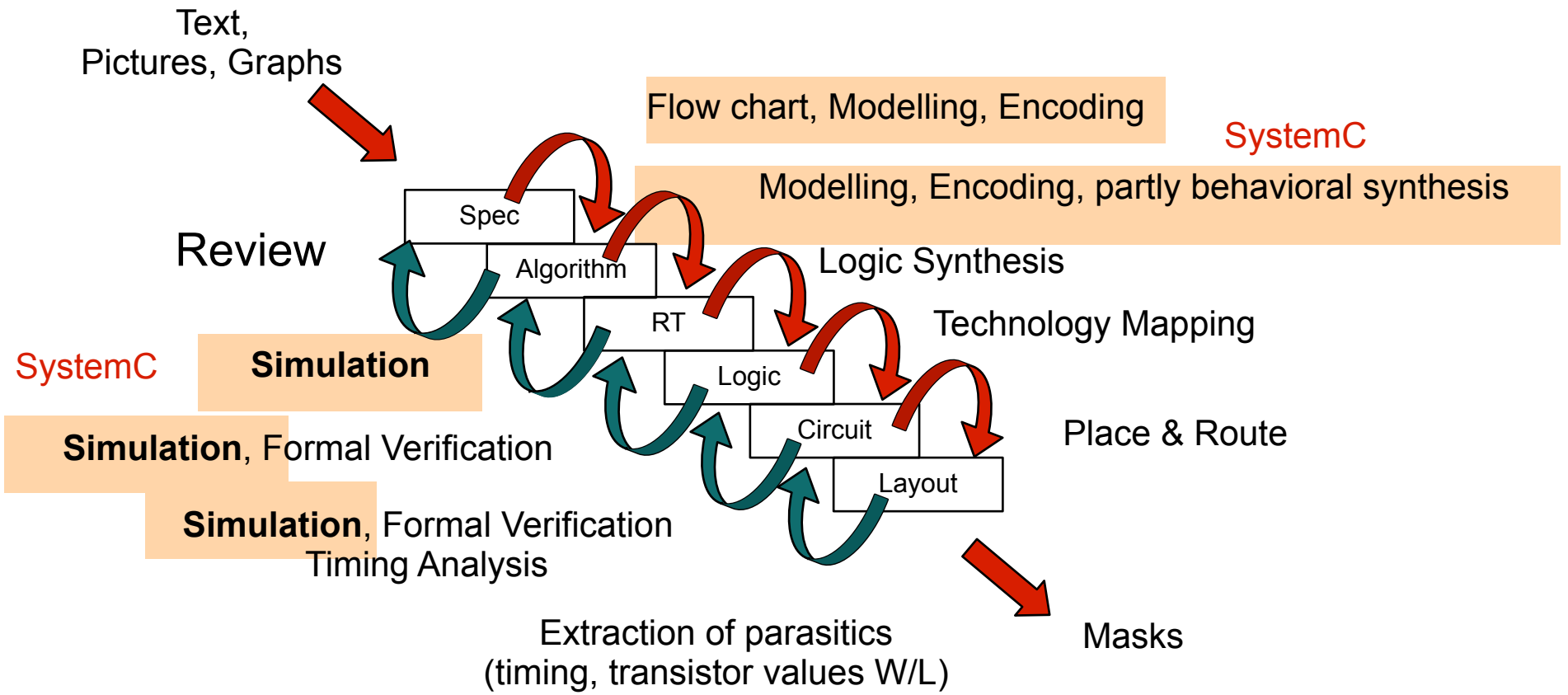
Chair for Circuit and System Design

Modeling and Simulation of Digital Systems with SystemC

CHEMNITZ UNIVERSITY OF TECHNOLOGY

Waterfall Model: Prove of Correctness



Overview

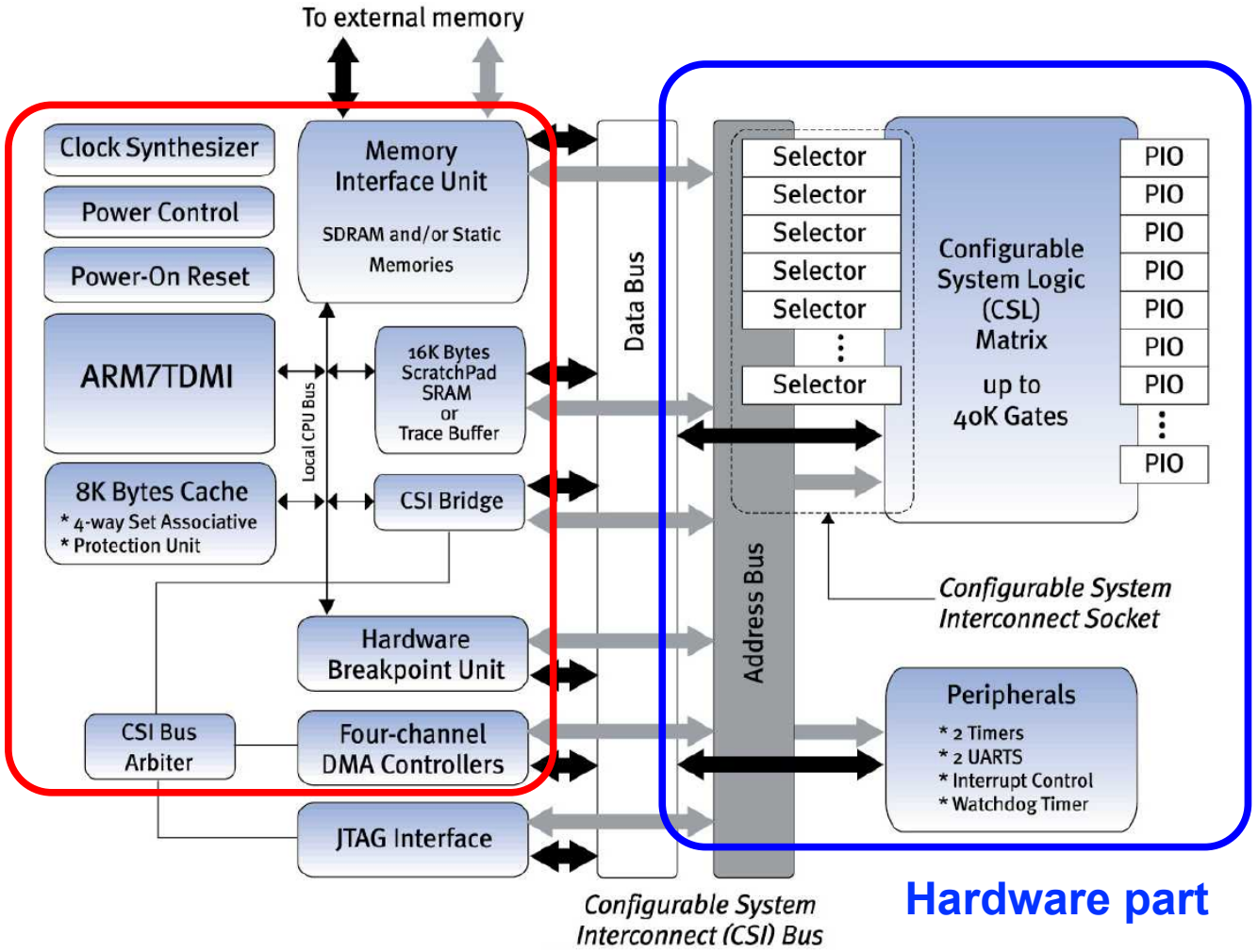
- SystemC
  - Introduction to SystemC
    - Challenges of HW/SW Co-Design
  - Modelling RT Level
    - Structural Modelling: Modules, Ports and Signals
    - Behavioural modelling: Processes and Sensitivity
    - Structure of a SystemC Model
  - SystemC Simulation kernel
  - High Level Modelling on System Level
    - Interfaces, Channels and High Level Communication
    - Events and dynamic sensitivity

Information & Material

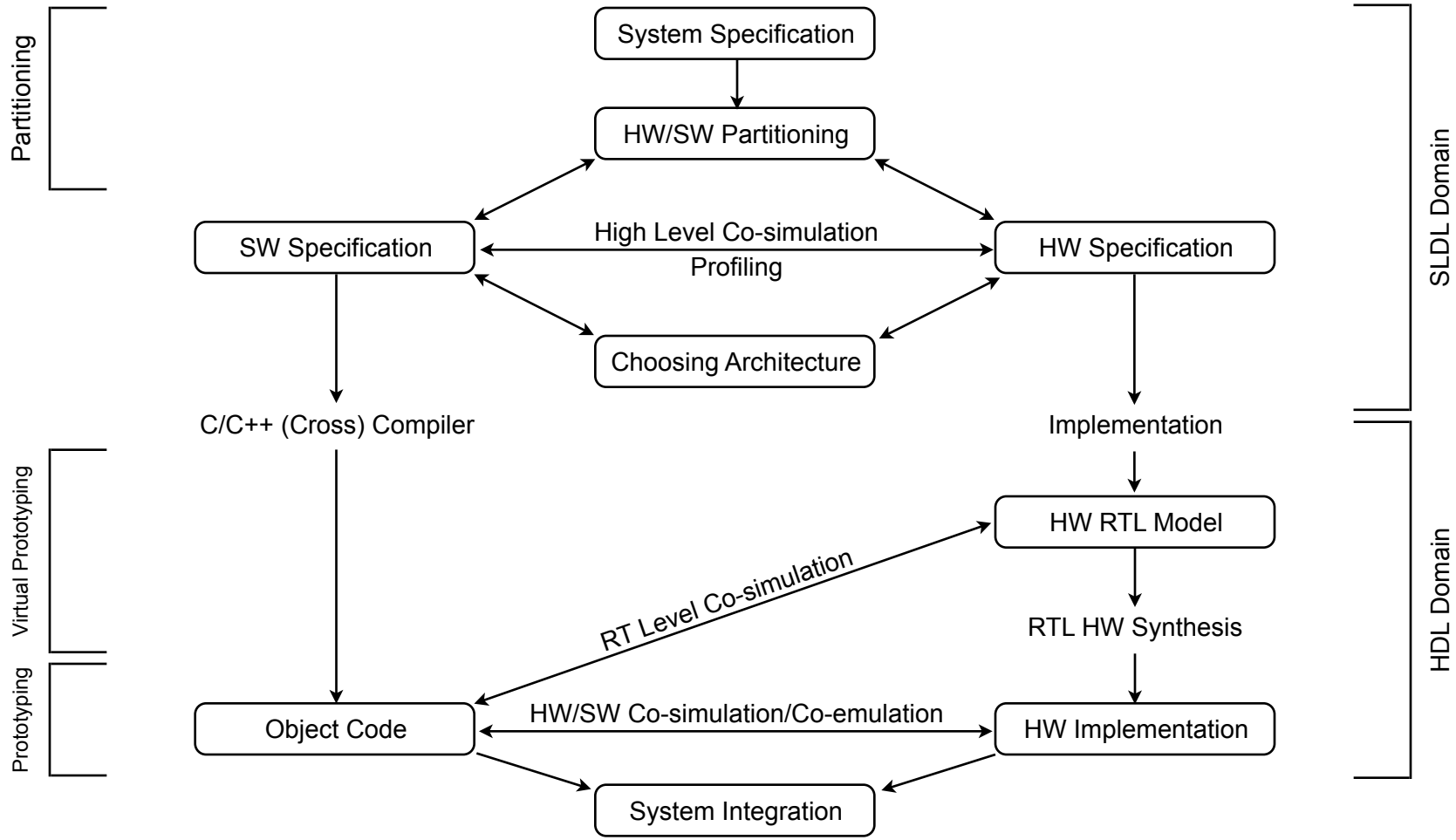
- SystemC Package
  - Source Code (Reference Implementation of Library and Simulation kernel)
  - Documentation (Reference Manual, User Guide, Functional Specification, White Paper)
  - Collection of Examples
- www.systemc.org
- Literature
  - System Design with SystemC - Groetker, Liao, Martin, Swan
  - SystemC: From The Ground Up - Black, Donovan
  - Modellierung von digitalen System mit SystemC - Kesel (German only)
  - Discussion Forum at www.systemc.org

Architecture of digital HW/SW Systems

Triscent
A7 CSoC Family



Design flow of digital HW/SW systems



What is SystemC?

- SystemC is a C++ based System specification and modelling language
  - System Level Description Language (SLDL)
  - Modelling hardware and software on different abstraction levels of the design flow
- Building Software components in C/C++ (trivial)
- How can Hardware components be modelled in C++?
  - HW is concurrent
  - HW has timing
  - HW is reactive

} C++ is a Programming languages (and does not know these concepts)

- Hardware „objects“ are realized by pre-defined C++ classes, which, in combination with a simulation kernel, model the above mentioned hardware characteristics

Composition of SystemC language elements

Standard Channel Models for different Models of Computation

- Synchronous Data Flow
- Kahn Process Networks
- CSP
...

Channel Models specific to particular methods

- Master/Slave Library
- Protocol Stack

Basic Channels for Process Communication and Synchronization

Signals, FIFOs, MUTEXs, Semaphores, ...

SystemC „Core“ Elements

- Modules, Ports, Processes
- Interfaces, Channels
- Events

SystemC Data Types

- Logic, Logic Vector (01XZ)
- Bit, Bit Vector
- Integer (Fixed Point, ...)

C++ Language Standard

RT Level Modelling with SystemC

- Modules, Signals and Ports
- Processes and Sensitivity
- Structure of a SystemC Model
- Compilation and Execution
- Displaying and Tracing Simulation results/data

Modules

- Modules encapsulate functionality

```
// 4 bit counter
SC_MODULE(counter) {

   // Ports
   sc_in_clk clk;            // Clock
   sc_in<bool> reset;        // Reset
   sc_in<bool> ce;           // Count Enable
   sc_out<sc_uint<4> > cnt; // Counter output

   // local counter variable
   sc_uint<4> counter;

   SC_CTOR(counter) {
      ...
   }
};
```

Signals and Ports

- On RT Level, modules communicate using signals and ports
- Declaration:

```
sc_signal<int> s1;
sc_in<bool> p1; sc_out<char> p2; sc_inout<my_type> p3;
```

- Described by a name (*s1*) and data type (*int*)
- Ports additionally have a direction (*in/out/inout*)
- Data types can be:
  - SystemC data type (*sc_uint<N>, sc_logic, sc_lv<N>, ...*)
  - Elementary C++ data type (*int, bool, float, char, ...*)
  - User defined data type (*class XType { char x; sc_uint<3> y; }*)
- SystemC signals are handled like VHDL signals:
  - The values of all respective signals are updated at the same time at the end of a simulation delta cycle

Processes (I)

- **Processes are used to describe the (reactive) behaviour of modules and are executed (virtually) in parallel**
- **3 types of processes:**

SC_METHOD	SC_THREAD	SC_CTHREAD
sensitive to alteration of arbitrary signals	sensitive to alteration of arbitrary signals	sensitive to alteration of a Clock signal
activated each time a signal in sensitivity list changes	only activated once	only activated once
once activated, a method is processed until its end	once activated, a thread is processed until the next **wait()** statement and suspended	once activated, a thread is processed until the next **wait()** statement and suspended
no wait() statements allowed inside the process	can contain **wait()** statements	can contain **wait()** statements

Processes (II)

- Processes are defined as C++ Methods of an SC_MODULE:

  `void process_name() { // process code }`

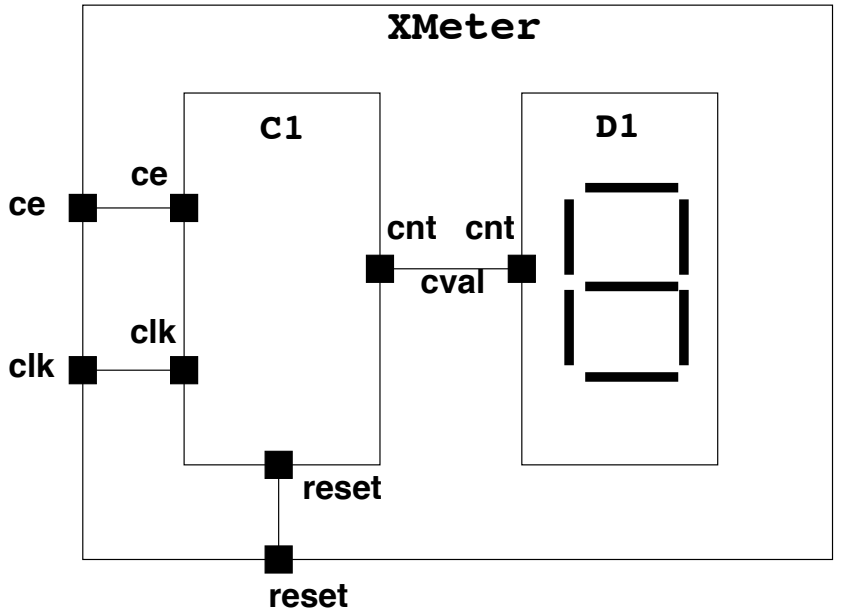
  and registered as process of a certain type inside the module's constructor:

```
SC_MODULE(counter) {

   ...

   void incr (); // process

   SC_CTOR(counter) {
      SC_METHOD (incr);
      sensitive << reset.neg();
      sensitive << clk.pos();
   }
};
```

```
#include „counter.h“

void counter::incr () {
   if ( ! reset.read())
      counter = 0;
   else if (ce.read())
      counter++;
};
```

Description of hierarchical netlists

```
SC_MODULE(XMeter) {

   sc_in_clk clk;
   sc_in<bool> ce, reset;

   counter* C1;
   display* D1;
   sc_signal<sc_uint<4> > cval;

   SC_CTOR(XMeter){
      C1 = new counter(„C1“);
      D1 = new display(„D1“);

      C1->clk(clk);
      C1->reset(reset);
      C1->ce(ce);
      C1->cnt(cval);
      D1->cnt(cval);
   }
};
```



Inside the module constructor

- instantiated submodules are allocated
- static binding of ports to signals/ports

Structure of a SystemC Model

Module Declaration + **Module Implementation**

Module1.h

```
#include „systemc.h“

SC_MODULE(Module1) {
   ...
};
```

Module1.cc

```
#include „Module1.h“

void Module1::process () {
   ...
};
```

Module Specification

main.cc

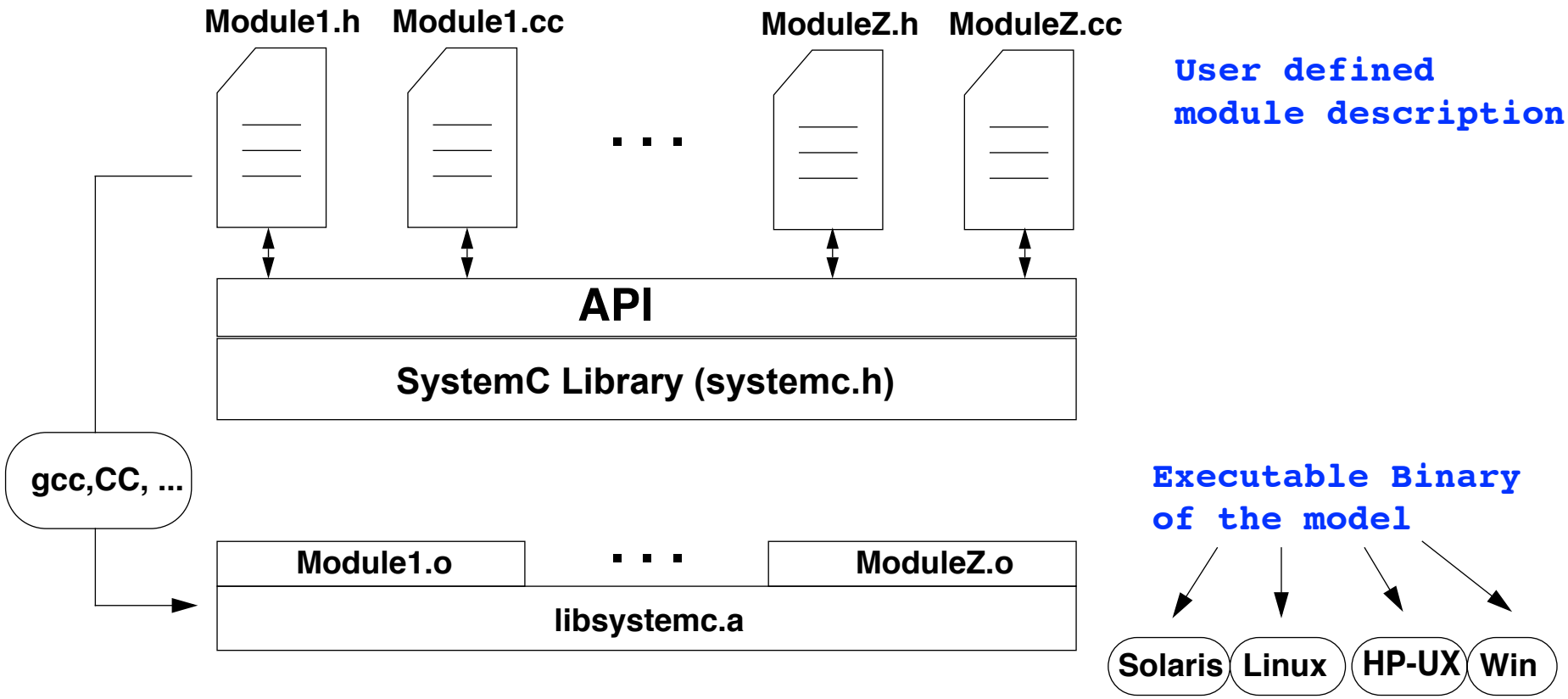
```
#include „Module1.h“

int sc_main (int argc, char* argv[]) {
   Module1 m1 („M1“);
   ...
   sc_start (10000, SC_NS);
   return 0;
};
```

Module Instantiation

Starting Simulation Kernel

Structure of a SystemC Model (II)

- A SystemC Model is a C++ Program which makes use of the SystemC class library



Compilation and Execution of a SystemC Model

```
# Compiling Modules and main.cc

g++ -fpic -I$SYSTEMC/include -c -o Module1.o Module1.cc
g++ -fpic -I$SYSTEMC/include -c -o Module2.o Module2.cc
...
g++ -fpic -I$SYSTEMC/include -c -o ModuleN.o ModuleN.cc
g++ -fpic -I$SYSTEMC/include -c -o main.o main.cc

# Linking Modules and main (against SystemC lib)

g++ -static -fpic -L$SYSTEMC/lib-linux -lsystemc -o model \
Module1.o Module2.o ... ModuleN.o main.o

# Executing model

./model
```

Displaying and Tracing Simulation results/data

- Simulation information can be written to Standard Output / Shell:

```
std::cout << "Starting Testbench Process" << std::endl;
```

- Simulation time and Module name can be accessed from within simulation

```
std::cout << "Module name: " << name() <<
"\nSimulation time: " << sc_time_stamp() << std::endl;
```

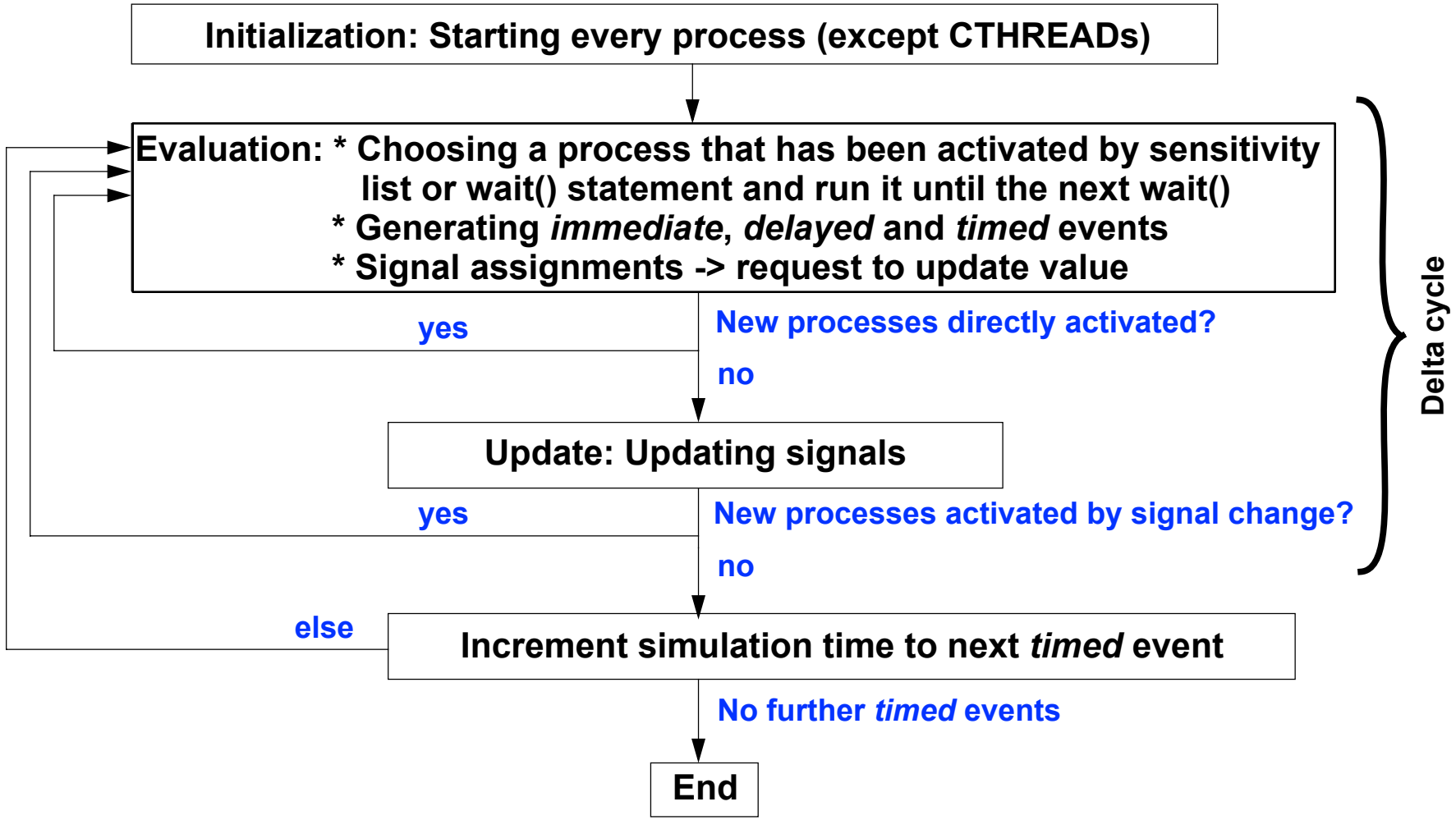
- Trace files can be generated and signals to be traced can be specified

```
sc_trace_file *tf = sc_create_vcd_trace_file ("VCDfile");
sc_trace (tf, tb.reset_s, "reset_s");
```

- Tracing functions can be generated specific to modules

```
void sc_trace (sc_trace_file *, const Module1 & );
```

SystemC Simulation Kernel

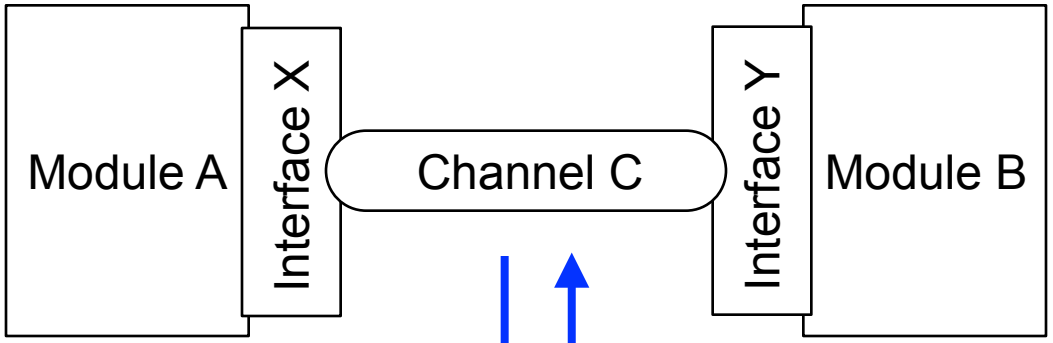


High Level Modelling using SystemC

- Communication methods so far:
  - Modules communicate using signals and ports, signals correspond directly to electrical connections
  - Process synchronization using signals
- Behavioural description so far:
  - Number and kind of processes is static (fixed at compile time)
  - Sensitivity of processes is static (fixed at compile time)
- Extended communication and behavioural description:
  - Abstraction of signals to channels
  - Dynamic sensitivity, Events

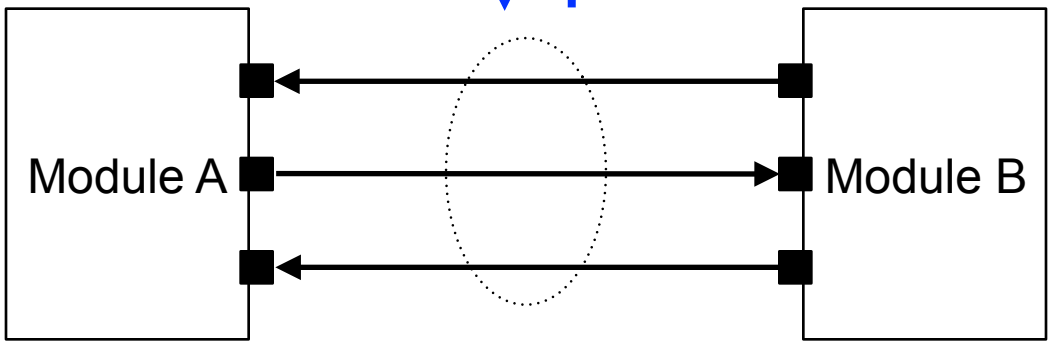
Interfaces and Channels

High Level Model



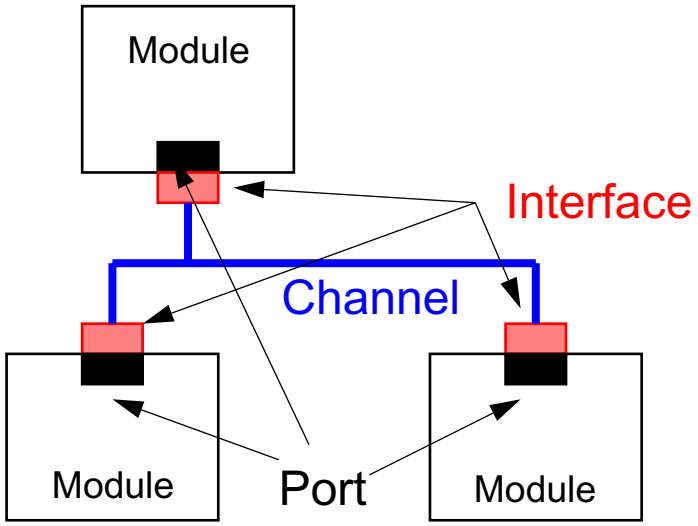
Refinement **Abstraction**

RT Level Model



Interfaces, Channels and Ports

- **Interface:**
  - declares interface methods, does not define them
- **Channel:**
  - implements one or more Interfaces, i.e. defines the functions declared by interface
- **Port:**
  - connects Channel to Module, is of type Interface
  - can only be used with Channels implementing this Interface
  - Module calls Interface function of Channel via Port



IMC (Interface Method Call): Process calls Interface Method of Channel

Interfaces

- *Interface* defines methods to access a *Channel*.
- Interfaces are derived from SystemC base class sc_interface

```
template <class T> class read_if : virtual public sc_interface {
   public:
      virtual bool read (T&) = 0;
};
```

```
template <class T> class write_if : virtual public sc_interface {
   public:
      virtual bool write (const T&) = 0;
};
```

```
// Combine several interfaces in a single interface
template <class T> class read_write_if :
   public read_if<T>,
   public write_if<T> {};
```

Channels

- Channels are communication objects, connect Modules
- Channels implement one or more Interfaces

```
// Channel implementing read_write_if
template <class T> class my_channel :
   public read_write_if<T>,      // derive from interface
   public sc_channel {           // SystemC Channel base class
   T data;                       // data stored in Channel
   bool full;                    // no storage space available
   public:
   bool read (T& out) {
      if (full) { out=data; full=false; return true; }
      else return false;
   }
   bool write (const T& in) {
      if (full) return false;
      else { data=in; full=true; return true; }
   }
   SC_CTOR (my_channel) {
      full = false;
   }
};
```

Channels and Ports

- Modules access the Interfaces of a Channel via Ports.

```
SC_MODULE (A) {

   sc_port <write_if<int> > p;
   ...

   void send () {     //process
      int data = 42;
      while (!p->write (data))
         wait ();
   }
};
```

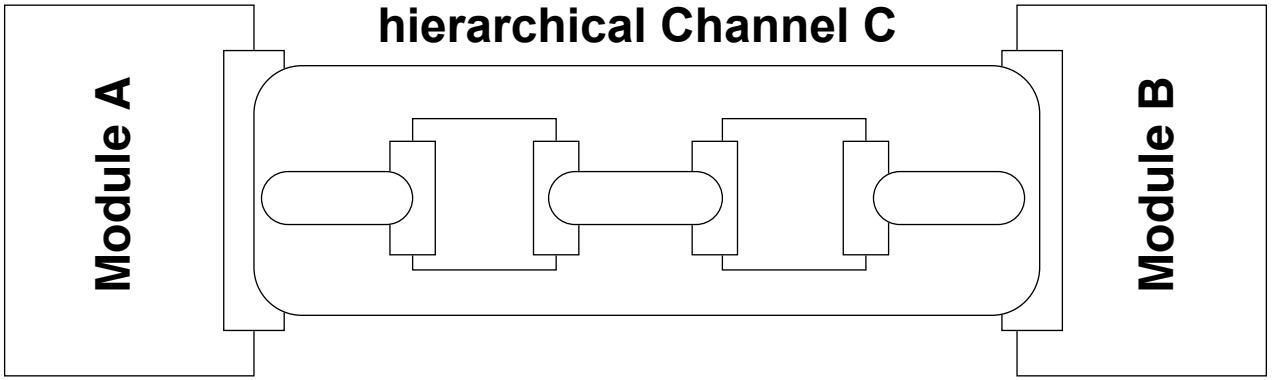
```
SC_MODULE (B) {

   sc_port <read_if<int> > p;
   ...

   void receive () {     //process
      int data;
      while (!p->read (data))
         wait ();
   }
};
```

```
my_channel<int> c("c");
A a("a");
B b("b");

a.p (c);
b.p (c);
```

Channels and Ports have to use the same Interface (i.e. be of the same type)

Primitive and hierarchical Channels

- In SystemC there are primitive and hierarchical Channels.
- Primitive Channels have no inner structure
  - sc_signal<T>
  - sc_fifo<T>
  - my_channel<T>
- Hierarchical Channels have an inner structure
  - composed of Modules and Channels



Dynamic sensitivity

- Processes can be made sensitive to the occurrence of events during runtime
- Events are objects and have to be defined

```
sc_event my_event; // usually an element of SC_MODULE
```

- Events are used with wait() statements

```
wait(my_event);    // Process suspended until 'my_event' occurs
```

- Events are generated inside of other processes

```
my_event.notify();           // Event occurs instantly
my_event.notify(10, SC_NS); // Event occurs after 10ns
```

- Events can be deleted (if time of notification not yet reached)

```
my_event.cancel();
```

Events and synchronization

```
// Channel implementing BLOCKING read_write_if
template <class T> class my_channel_blocking :
   ...
   sc_event data_written;        // event for „data written“
   sc_event data_read;           // event for „data read“
   public:
   bool read (T& out) {
      if (!full) wait (data_written);
      out=data;
      full=false;
      data_read.notify();
      return true;
   }
   bool write (const T& in) {
      if (full) wait (data_read);
      data=in;
      full=true;
      data_written.notify();
      return true; }
   }
   ...
};
```

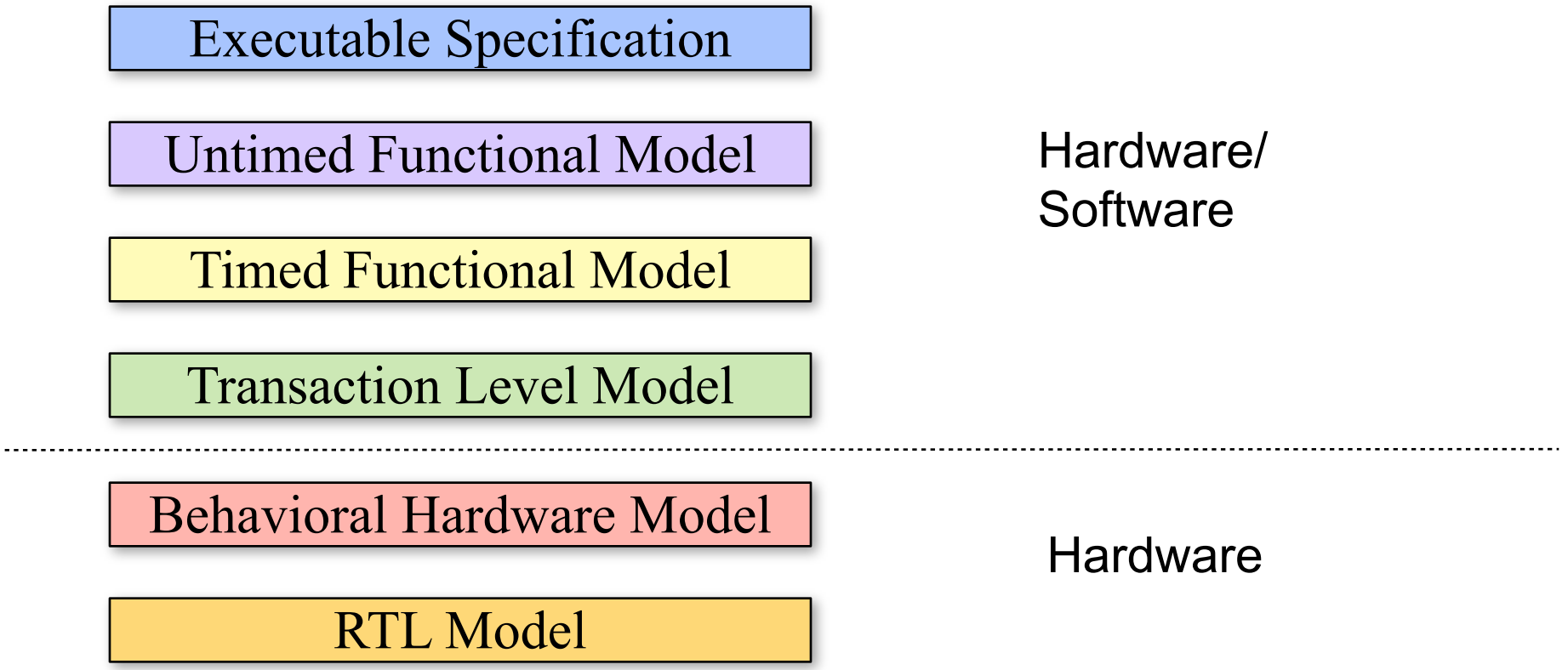
Wait for channel to be filled with data

Notify read() that data has been written

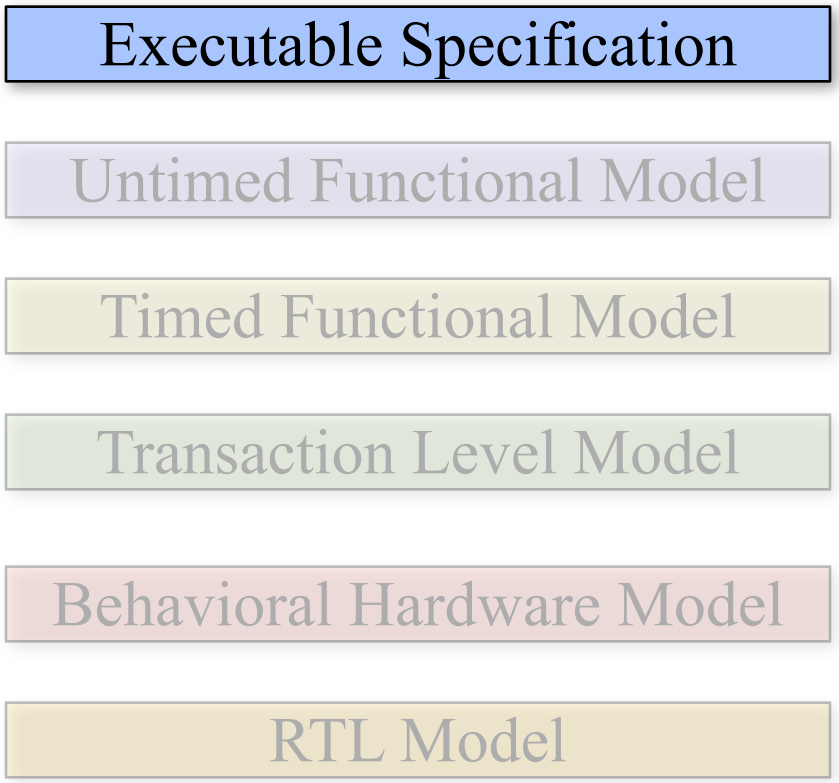
System Level Modelling

- Requirements for a System Level Description Language
  - *Specification and Design at various levels of abstraction*
  - *Creation of executable specification*
  - *Fast simulation, e.g. for Design Space Exploration*
  - *Separation of functionality from communication*

Modelling levels / Levels of Abstraction

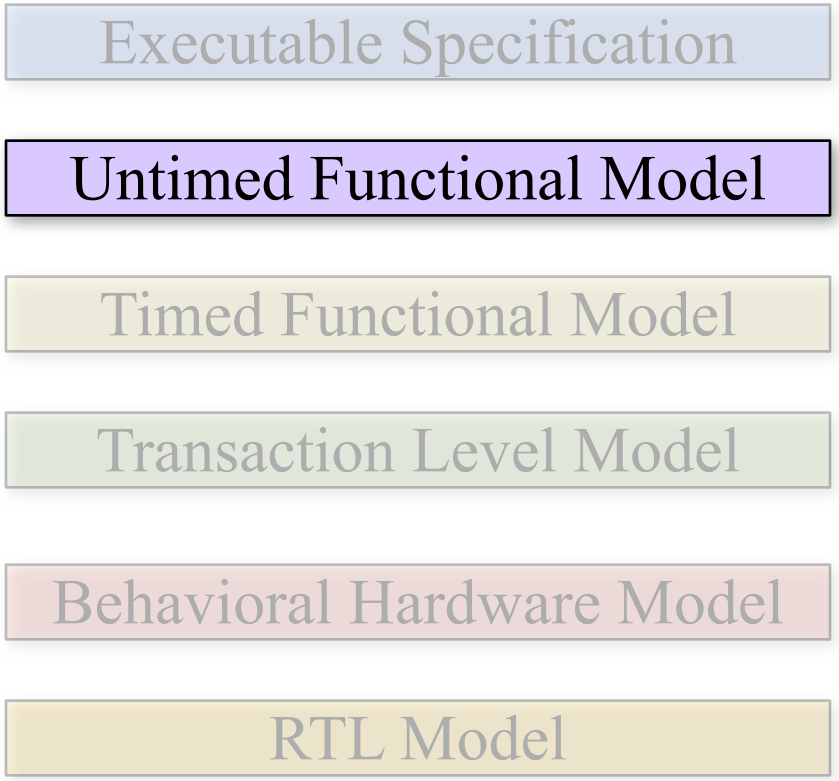


Modelling levels / Levels of Abstraction



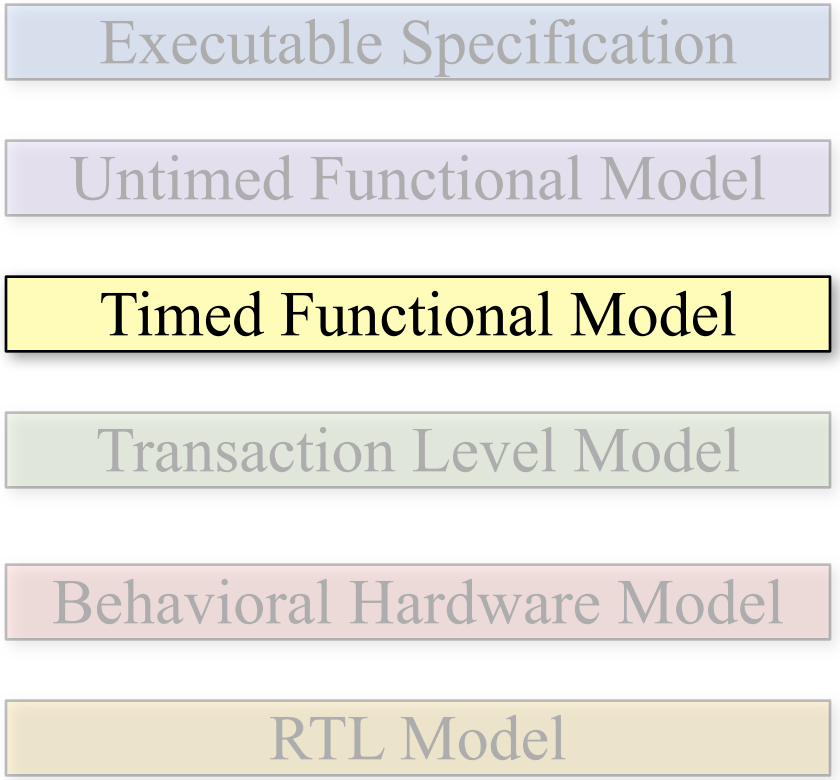
- System specification in SystemC
- Completely implementation independent

Modelling levels / Levels of Abstraction



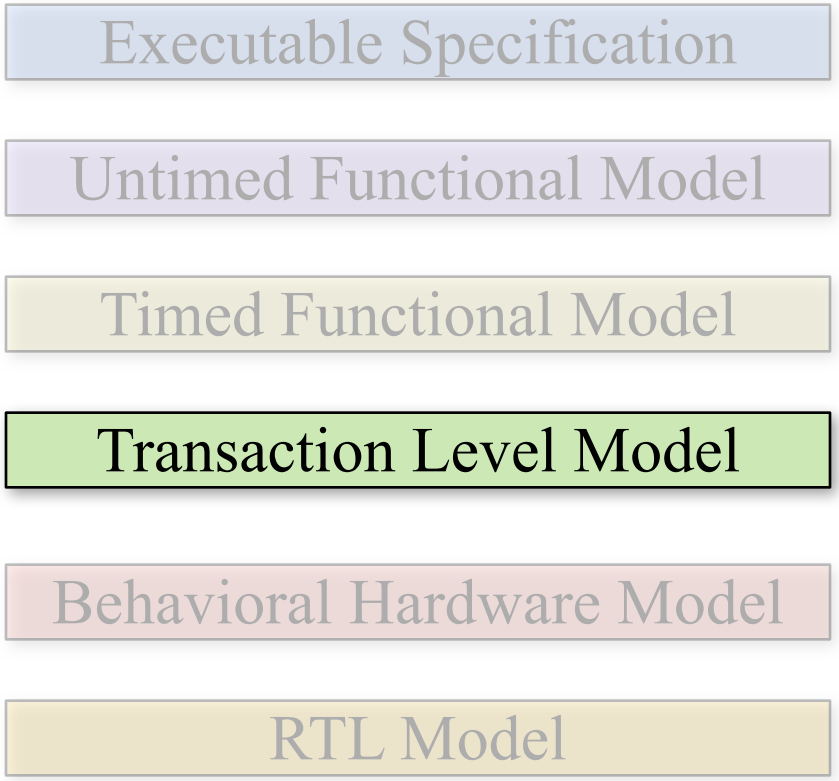
- Similar to Executable Specification
- No time delays
- Communication between modules is point-to-point
- No shared communication resources (e.g. busses)

Modelling levels / Levels of Abstraction



- Communication between modules still point-to-point
- Time delays are added to reflect timing constraints of specific target implementation
- Early Hardware/Software trade-off analysis

Modelling levels / Levels of Abstraction



- Communication between modules using function calls
- Data contained in abstract structures

Models of Computation

Model of Computation has impact on:

- model of time employed and event ordering constraints within the system
- supported methods of communication between concurrent processes
- rules of process activation

Models of Computation

SystemC Kernel based on a Discrete Event MoC
Base MoC that is extremly general
Other MoCs can be implemented on top of it
SystemC constructs for the mapping of other MoCs:

- Events (notify(), wait())
- Specially defined Channels, Interfaces, Ports and Modules

Models of Computation

MoCs, that can be modelled in SystemC:

- Static Multirate Dataflow
- Dynamic Multirate Dataflow
- Kahn Process Network
- Discrete Event („native“)
  - RTL Hardware Models
  - Network Models
  - Transaction based SoC Platform Models

Models of Computation

Kahn Process Networks (KPN):

- Effective MoC for description of algorithms in signal processing systems
- Blocks (or processes) are executed concurrently, are connected by Channels carrying data tokens
  - infinite FIFOs
- Blocking read and nonblocking write operations
- No concept of time
- Deterministic: order of process execution has no influence on data tokens in FIFOs

Models of Computation

Static Dataflow (SDF) Networks:

- Special case of KPN:
  - Functionality inside a process is clearly separated in 3 stages:
    - 1. Reading input tokens
    - 2. Execution of computation
    - 3. Writing output tokens
  - Number of tokens read and written by a process is fixed and known at compile time
- Tools can create static scheduling and compute limits for all FIFOs at compile time
- Thus faster than KPN (dynamic scheduling)

Models of Computation

Modelling KPN and SDF in SystemC:

- In practise, there are no infinite FIFOs, write operations might be blocking
- Designer has to spezify the size of FIFOs and initial states in a way that no deadlocks can occur
- Mapping to SystemC‘s Discrete Event Kernel diminishs benefits:
  - no static scheduling, SDF is scheduled dynamically by kernel
  - high number of context switches reduce simulation speed
- SystemC is Open Source, its Kernel‘s functionality can be enhanced such that other MoCs can be „directly“ implemented

Functional Level Modeling

Untimed Functional Modeling:

- Model based on KPN
  - FIFOs of limited size with blocking read and write operations
- Modules made up of SC_THREAD processes
- No time present, every computation step takes place in delta cycles
- Implicit Synchronisation using FIFOs
  - Read operation blocks until data is available
  - Write operation blocks until free memory is available
- Initial Values must be present in the system
  - FIFOs filled with data prior to simulation
  - Insert data producing elements into system
- Results have to be read out, otherwise deadlocks occur if FIFOs are full

Functional Level Modeling

- Example

```
// Simple Adder UTF
SC_MODULE (Adder) {
  sc_fifo_in <int> in1, in2;
  sc_fifo_out <int> out;

  void add () {
    while (1) { out.write(in1.read()+in2.read()); }
  }
  SC_CTOR (Adder) {
    SC_THREAD (add)
  }
};
```

Functional Level Modeling

Timed Functional Modeling:

- Notion of time required when functional models are used with models on a lower level of abstraction
- Timing behaviour is modelled using *wait ()* statements
- Timed and untimed models can exist in parallel and interact with each other
- FIFO based and signal based communication can be mixed

Functional Level Modeling

- Example

```
// Simple Adder TF
SC_MODULE (Adder) {
  ...
  void add () {
    while (1) {
      int sum = in1.read() + in2.read();
      wait (100, SC_NS);                    // models delay
      out.write (sum);
    }
  }
  ...
```

Transaction Level Modeling

High level approach for modelling digital systems

Details of communication are separated from details of implementation of functional units or architecture

- Main focus on functionality of communication - what data is transferred from where and to where - and less on the actual implementation, e.g. which protocol is used
- Synchronization abstracted in blocking and nonblocking communication

Communication via function calls

- Transactions, consisting of start time, end time, and payload data

Transaction Level Modeling

no *SC_SIGNAL* Channels, transfer of data between processes by reading and writing shared variables

- Synchronization very important

TLM much faster in simulation than corresponding RTL model

allows simulation of reasonable magnitude of software together with a hardware model

Cycle accurate models can provide a defined interface for hardware and software design

- TLM model is used as a reference for designed hardware and software

Transaction Level Modeling

Why do TLM models make sense?

- Relatively easy to develop, understand, use and expand
- High accuracy of model (both software and hardware)
- Created in early design stages, design space exploration, trade-off analysis
- Fast and accurate enough to validate software, even before hardware model or implementation is available

TLM is beneficial for:

- Functional Modelling (Untimed and Timed)
- Platform Modelling
- Test benches

Transaction Level Modeling

Increasing impact leads to standardization by Open SystemC Initiative (OSCI)

Reference Manual defines Interfaces and Protocols

TLM-2.0 classes as additional layer on top of SystemC elements

Example „Simple Bus“

- High-performance, cycle accurate platform Transaction Level Model
- Source code and documentation available as part of the SystemC package
- contains different block types: Bus, Master, Slaves, Arbiter, Clock Generator

Levels of abstraction

Untimed Functional Level

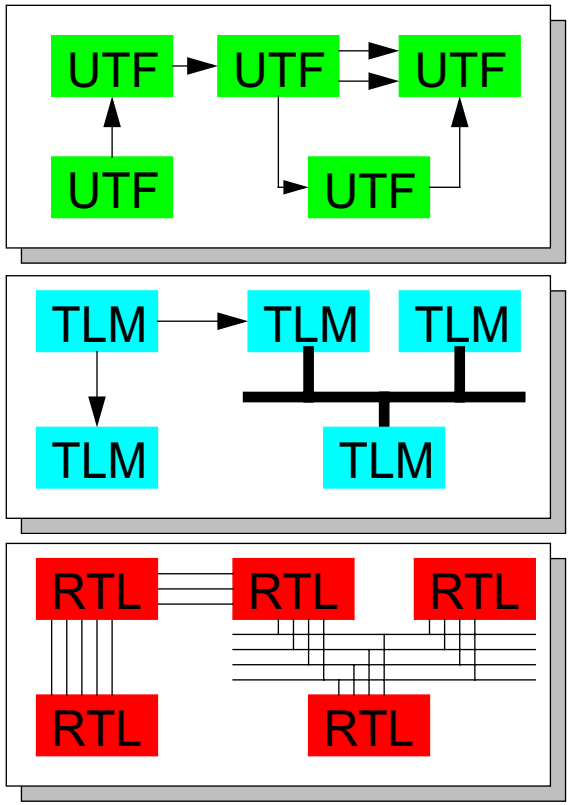
Executable specification

Transaction Level

Platform Design,
HW/SW Co-Verification

RTL

RTL/Behavioural HW Design
and Verification

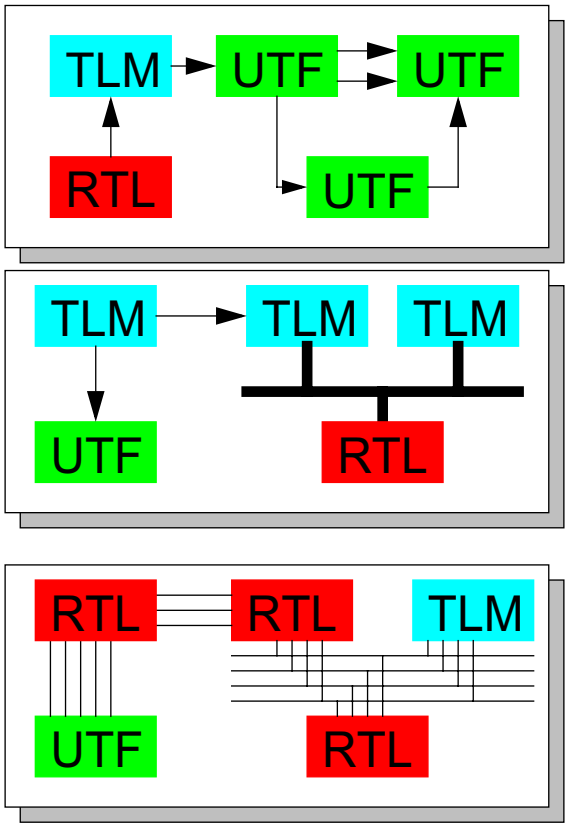


Levels of abstraction

No large step required
from executable
specification to RTL

Bus cycle accurate Transaction
Level Models for fast Platform
simulation (>>100k cycles/s)
in early design stages

No need to couple different
simulators for co-simulation



Use of SystemC in Design flow

- Functional Modeling
  - System and Algorithmic Level
- Transaction Level Modeling
  - Fast Hardware/Software Co-Simulation
  - Trade-off Analysis
  - Testbench
  - Reference Model for RTL Design
- RTL Design still done with VHDL or Verilog
  - no way to replace traditional method / HDLs
- Future: High Level Synthesis?