes/ISSS 2003)



Actually 4 different classes of transaction level modeling (TLM) ☞

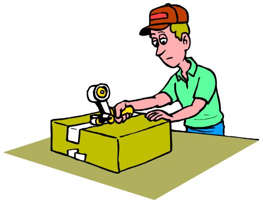
# Classification according to Gajski (Codes/ISSS 2003)

| **Model** | **Communication** | **Functionality** |
|---|---|---|
| SAM (system architectural model) | UT | UT |
| Component assembly | UT | AT |
| Bus arbitration | AT | AT |
| Bus functional | CT | AT |
| Cycle accurate computation | AT | CT |
| RTL | CT | CT |

# Advantages and limitations of SystemC

**Pros:**

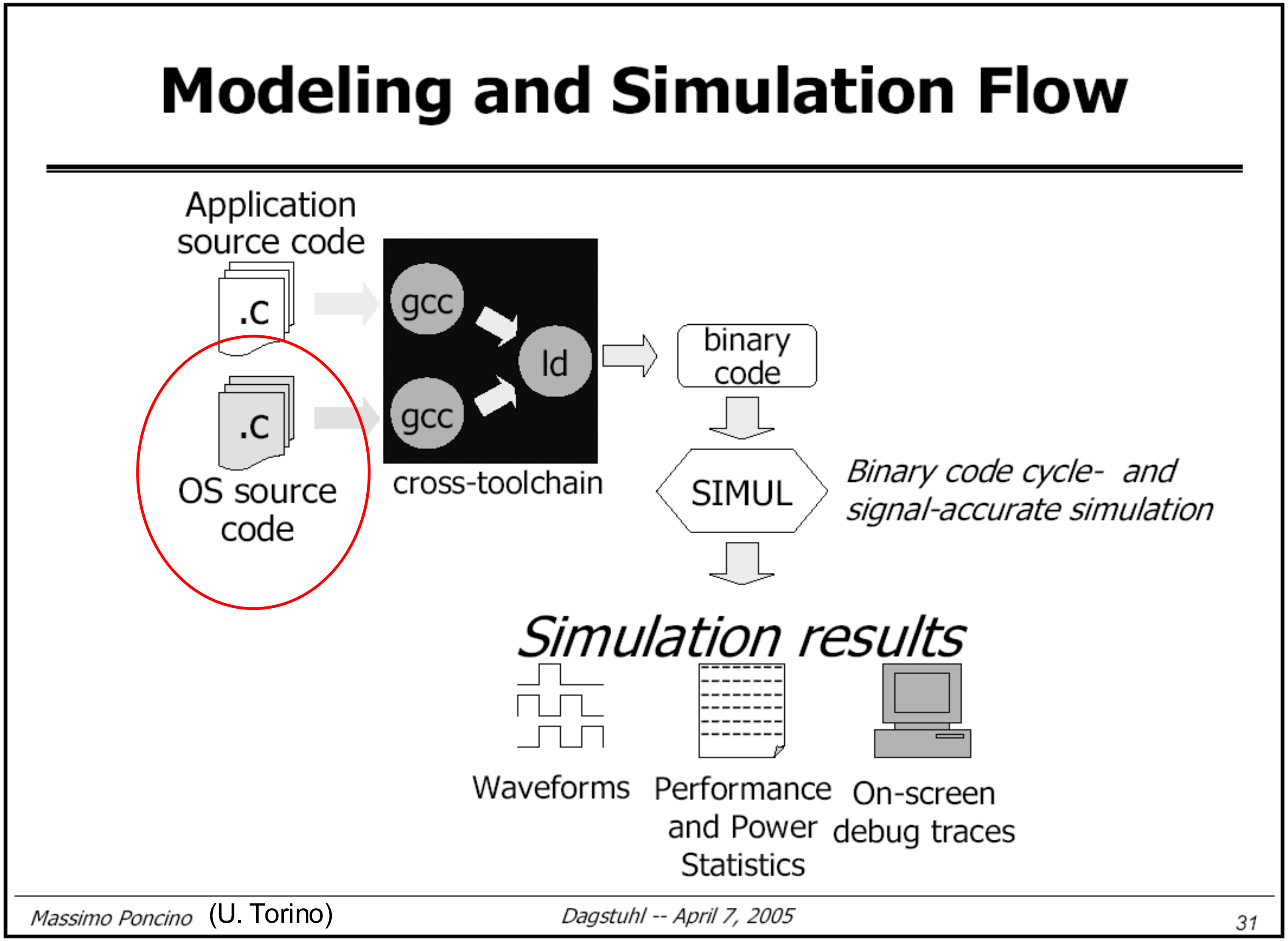
- Enables focus on **modeling** (essential for CS) at a high level.
- Enables development of wrappers around standard algorithms described in C (mpeg, gsm, …)



- Enables path from C to HW, replacing vendor-specific languages from Celoxica (Handel-C) or from CoWare.
- Avoids development of interfaces between HW & SW simulators (such as VHDL/C links).
- High simulation speed is possible with TLM models.
- Full power of OO programming available for HW design.
- Standard libraries such as STL or Boost can be used.

# "Seamless" interface to middleware and OS

e.g.: OS is included in MPARM SystemC-based multi-processor simulation framework from U. Bologna (Benini et al.)



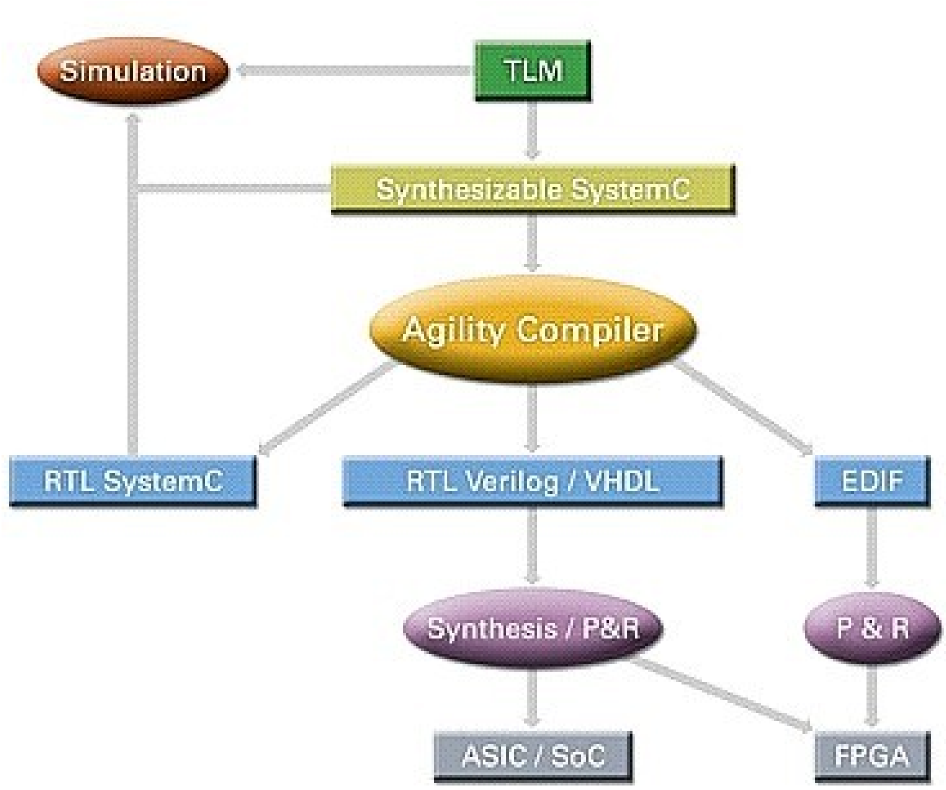
## Modeling and Simulation Flow

*Massimo Poncino* (U. Torino) *Dagstuhl -- April 7, 2005* *31*

# Advantages and limitations of SystemC

First synthesis tools from SystemC start to exist:

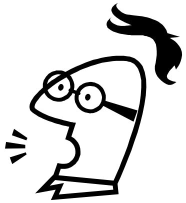
- Cynthesizer (http://www.forteds.com/products/cynthesizer.asp)
- Agility (http://www.celoxica.com/products/agility/default.asp)



# Advantages and limitations of SystemC

**Cons:**

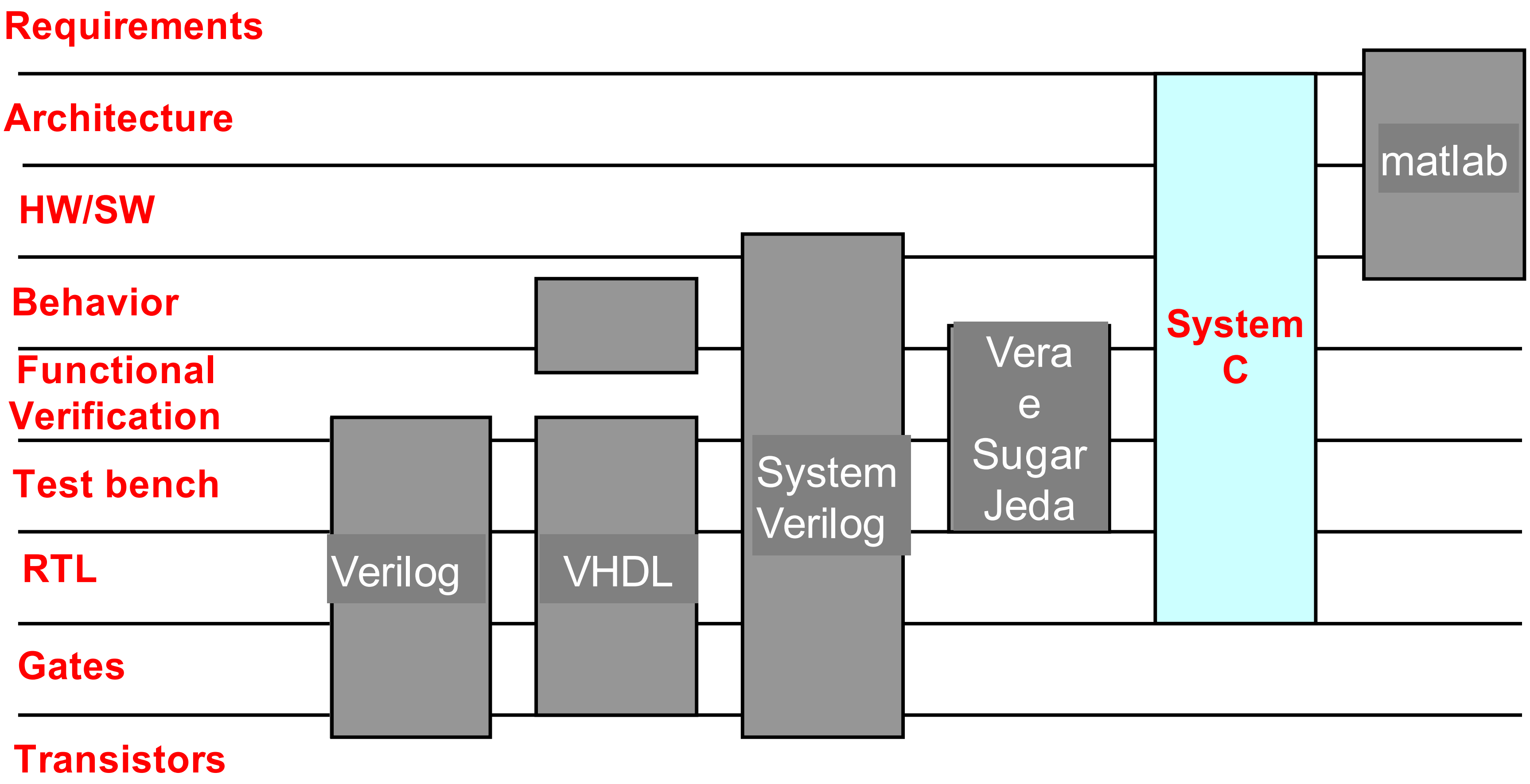
- Mature synthesis technology?
- Semantics definition simulation-oriented.
- Several restrictions due to being based on C++, e.g.
  - Event finders
  - Varying syntax for structural descriptions
  - Resolution functions difficult to use.
  - Error reporting, …
- Difficult to use for a hardware engineer.



# Timeline development of SystemC

| Date | Version | Notes |
|---|---|---|
| Sept 1999 | 0.9 | First version; cycle-based |
| Feb 2000 | 0.91 | Bug fixes |
| March 2000 | 1.0 | Widely accessed major release |
| Oct 2000 | 1.01 | Bug fixes |
| Feb 2001 | 1.2 | Various improvements |
| Aug 2001 | 2.0 | Add channels & events; cleaner syntax |
| Apr 2002 | 2.01 | Bug fixes; widely used |
| June 2003 | 2.01 | LRM in review |
| Spring 2004 | 2.1 | LRM submitted for IEEE standard |
| Dec. 2005 | 2.1 | SystemC accepted as IEEE Standard 1666-2005 |
| June 2006 | 2.2 | Draft & proof of concept |
| June 2006 | | Synthesizable subset |

# SystemC contrasted with other design languages



# Conclusion

- Dynamic processes
- Test benches
- Modeling styles
  - transaction level modeling
  - untimed model
  - timed model
  - RTL model
- Evaluation and comparison of SystemC