System Modelling

Transaction Level Modelling

Prof. Dr.- Ing. Sorin Alexander Huss

Integrierte Schaltungen und Systeme
Fachbereich Informatik
Technische Universität Darmstadt

SS 2006

Transaction-Level Models with SystemC

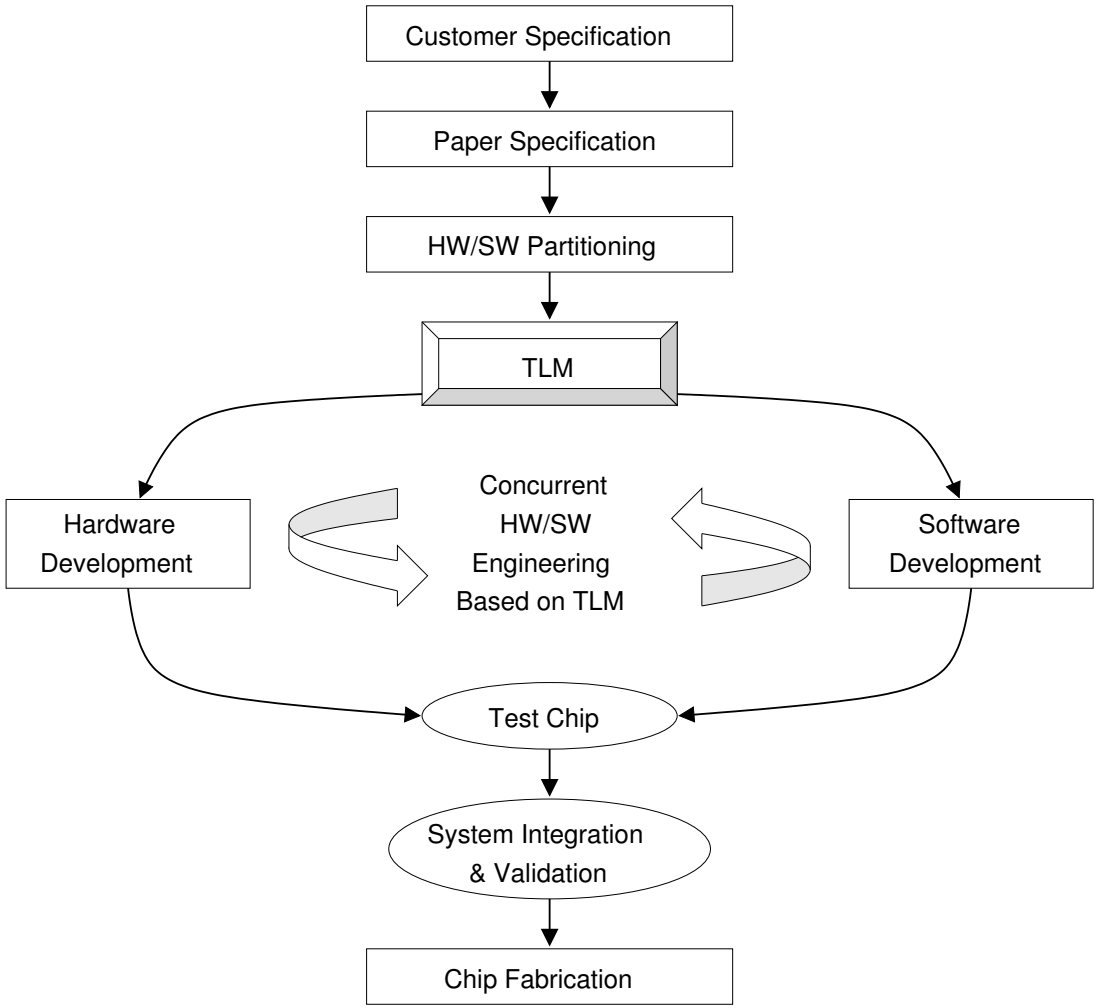
- TLM Methodology
- Introduction to SystemC
- Transaction Level Modelling in SystemC

(Source: Frank (Ed.) Ghenassia, editor. Transaction-Level Modeling with SystemC. Springer, June 2005)

TLM provides a methodology for

- Early software development
- Architecture analysis
- Functional verification

Novel design flow for embedded systems



Triple abstraction stucture

1. Functional view
2. Architecture view
3. Micro-architecture view

1. Functional view

- abstracts the expected behavior of a given system
- executable specification of ths system function composed of algorithmic SW
- no consideration of any implementation details - such as architecture or address mapping information - and of perfomance figures

2. Architecture view

- provides the platform for associated SW development
- provides the reference model for verification purposes, i.e., for the generation of functional verfication tests of subsequent implementation models

3. Micro-architecture view

- captures all required information for timed and cycle-accurate simulation of RT level models
- validates low-level embedded SW in the real HW simulation environment
- validation of micro-architecture for real-time requirements

Principles of TLM

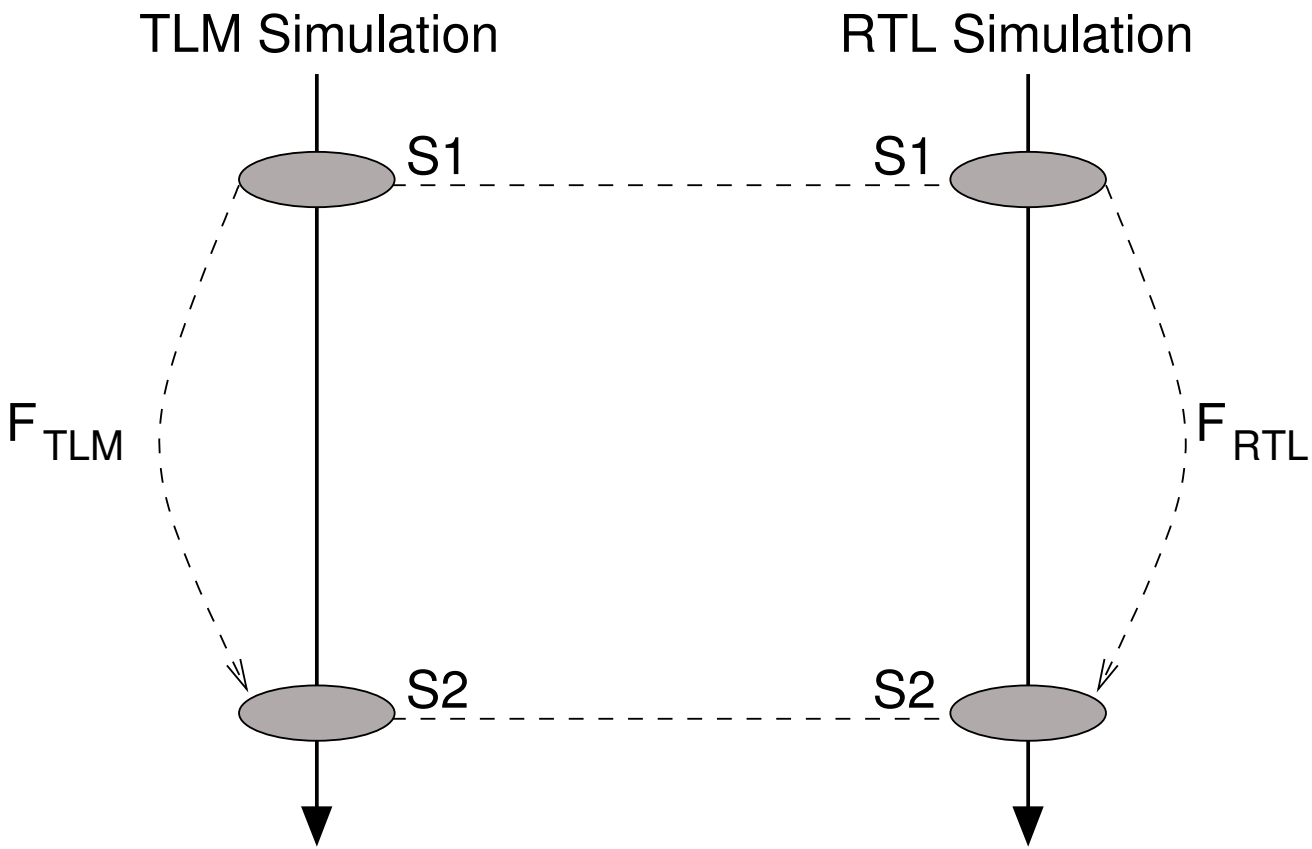
- Modelling of components as **modules**
- Communication structure by means of **channels**
- Modules and channels are bound to each other by communication **ports**
- A set of data is exchanged by a **transaction**
- **System synchronisation** is an explicit action between modules, e. g., interrupt by DMA to notify a transfer completion

Modelling approach

TLM modules must hold the following characteristics:

- Bit-true behavior of the component
- Register-accurate interface of the component
- System synchronisation managed by the component

TLM vs. RT level simulation



Modelling accuracy

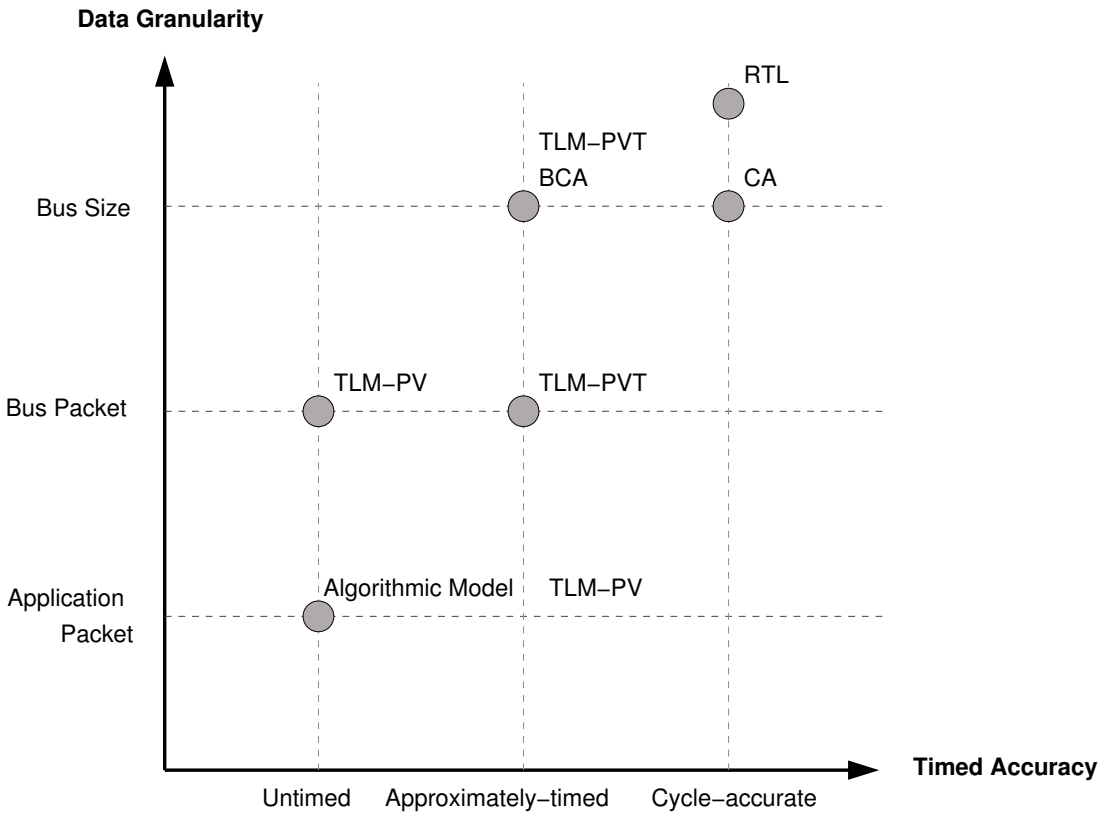
- Granularity of communication data

  Levels: *application packet, bus packet, bus size*
  (Example video transfer: frame, line, pixel)

- Timing accuracy

  Levels: *untimed, approximately-timed, timed TLM*

Modelling accuracy (2)



Legend:

RTL	register transfer level
BCA	bus cycle accurate
CA	cycle accurate
PV	programmer's view
PVT	programmer's view plus timing

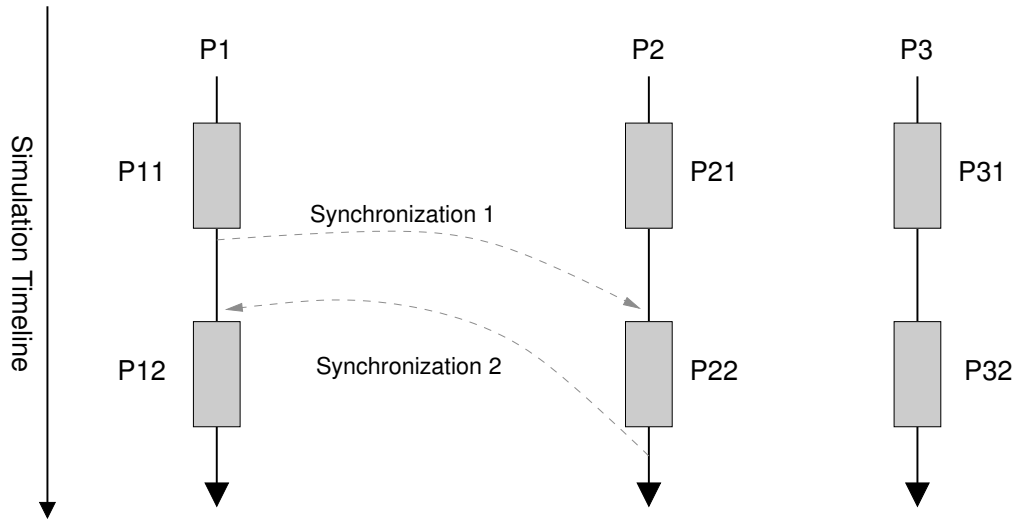
Untimed TLM - Model of Computation

Characteristics

- Concurrent execution of independent processes
- Respect for causal process dependencies by using system synchronisation
- Bit-true behavior
- Bit-true communication

Untimed TLM - System Synchronisation

Execution order within P1, P2, P3: System syncronisation between concurrent processes:



1. P11 → P12 for P1
2. P21 → P22 for P2
3. P31 → P32 for P3
4. P11 → P22
5. P22 → P12

Untimed TLM - System Synchronisation (2)

Different overall execution order (**process interleaves**) examples:

1. P21 $\rightarrow$ P11 $\rightarrow$ P22 $\rightarrow$ P12 $\rightarrow$ P31 $\rightarrow$ P32
2. P31 $\rightarrow$ P32 $\rightarrow$ P21 $\rightarrow$ P11 $\rightarrow$ P22 $\rightarrow$ P12

Syncronisation kinds:

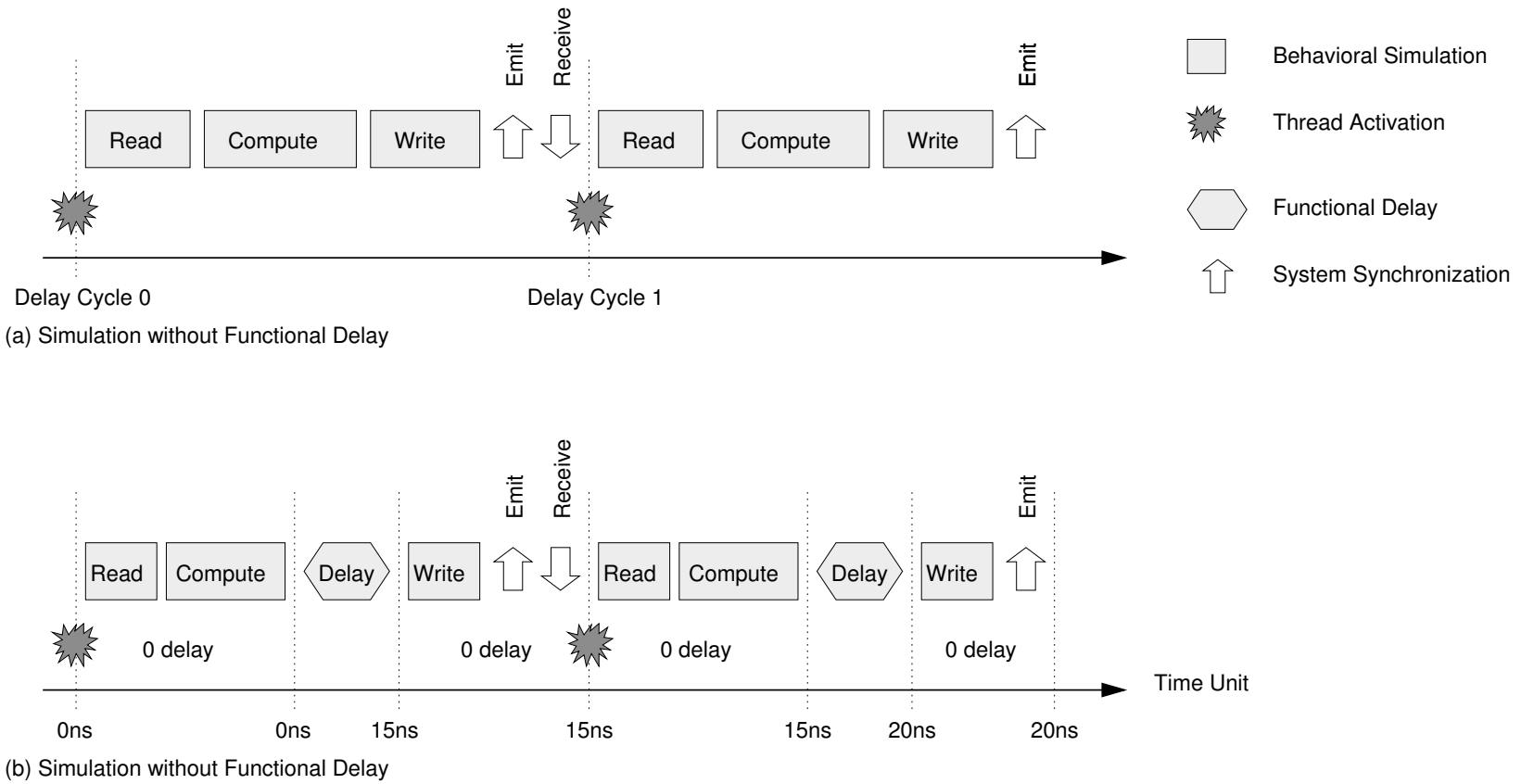
Emit-synchronisation A process sends out a synchr. that may influence the behavior or state of other processes.

Receive-synchronisation A process waits for an event from the system that may influence its behavior or state.

Untimed TLM - Deterministic Behavior

Step	Action
1	*Activate* or resume a process
2	Read input data for control flow and data processing
3	Computation
4	Write output data if there is any
5	Return to Step 2 if more computation is required
6	Synchronisation:
	a) if it is 'emit-synchronisation', then return to Step 2
	b) if it is 'receive-synchronisation', then the process will be *suspended*.

Approximately-Timed TLM - Insertion of Functional Delays



Note: Functional delays inserted at *architecture* level.
Insertion of functional delays must not change causal dependencies in a given system!

Timed TLM - Objectives and Modelling Approach

Objectives:

- Benchmarking of the performance of a given *micro-architecture*
- Fine tuning of the micro-architecture
- Optimizing the SW for a given micro-architecture to meet real-time constraints

Modelling approaches:

- Time annotated untimed model
- Standalone timed model

Time annotated TLM

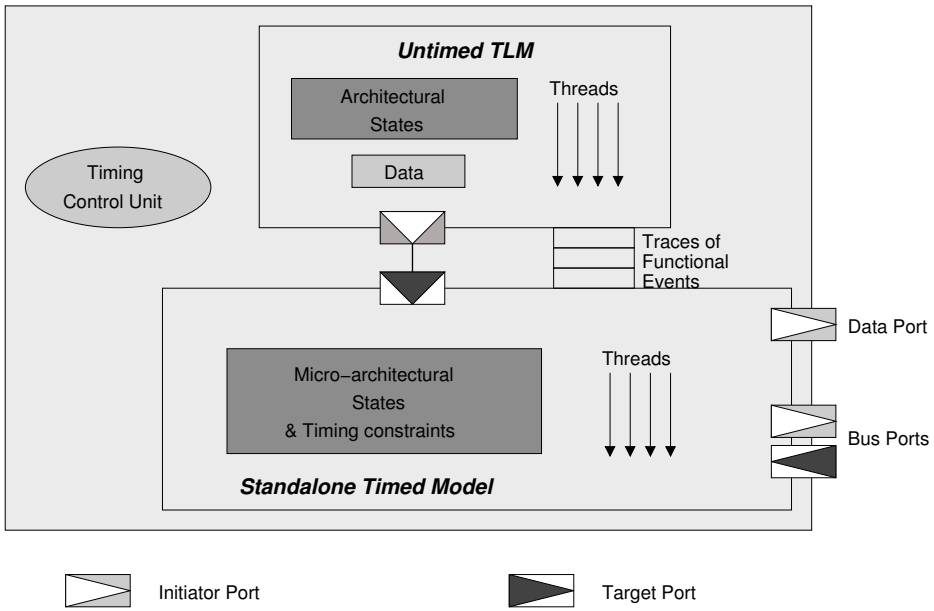
- Well-suited in case of a fairly good match of the untimed model structure to the micro-architectural model (re-use of untimed TLM): Simply insert parametrized *wait* statements related to the concerning computation times.
- In general: Define *delay* of each possible *activation-synchronisation* in a process based on the control flow of the component. This may result in rather complex graphs and in a large set of timing attributes.

Standalone Timed TLM

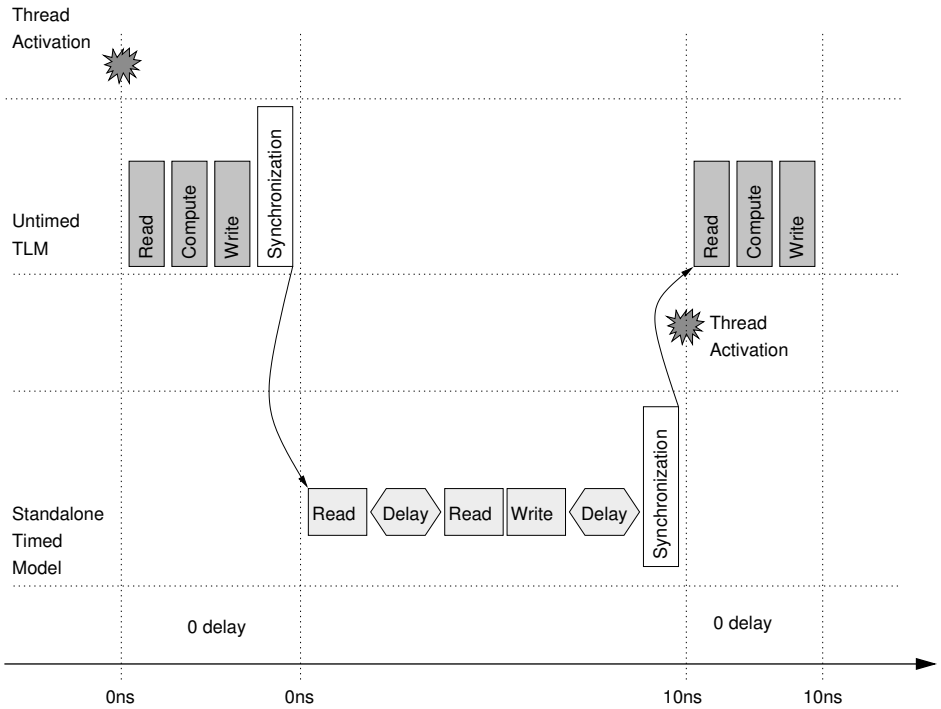
- Detached model incorporated with timing information: High-level analytical timing model *without* functional information. Suitable in case of large differences between the structures of the algorithm and of the related micro-architecture.
- Depedancy of timing behaviour on functional component behaviour: Control of the timed model by an untimed TLM by tracing and forwarding of all functional events produced by the untimed model to the standalone timed model.

Combined Untimed and Standalone Timed TLM

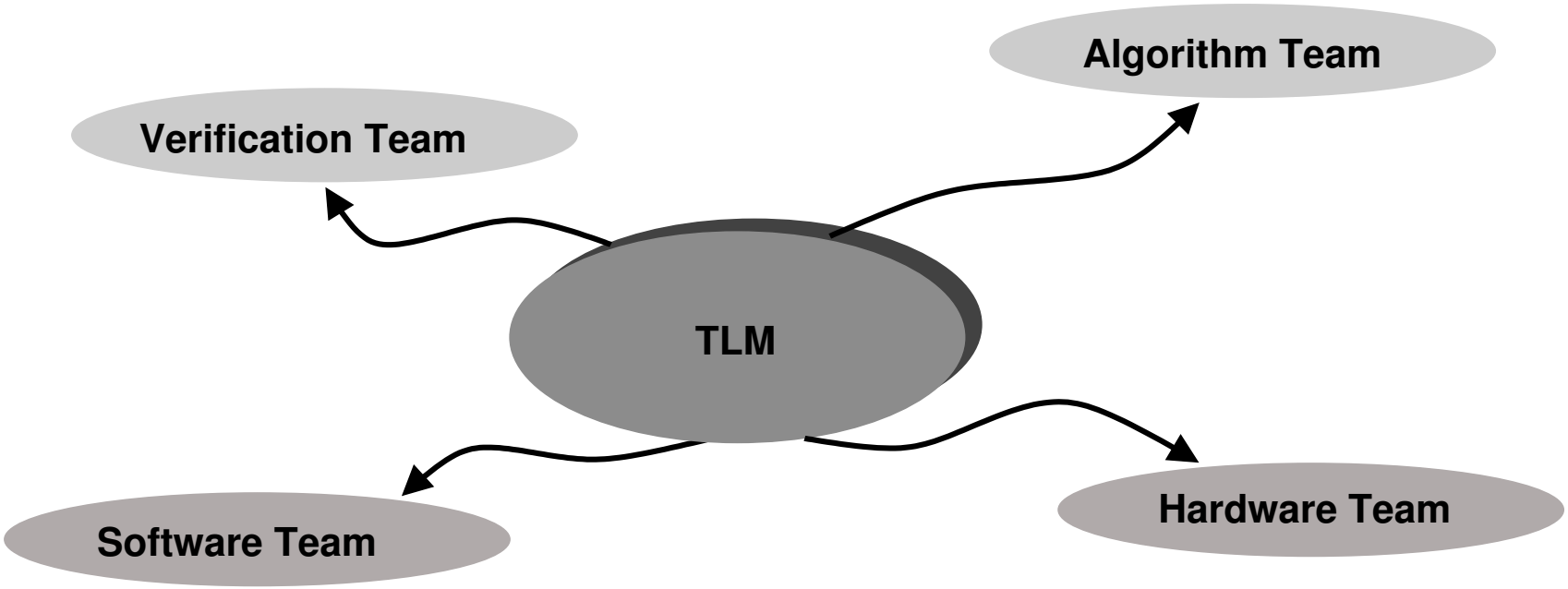
Model architecture



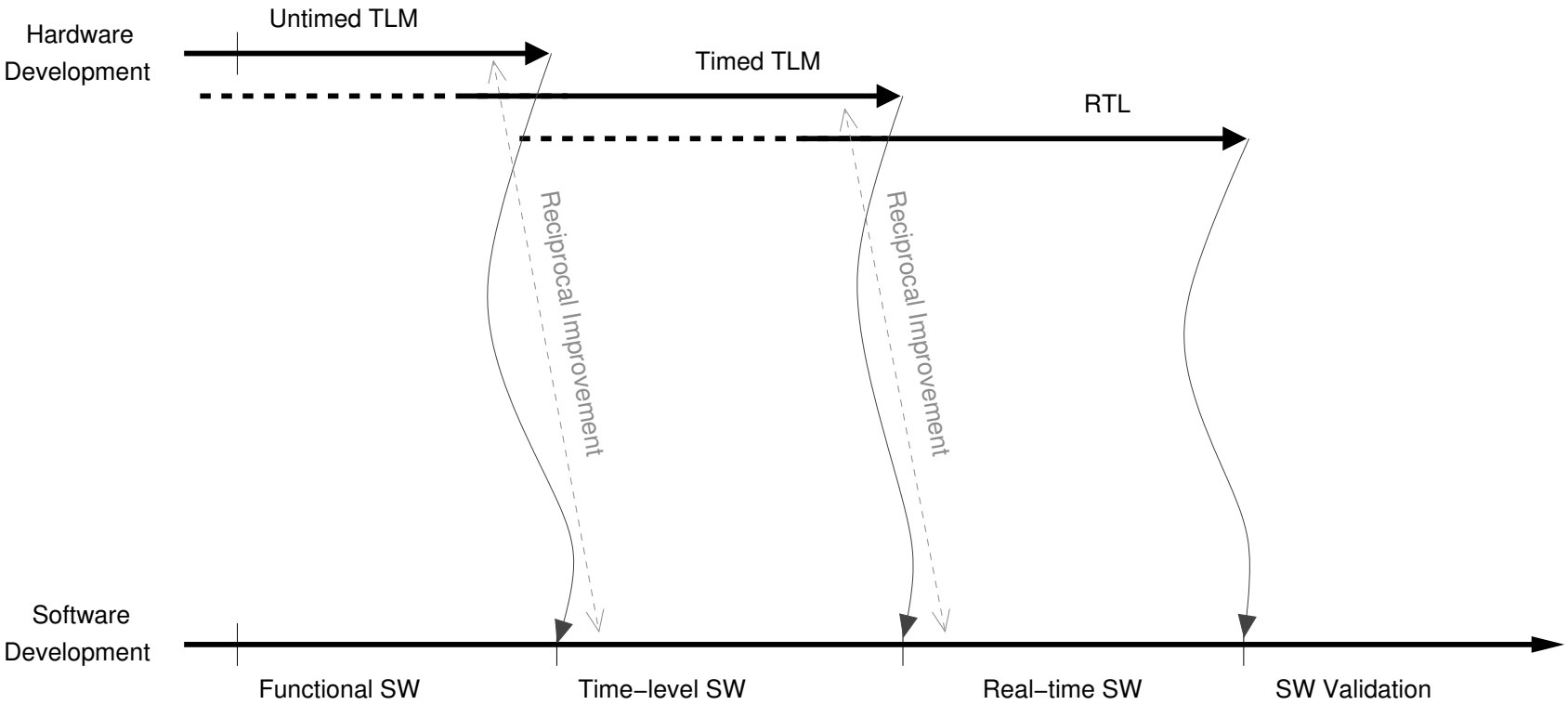
Model execution



TLM as Unique Reference Model



TLM-oriented HW/SW Development



Introduction to SystemC

SystemC

- Is a programming language implemented as a C++ class library such that C++ is a subset of SystemC. It is an IEEE Standard and it is freely downloadable from `www.systemc.org`.
- Enhances C++ such that it becomes more suitable for specific modelling purposes of embedded systems, i. e., tightly cooperating HW and SW functional modules.
- Supports concurrency, timed behaviour, HW specific data types, and structural descriptions of embedded systems models.
- Incorporates a strict separation of *computation* and *communication tasks* within a distributed information processing system.

SystemC Interfaces, Modules, Channels, Ports (1)

Interface A set of methods which are usable for particular communication channels. It is described using an abstract base class in C++ containing pure virtual methods only.

Module An entity for structuring architectural descriptions. It may contain several processes, which execute concurently.

SystemC Interfaces, Modules, Channels, Ports (2)

Channel A communication path between processes. It may be as simple as a signal or rather complex featuring an own internal structure and processes (hierarchical channel). It may be refined in an object-oriented manner.

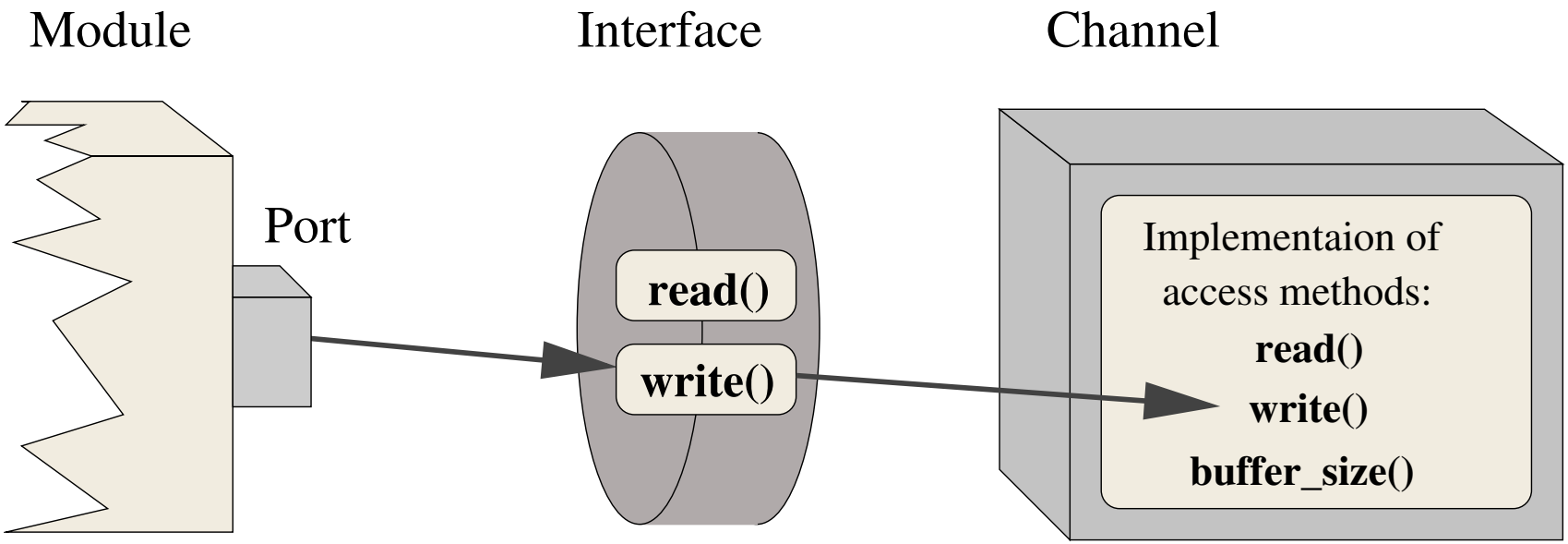
Port This is the communication access point of a module. A port is typed according to the type of the interface it is bound to.

Summary:

- Channels are accessed via interfaces.
- Modules access channels via ports.
- Ports are related to the type of interface they may connect to.

SystemC Communication Concept

Concept: Modules are linked to channels via interfaces.



Signals are predefined channels

Module Example

Example

```
#include "systemc.h"
struct MAC: sc_module {
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> acc;
    sc_signal<int> prod;
    void action1() { prod = a * b; }
    void action 2() { acc = acc + prod; }

    SC_CTOR (MAC) {
        SC_METHOD(action1);
            sensitive << a << b;
        SC_METHOD(action2);
            sensitive << prod;
    }
};
```

SystemC Processes

Types of processes:

- SC_METHOD
- SC_THREAD
- SC_CTHREAD

Declaration and activation:

```
SC_CTOR(Module) {
     SC_THREAD(thread);
          sensitive << a << b;   // static sensitivity list
}
void thread() {
    for (;;) {
        wait(10, SC_NS); // dynamic sensitivity - time value in nanoseconds

        wait (e);        // dynamic sensitivity - event
    }
}
```

SystemC Data Types

`sc_int<>`	64-bit signed integer
`sc_uint<>`	64-bit unsigned integer
`sc_bigint<>`	arbitrary precision signed integer
`sc_biguint<>`	arbitrary precision unsigned integer
`sc_logic`	4-valued single bit
`sc_bv<>`	vector of 2-valued single bits
`sc_lv<>`	vector of 4-valued single bits
`sc_fixed<>`	templated signed fixed point
`sc_ufixed<>`	templated unsigned fixed point
...	

TLM in SystemC: Example (1)

1. Interface for the master side of a bus:

```
#include "systemc.h"
class master_if : virtual public sc_interface {
    public:
        virtual void write(int priority, int address, int data) = 0;
        virtual void read(int priority, int address, int* data) = 0;
};
```

Note: The interface `master_if` is derived from the class `sc_interface`. It just creates two methods: `read, write`. The bus is then created as a module, derived from the interface class.

(Source: J. Aynsley, A. Fitch (Doulos Ltd.) - Euro DesignCon 2004 tutorial)

TLM in SystemC: Example (2)

2. Bus model

```
class Bus : public sc_module, public master_if {
    public:
        sc_in<bool> clock;
        sc_port<slave_if,2> slave_port;

        Bus() :
        {
                SC_THREAD(busarbiter);
                  sensitive << clock.pos();        //static sensitivity
         }
         void write(int priority, int address, int data);
         void read  (int priority, int address, int* data);
         void busarbiter (...);

     private:
         bool* request;
         sc_event* proceed;
};
```

TLM in SystemC: Example (3)

3. Bus write function (part of)

```
void Bus::write(int priority, int address, int data} {
     request[priority] = true;
     wait (proceed[priority]);
     ... }
```

Note: Set flag when master attepts to write to the bus and wait until permission is given. The call to `wait` overrides the sensitivity list of the master - it is an Interface Method Call, IMC.

TLM in SystemC: Example (4)

4. Use of the bus

- Port declaration in the master module: `sc_port<master_if>` busport;
- Create and bind instances:

```
Bus MyBus("MyBus");
Master MyMaster("MyMaster");   // bind port to interface of the bus
MyMaster.busport(MyBus);
```

Note: Now the code in `MyMaster` may invoke the `read` and `write` methods implemented in the bus.

TLM Summary

TLM methodology is:

- HDL independent.
- An IEEE Standard.
- A key stone in modern embedded systems design flows.

End

Questions ?