Modeling

Hardware Software CoDesign

September 2011

Agenda

Modeling

1. Basic Themes that came out through the Historical Background (repeat)
2. Basic Modeling Requirements
3. SystemC Language based Modeling

Modeling

Basic Themes

1. **Modeling** :: Because parallelism is important, the choice of an appropriate modeling paradigm is also important. Different modeling formalisms need to be developed that capture the various aspects of system behavior.

2. **Analysis and Estimation** :: Different models of the same system behavior need to be analyzed and estimated for performance. Performance would include tradeoffs for Area, Speed, Power. Cost to build, Time to Market are other factors.

3. **System-level partitioning, synthesis and interfacing** :: The basic steps of co-synthesis. A range of methodologies are applied for this.

4. **Implementation generation** :: Once the architecture has been generated and trade-offs known, the designs for the hardware and software components must be created.

5. **Co-simulation and Emulation** :: This helps designers to evaluate architectures and validate assumptions on implementations. Emulation uses FPGA / other emulation techniques to further speed up execution of system models.

Modeling

Basic Modeling Requirements

1. **Concurrency** :: H/W :: The ability to have multiple processes run simultaneously.

2. **State Transitions** :: H/W & S/W :: Conceptualize the system into different modes or states of operation (behavior). The states change based on input and the state transitions imply certain output characterestics.

3. **Hierarchy** :: H/W :: Conceptualize a system into building blocks or subsystems.

4. **Programing Constructs** :: S/W :: The ability to describe sequential algorithms.

5. **Exception Handling** :: H/W & S/W :: The ability to respond to events external and internal.

6. **Timing** :: H/W & S/W :: The ability to create a protocol and model it not only behavioraly but also by specifying a timing constraint or delay in signals.

7. **Communication** :: H/W & S/W :: The ability to describe a channel of communication. It could be as complete as a protocol, or as behaviroral as just transfer of data through a global variable.

8. **Process Synchronization** : H/W & S/W :: The ability to generate data and events to be used by other processes.

Modeling

Basic Modeling Requirements – Concurrency

1. In computer science, concurrency is a property of systems in which several computations are executing **simultaneously**, and potentially interacting with each other.

2. From the point of System level modeling, one can further divide concurrency into different types based on concepts of hardware modeling.

   (a) **Data-driven Concurrency** :: As in a data flow graph (DFG) model, execution depends on availability of data.

   (b) **Control-driven Concurrency** :: There are constructs that specify multiple threads of control, each of which can execute in parallel. eg. fork-join.

   (c) **Pipeline-driven Concurrency** :: Specific to signal processing and driven by throughput / latency requirements of systems.

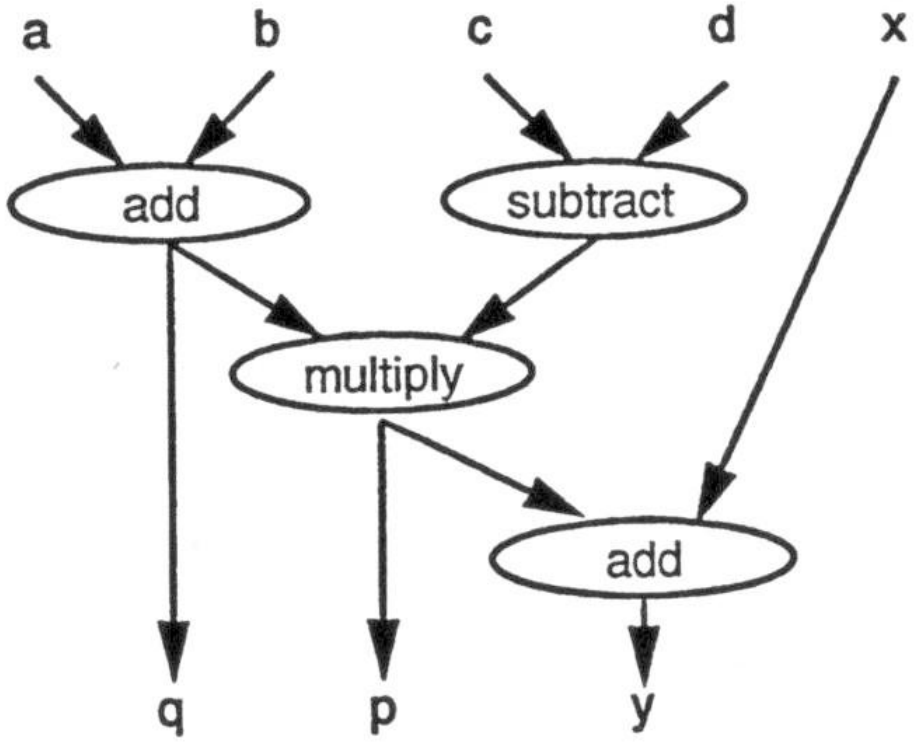
3. Concurrency can occur at Task level, Statement level or Operation level.

Modeling

Basic Modeling Requirements – Concurrency - Data-Driven

1: q = a + b
2: y = x + p
3: p = (c – d) * q

(a)



(b)

Modeling

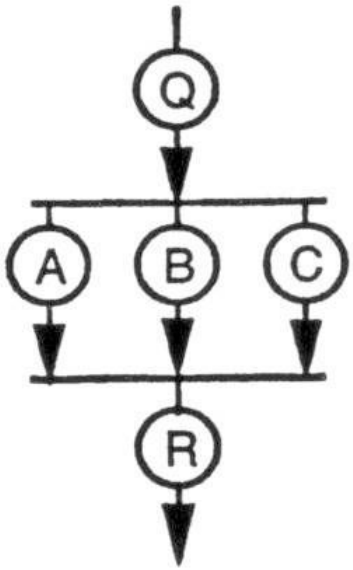
Basic Modeling Requirements – Concurrency - Control-Driven

```
sequential behavior X
begin
    Q();
    fork  A(); B(); C(); join;
    R();
end behavior X;
```

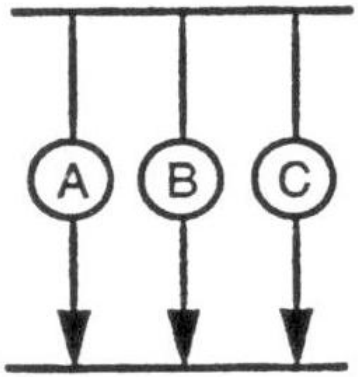
(a)

```
concurrent behavior X
begin
    process A();
    process B();
    process C();
end behavior X;
```

(b)



(c)

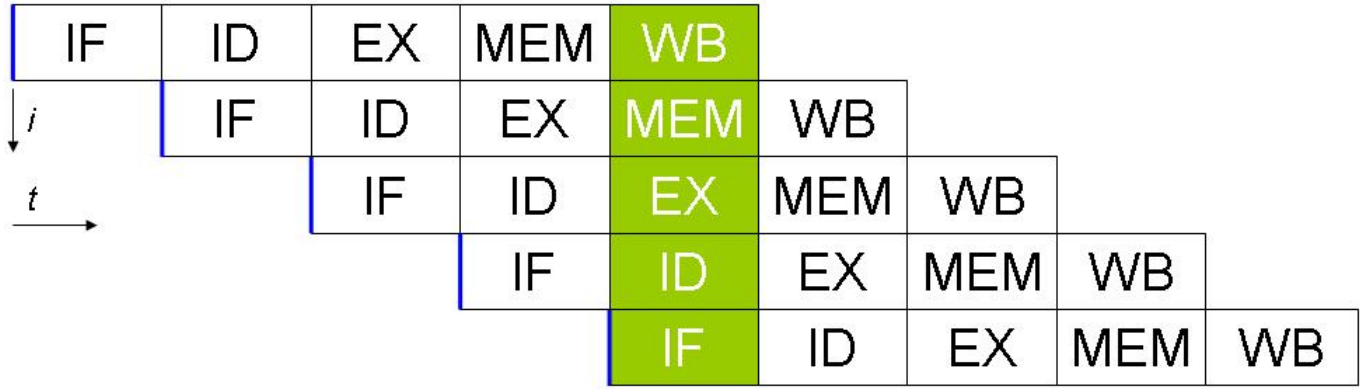


(d)

Modeling

Basic Modeling Requirements – Concurrency - Pipeline-Driven

Below is example of a typical five-stage pipeline in a RISC machine.



1. IF = InstructionFetch
2. ID = InstructionDecode
3. EX = EXecute
4. MEM = MEMory access
5. WB = register WriteBack

Modeling

Basic Modeling Requirements – State Transitions

1. Systems are best conceptualized as having various modes of operation or states, of behavior.

2. State transitions are triggered by the detection of certain events or certain conditions.

3. In software :: Would an `if-then-else` indicate a state transition?

4. In software :: Would a `goto` statement indicate a state transition?

5. Actions can be associated with each transition, and a particular mode or state can have an arbitrarily complex behavior or computation associated with it.

6. In case of the touchScreen example, the detection of a touch, followed by the invokation of the said task is an event driven State Transition.

Modeling

Basic Modeling Requirements – Hierarchy

1. Hierarchical models enable system to become modular and one can look at it as a set of smaller subsystems.

2. These subsystems further down can be looked upon as another system with its own characterestics.

3. This conceptual view helps development, due to inherent modularity and making a complex system look as blocks of easily understood subsystems.

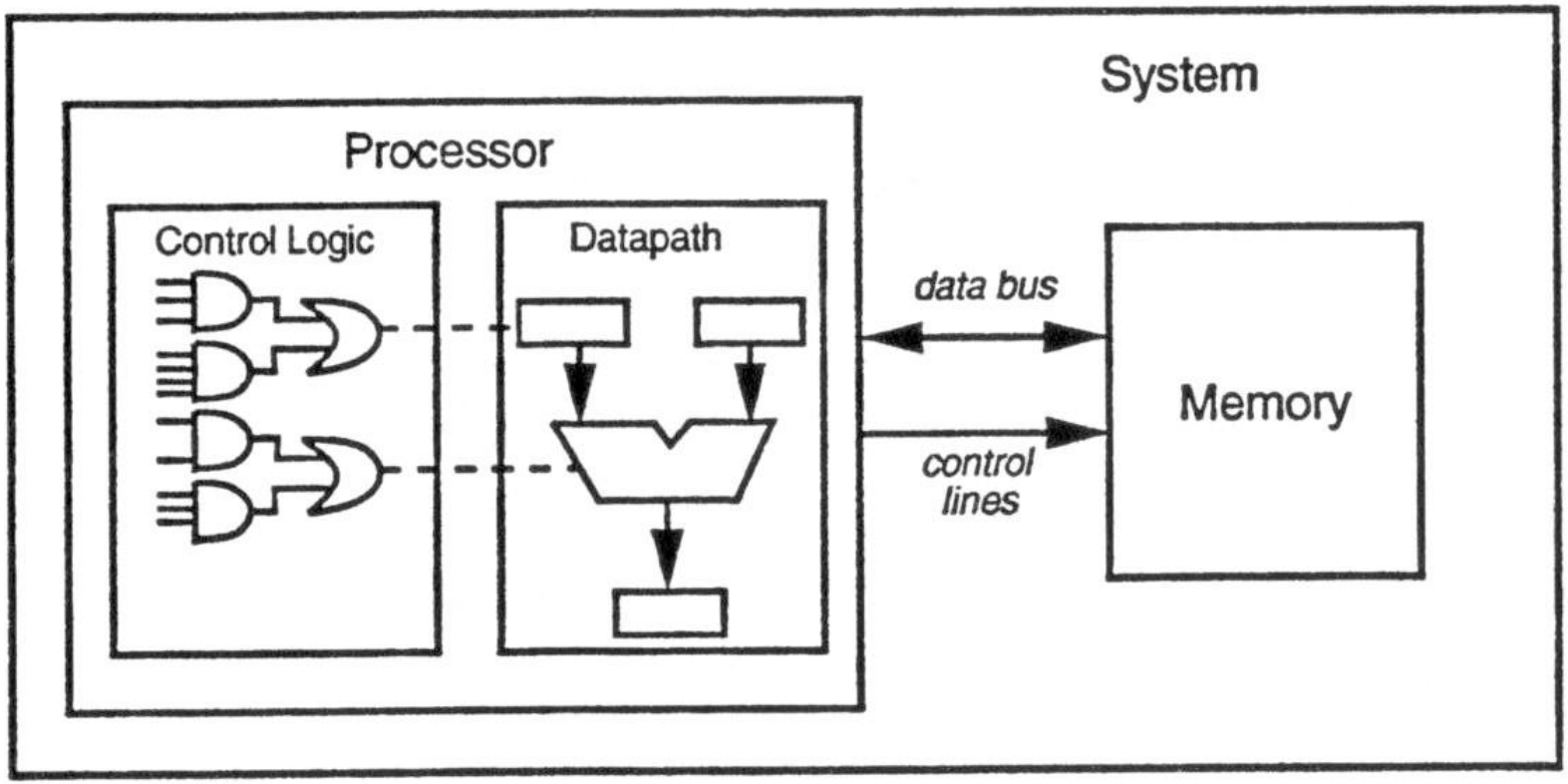
4. Hierarchy can be used to scope objects.

5. What would happen if there is no hierarchy in hardware or software?

6. Hierarchy can be of two types basically coming from the h/w & s/w backgrounds :: Structural hierarchy & Behavioral hierarchy

Modeling

Basic Modeling Requirements – Hierarchy - Structural

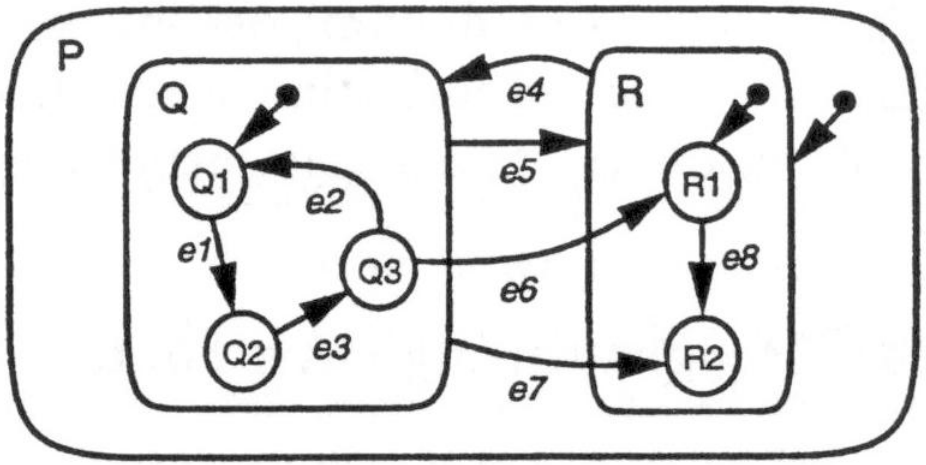


Modeling

Basic Modeling Requirements – Hierarchy - Behavioral

```
behavior P
    variable x, y;
begin
    Q(x) ;
    R(y) ;
end behavior P;
```

(a)



(b)

1. Is defined as the process of decomposing a behavior into distinct sub-behaviors, which can be either sequential or concurrent.

2. Behavioral hierarchy can further be of different types like sequential, parallel, or pipelined.

3. The Transitions can be simple (as within the blocks Q, R)

4. The Transitions can be grouped (as e4, e5) which looks like a request, grant pair.

5. The Transitions can be hierarchical (as e6, e7)

Modeling

Basic Modeling Requirements – Programing Constructs

1. Many behaviors may be best described using programming constructs : it allows the computations to be expressed explicitly.

2. Behaviors can be described as Sequential Algorithms

3. Typically, the construct includes assignment statements, branching statements, iteration statements and procedures.

4. In addition, data types are used like records (pascal?), arrays, linked lists and other complex data structures (or structure of structures).

```
type      buffer_type  is array (1 to 10) of integer;
variable  buf : buffer_type;
variable  i, j : integer;

for i = 1 to 10
    for j = i to 10

        if (buf(i) > buf(j)) then
            Swap(buf(i), buf(j));
        end if;

    end for;
end for;
```

Modeling

Basic Modeling Requirements – Behavioral Completion

1. Ability of the behavior to indicate that it has completed.

2. Other behaviors should be able to detect it.

3. Behavioral completion is required because :-

   (a) In hierarchical specification, it allows to view a subsystem as an independent module and allows for its independent developement, once the boundaries (input, output) are specified

   (b) Allows natural breakup of system behavior into sub-behaviors which can then be sequenced by the completion of transition arcs.

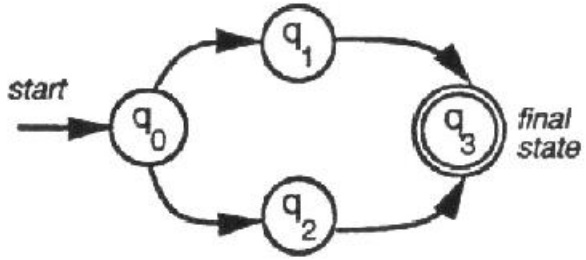
4. Behavioral Completion are represented by

   (a) Final state(s) in a FSM.

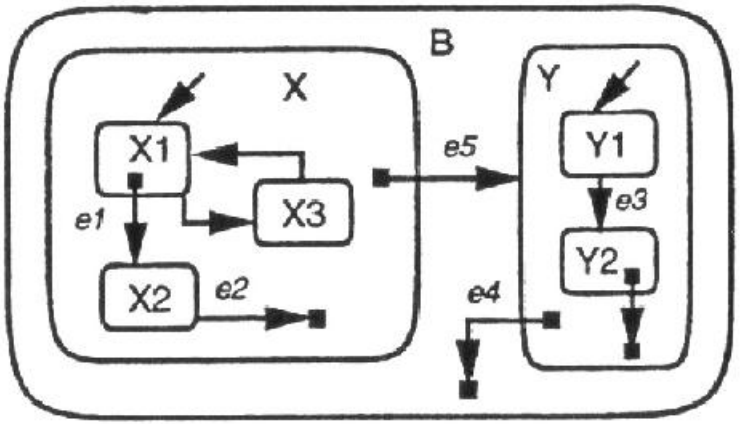
   (b) return statement in procedures.

Modeling

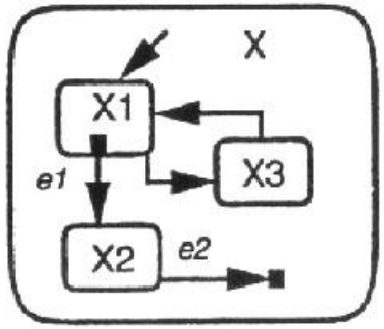
Basic Modeling Requirements – Behavioral Completion



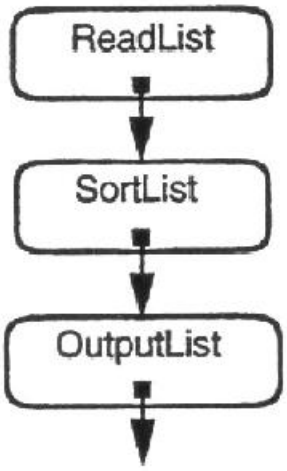
(a)



(b)



(c)



(d)

Modeling

Basic Modeling Requirements – Exception Handling

1. Occurance of a certain event can require that a behavior or mode be interrupted immediately, thus prohibiting the behavior from updating the values further. The NEXT behavior is specified explicitly.

2. It sometimes becomes crucial that occurance of an event should terminate current behavior immediately.

3. FSMs assume zero time computation in a state – hence an exception activates an appropriate transition.

4. Depending on the direction of the transferred control Exceptions can be further divided into :-

   (a) **Abort** :: the current behavior is terminated completly, and control transferred to event handling.

   (b) **Interrupt** :: the current behavior is context-switched, control transferred to event handling, and once the event handling is done, behavior is again context-switched to the completion of the previous tasks (from where it was interrupted).

Modeling

Basic Modeling Requirements – Timing

1. Timing constraints denote performance constraints, to be used by synthesis tools.

2. It can be checked through verification tools.

3. In general, a timing relation can be described as $(e1, e2, min, max)$ where $e1$ preceeds $e2$ by at least min time units and at most max time units.

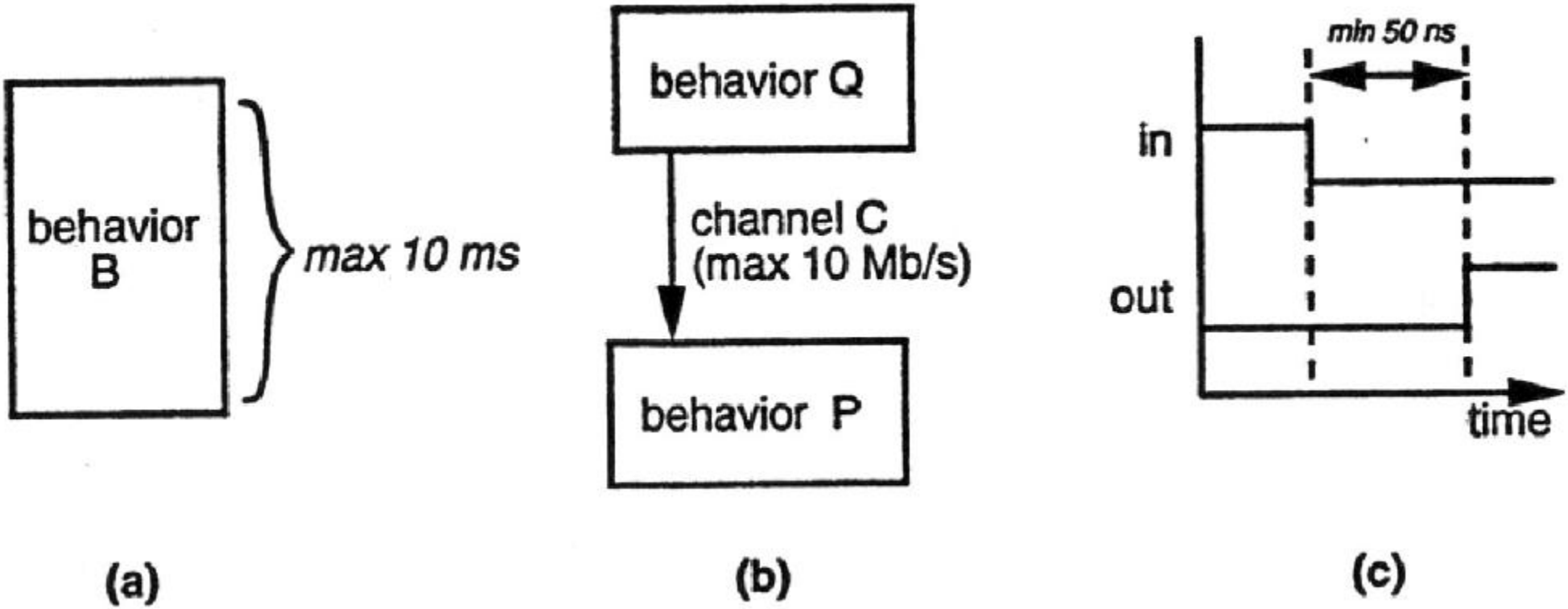
4. The different types of timing constraint are :-
   (a) Execution time constraints
   (b) Data transfer **rate** constraints
   (c) Inter event timing constraints

5. The timing information is important in **real time** embedded systems, whose performance is measured in terms of how well the implementation respects the timing constraints

6. From a specification point of view, constraints need to be specified as :-
   (a) Timing diagrams for IO event constraints
   (b) Statement labels for statement level constraints

Modeling

Basic Modeling Requirements – Timing



Modeling

Basic Modeling Requirements – Communication

1. Systems consist of several interacting behaviors which need to communicate with each other.

2. Traditionally (say the C programming language) Communication was through
   (a) For Functions :: communicate through global variables (or ports) which share the same memory space or via parameter passing.
   (b) For Local procedure calls :: information exchange is through a stack or through processor registers.
   (c) For Remote Procedure calls (RPC) :: parameters are passed through a complex protocol of sending/receiving data through network.

3. For embedded systems more is required in the form of :-
   (a) Separation of description of computation and communication
   (b) Declaration of abstract communication functions
   (c) Definition of a custom communication implementation

4. Crystalize these requirements into two basic types of communication
   (a) Shared Memory Model
   (b) Message Passing (Channel) Model

Modeling

Basic Modeling Requirements – Communication – Shared Memory Model

1. Uses shared medium such as global variable or port
2. Synchronization must be explicit like flag setting
3. May be broadcast mode – change sensed by all
4. Persistent mode – registered
5. Non Persistent – wired only (data is available only at the instant when it is written)

Modeling

Basic Modeling Requirements – Communication – Message Passing (Channel) Model

1. Messages sent through channels (an abstract medium, free from implementation details)

2. Requires send and receive primitives

3. "Send" destination may be explicit or implicit (specified by the interconnection)

4. Uni/Bi directional

5. Point to Point or multiway (more than two processes may communicate)

6. Communication can be blocking or non-blocking :-

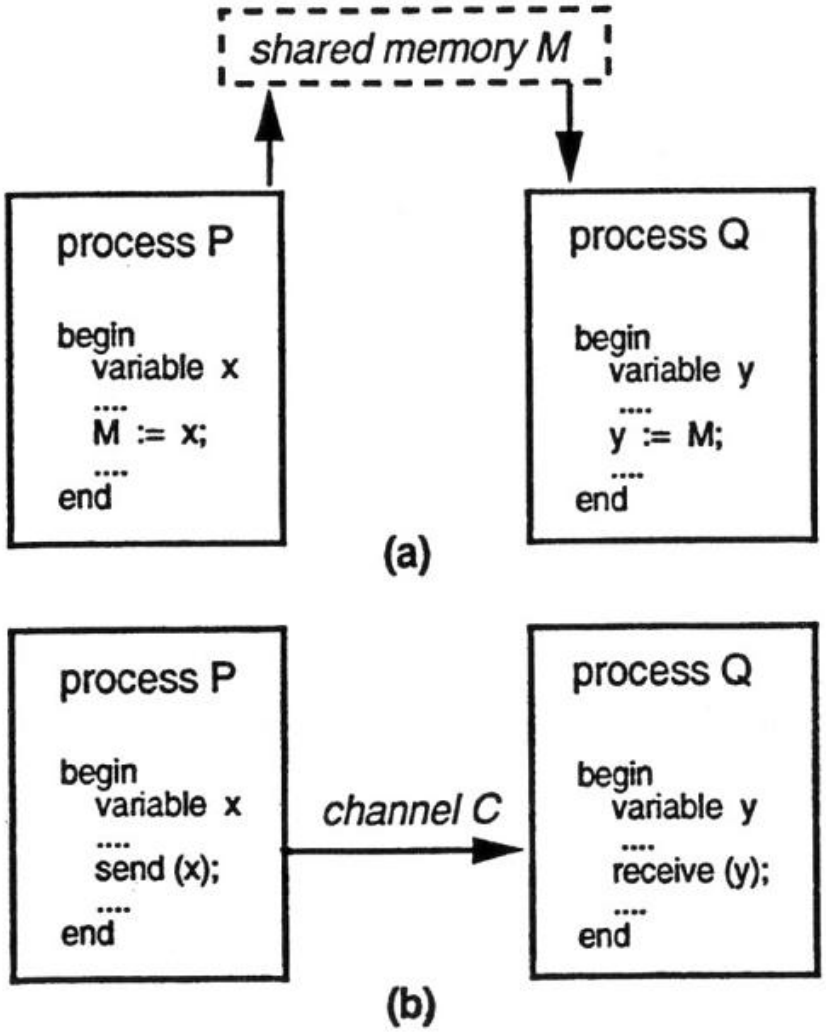
   (a) In Blocking Mode :: The sending process blocks itself until the receiving process is ready to receive data.
   - No extra storage required for storing data
   - Forced synchronization
   - Degrades performance

   (b) In Non-Blocking Mode :: storage is associated with the channel
   - For insufficient queue length, process may still block

7. A channel itself can be hierarchical. A channel may implement a high level protocol which breaks a stream of data packets into a byte stream, and in turn uses a lower level channel (eg. a synchronous bus, which transfers the byte stream one bit at a time)

Modeling

Basic Modeling Requirements – Communication



Modeling

Basic Modeling Requirements – Process Synchronization

1. Concurrent processes may generate data and events that need to be recognised by ohter processes. Hence Synchronization is a necessary requirement.

2. There are two basic types of Synchronization :-

   (a) Control Dependent Synchronization

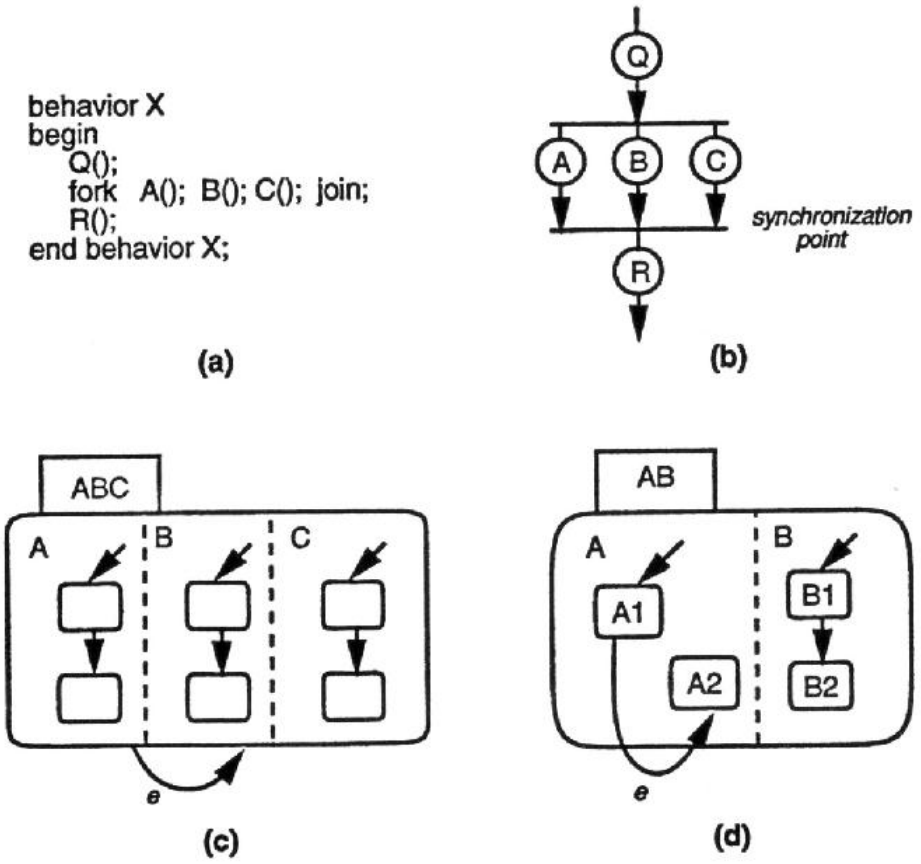
   (b) Data Dependent Synchronization

Modeling

Basic Modeling Requirements – Process Synchronization – Control Dependent

1. Control structure is responsible for synchronization eg. fork-join

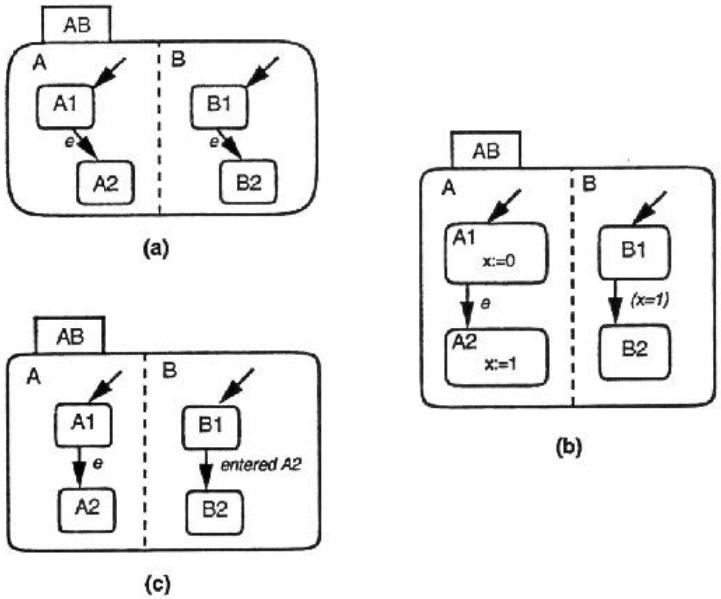
2. Initialization



Modeling

Basic Modeling Requirements – Process Synchronization – Data Dependent

1. Synchronized through Shared Memory
   - One process is suspended till the other updates the shared memory to a particular value.
   - Synchronization through a common event (a data value or event)
   - Synchronization through common data (variable)
   - Synchronization through status detection
2. Synchronized through Message Passing (blocking)



Modeling

Basic Modeling Requirements – Summary

1. State Transitions
2. Behavior Hierarchy
3. Concurrency
4. Exception Handling
5. Programming constructs
6. Behavior completion
7. Timing representation

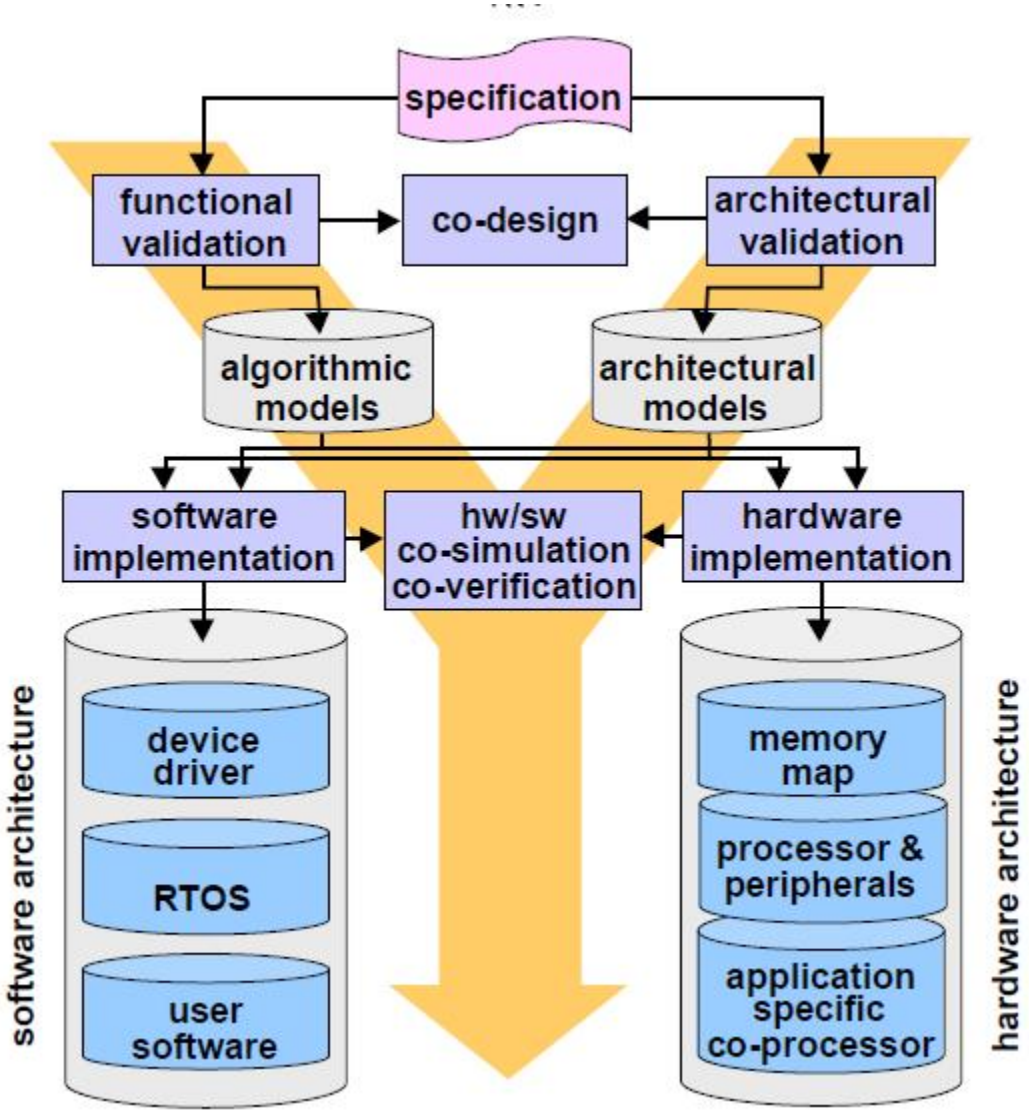
Modeling

Modeling Using SystemC

LOGICAL BREAK

Modeling

Modeling Using SystemC – (Restate) System Level Design Flow



Modeling

Modeling Using SystemC – Benefits of a C/C++ Based Design Flow

1. Productivity aspect
   - Specification between architect and implementer is executable
   - High speed and High Level simulation and prototyping capibility
   - Refinement, no translation into hardware (i.e, no "semantic gap")
2. System level aspect
   - Tomorrowś systems designers will be designing mostly software and less hardware (these are Joachim Gerlachś view and motivation factor)
   - Co-design, co-simulation, co-verification, co-debugging, co-....
3. Re-use aspect
   - Optimum re-use support by "Object Oriented" techniques
   - Efficient testbench re-use (for both software, hardware and system)
4. Especially C/C++ is widespread and commonly used

Modeling

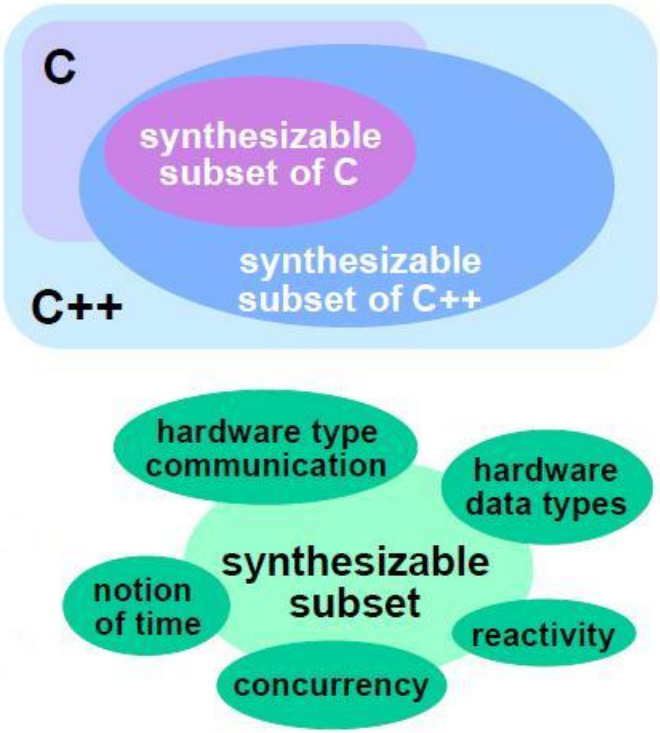
Modeling Using SystemC – **Drawbacks** of a C/C++ Based Design Flow

1. C/C++ is `not` created to design hardware !!

2. C/C++ does `not` support
   - Hardware style communication :: Signals, protocols
   - Notion of time :: Clocks, time sequenced operations
   - Concurrency :: Hardware is inherently concurrent, operates in parallel
   - Reactivity :: Hardware responds to stimulii, interacts with its environment ($\rightarrow$ requires handling of exceptions)
   - Hardware data types :: bit, bit-vector, multi-valued logic, signed and unsigned integer types, fixed-point types

Modeling

Modeling Using SystemC – How does one get there using C/C++



1. Restrict to synthesizable subset
2. Extend the language with hardware related components
3. The major requirements which need to be satisfied are :-

- Allow hardware/software co-design and co-verification
- Fast simulation for validation and optimization
- Smooth path to hardware and software
- Support of design and architectural reuse

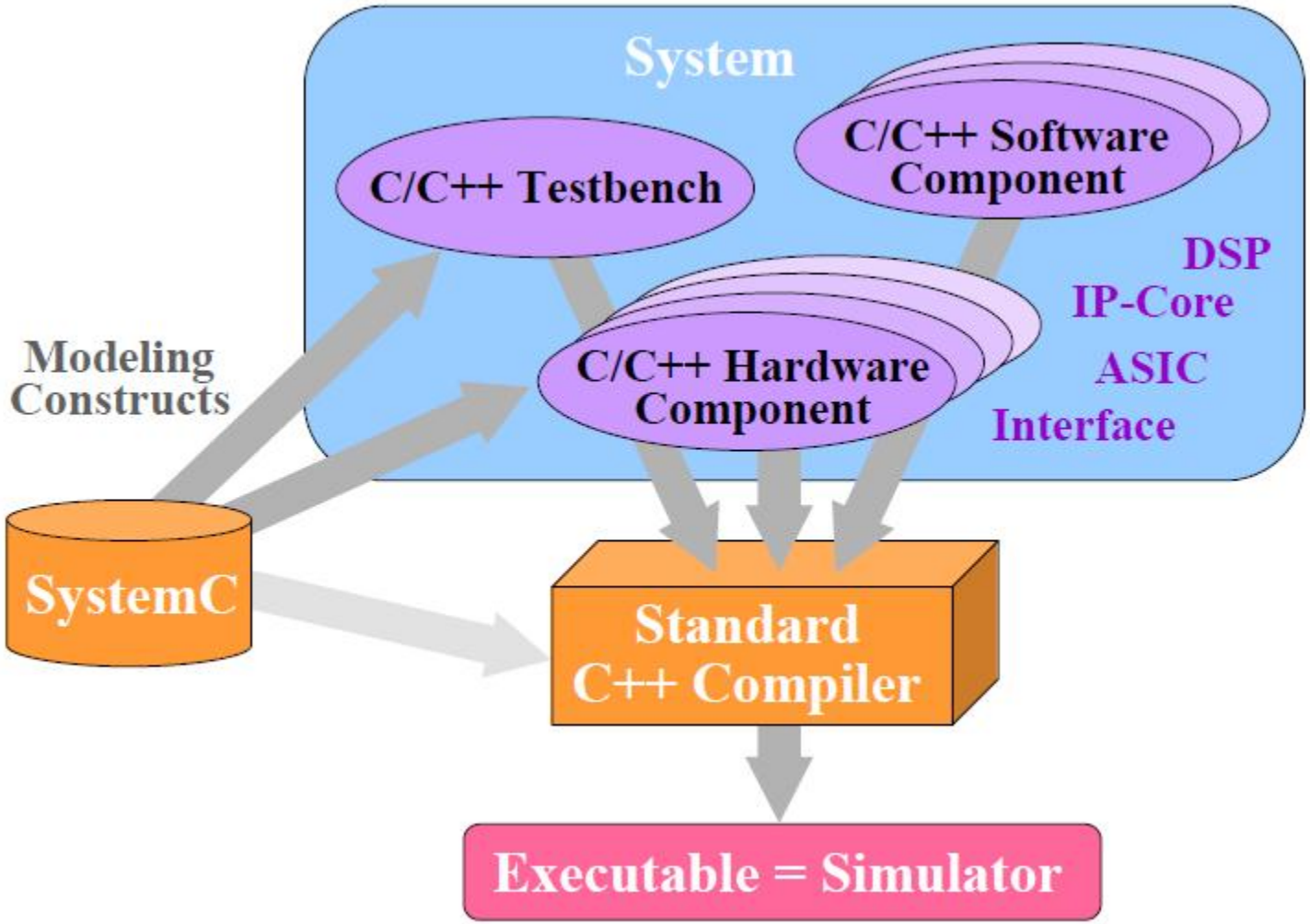
Modeling

Modeling Using SystemC – What is SystemC

1. A library of C++ classes
   - Processes (for concurrency)
   - Clocks (for time)
   - Modules, ports, signals (for hierarchy)
   - Waiting, watching (for events)
   - Hardware data types
2. A modeling style :: for modeling systems consisting of multiple design domains, abstraction levels, architectural components, real-life constraints
3. A light weight simulation kernel
   - Allow hardware/software co-verification

Modeling

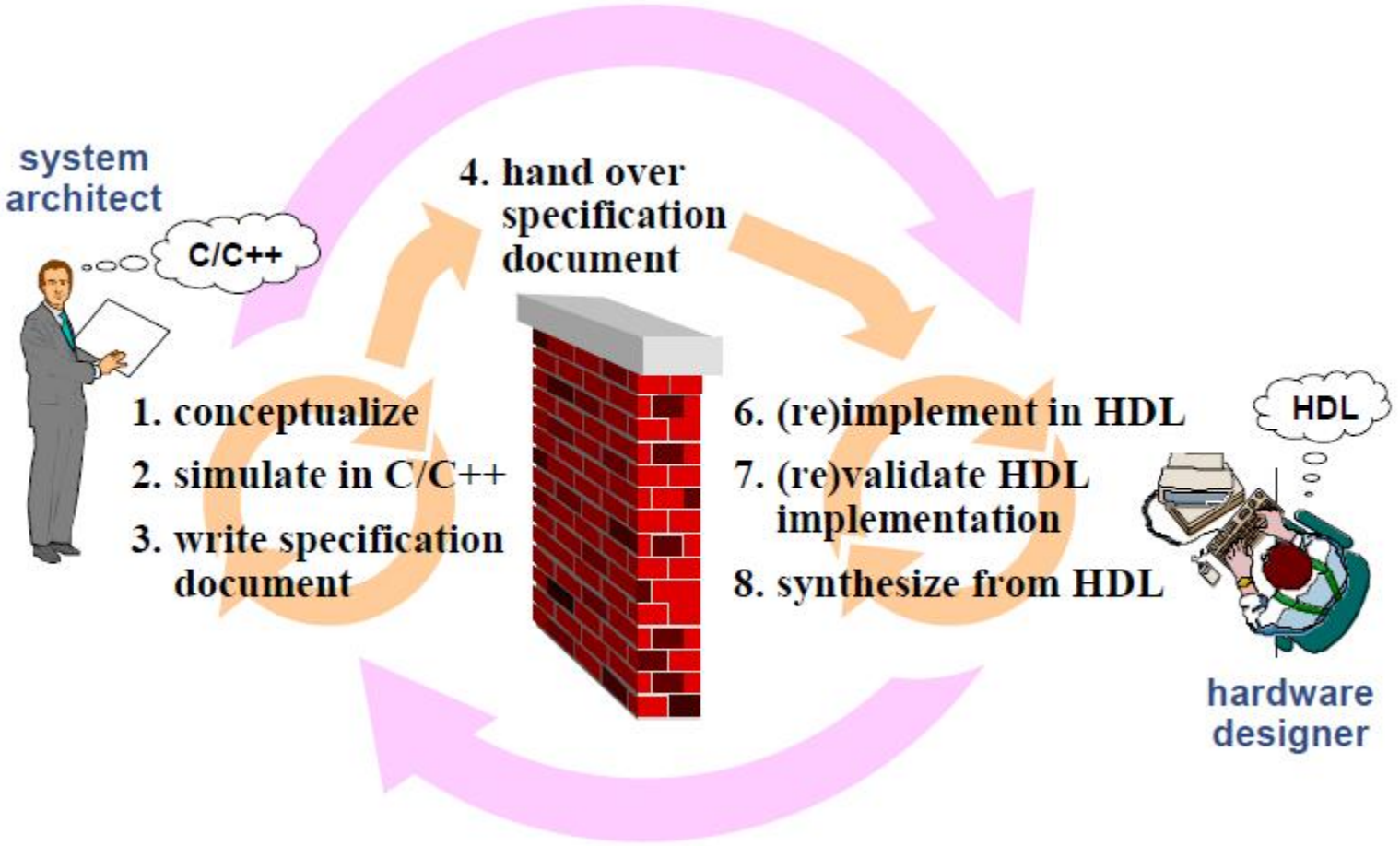
Modeling Using SystemC – How Does SystemC Work?



1. The approach is to promote a standard C/C++ modeling platform to model and exchange system level components and IP as also to build inter-operable tools infrastructure.

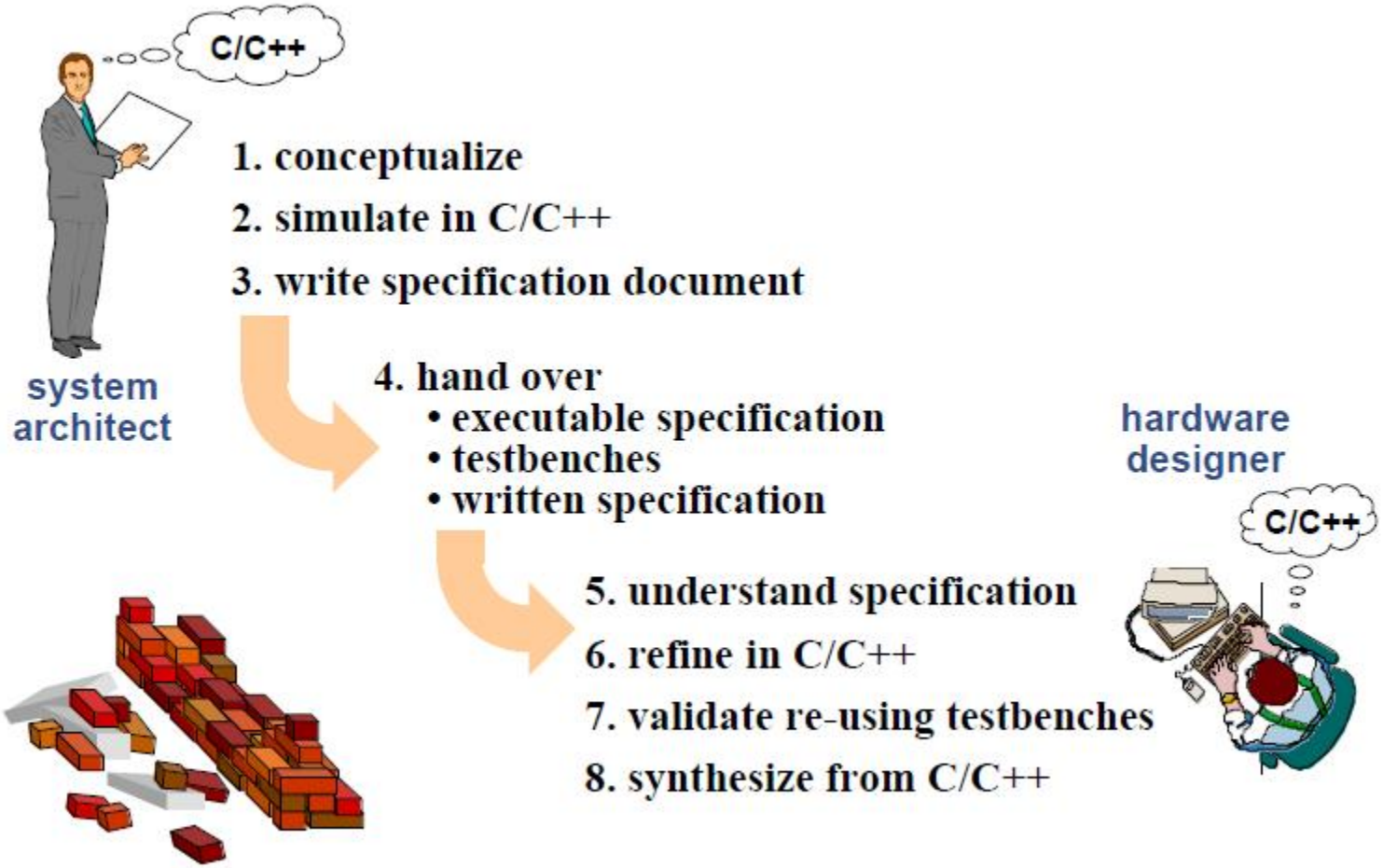
Modeling

Modeling Using SystemC – How Did it Work earlier ?



Modeling

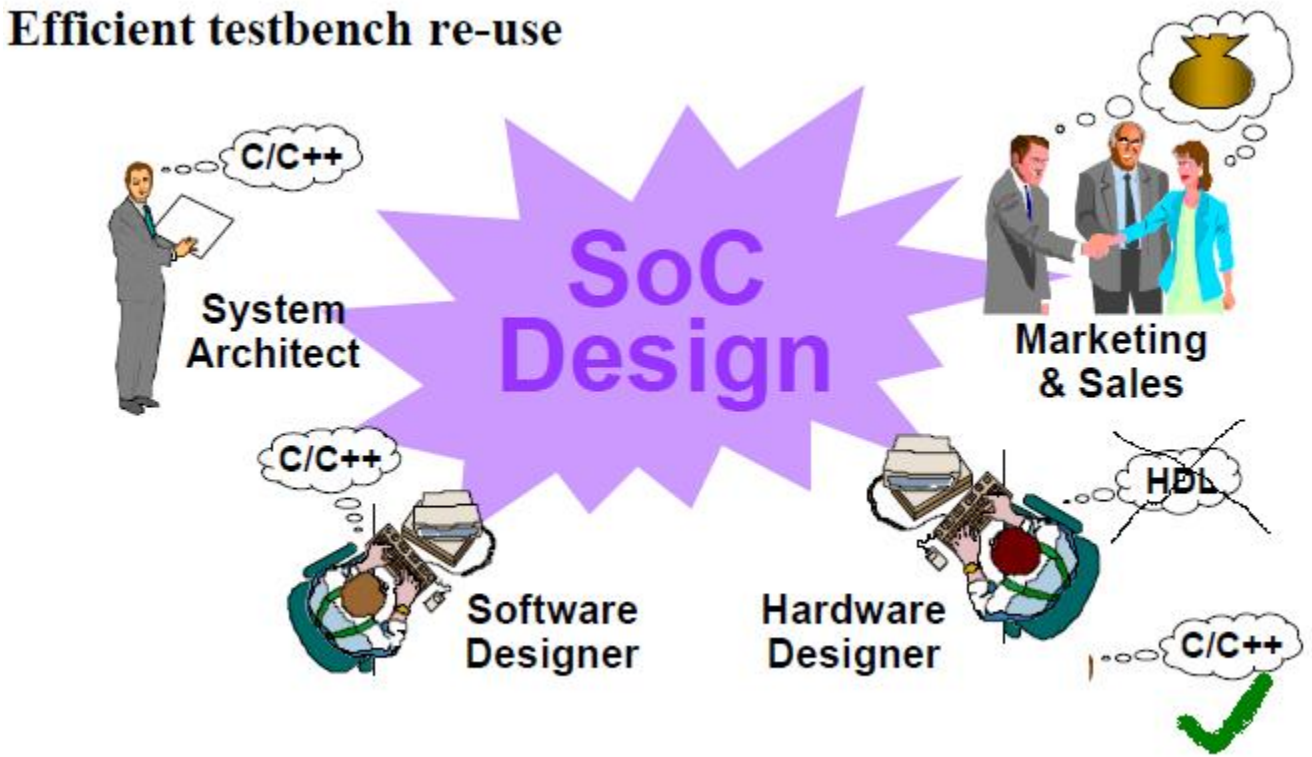
Modeling Using SystemC – What is the new Methodology?



1. The approach is to promote a standard C/C++ modeling platform to model and exchange system level components and IP as also to build inter-operable tools infrastructure.

Modeling

Modeling Using SystemC – What is the new Methodology?

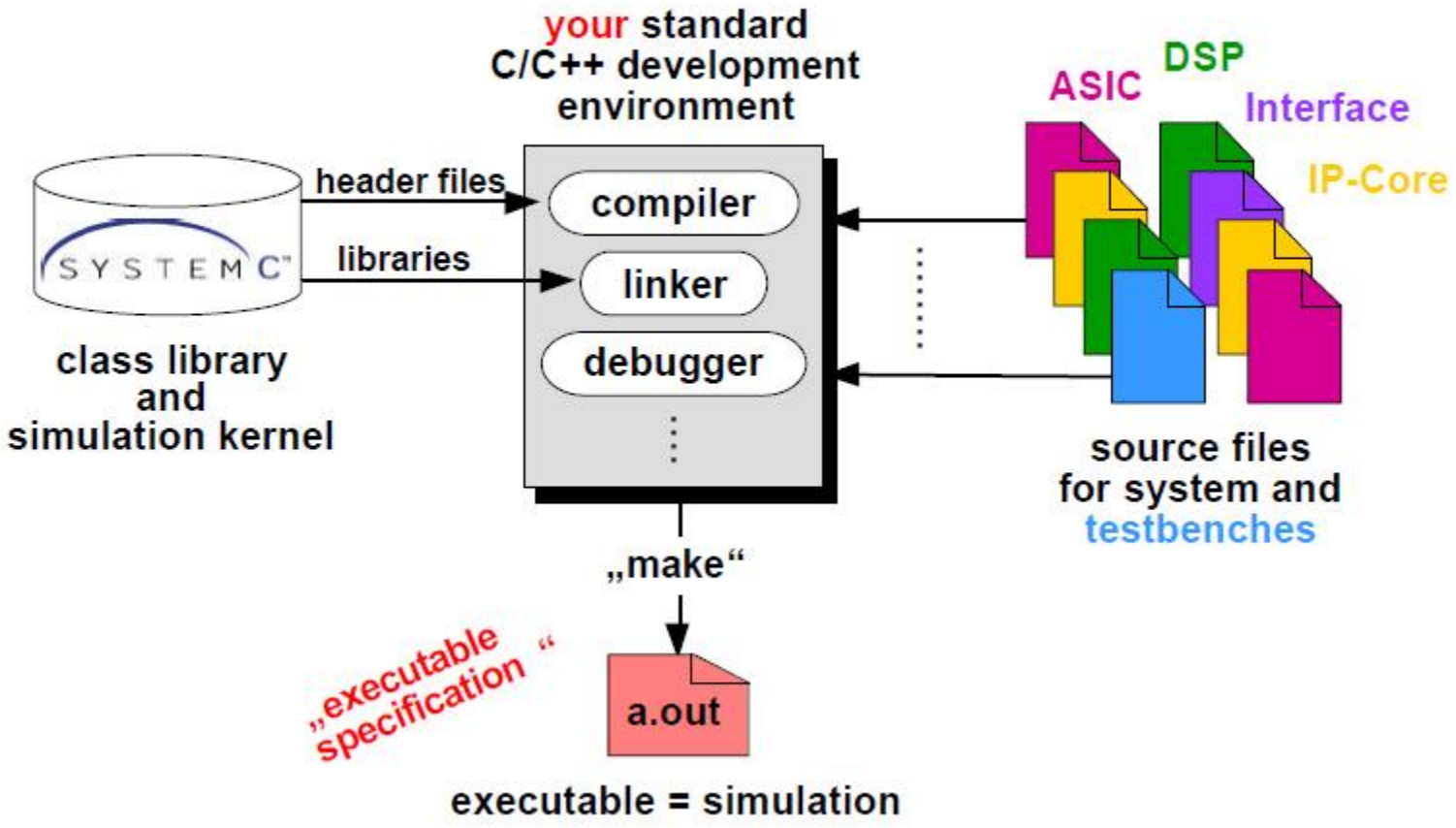


1. **Everyone talks the same language**

2. The approach is to promote a standard C/C++ modeling platform to model and exchange system level components and IP as also to build inter-operable tools infrastructure.

Modeling

Modeling Using SystemC – SystemC Design Methodology



Modeling

Modeling Using SystemC – Modules

1. Modules are basic building blocks of a SystemC design

2. A module contains processes (functionality) and / or sub-modules (structural hierarchy)

```
SC_MODULE( module_name ) {
    // Declaration of module ports
    // Declaration of module signals
    // Declaration of processes
    // Declaration of sub-modules
    SC_CTOR( module_name ) {          // Module constructor
        // Specification of process type and sensitivity
        // Sub-module instantiation and port mapping
    }
    // Initialization of module signals
};
```

3. A module corresponds to a C++ class
   - class data members :: ports
   - class member functions :: processes
   - class constructor :: process generation

Modeling

Modeling Using SystemC – Ports

1. Ports are the external interface(s) of a module
2. Passes data from and to processes / sub-modules
3. Used to trigger actions within the module
4. A port has a **mode** (direction) and a **type** (in, out, inout)

type: **C++ type, SystemC type, user-defined type**

```
// input port declaration
sc_in< type >  in_port_name;

// output port declaration
sc_out< type >  out_port_name;

// bidirectional port declaration
sc_inout< type >  inout_port_name;
```

Vector port / port array:

```
sc_out< int > result [32];
```

Modeling

Modeling Using SystemC – Signals

1. Signals connect port of one module to the port of another module
2. Signals are local to a module
3. Signals semantics are similar to VHDL and Verilog assignment semantics
4. A signal has a **type**

type: C++ type, SystemC type, user-defined type

```
// signal declaration
sc_signal< type >  signal_name;
```

Vector signal / signal array:

```
sc_signal< double > a[4];
```

5. NOTE :: Internal Storage of data is through local variables (not signals)

Modeling

Modeling Using SystemC – Ports & Signals Binding

1. Ports and Signals to be bound need to have the same type
2. A signal connects two ports
3. A port is bound to one signal (port-to-signal) or to one sub-module port (port-to-port)

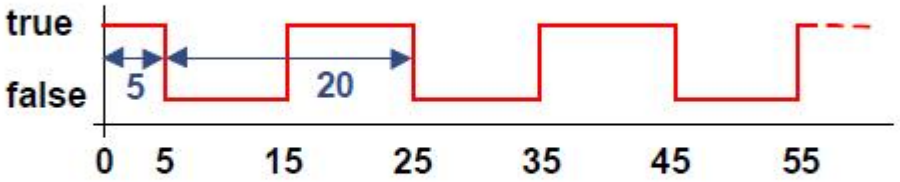
Modeling

Modeling Using SystemC – Clocks

1. SystemC provides a special object `sc_clock`
2. Clocks generate timing signals to synchronize events
3. Multiple clocks with arbitrary phase relationships are supported
4. Clock generation & binding

sc_clock *clock_name* ("*label*", *period*, *duty_ratio*, *offset*, *start_value*);

Example: sc_clock *my_clk* ("CLK", 20, 0.5, 5, true);



binding:

Example: *my_module*.clk(*my_clk*.signal());

Modeling

Modeling Using SystemC – Data Types

1. SystemC supports
   - Native C/C++ types
   - SystemC types
   - User-defined types
2. SystemC types
   - 2-value ('0', '1') logic / logic vector
   - 4-value ('0', '1', 'Z', 'X') logic / logic vector
   - Arbitrary sized integer (signed / unsigned)
   - Fixed point types (signed/unsigned, templated/untemplated)

Modeling

Modeling Using SystemC – Data Types

1. SystemC types

Type	Description
sc_bit	2-value single bit
sc_logic	4-value single bit
sc_int	1 to 64 bit signed integer
sc_uint	1 to 64 bit unsigned integer
sc_bigint	arbitrary sized signed integer
sc_biguint	arbitrary sized unsigned integer
sc_bv	arbitrary length 2-value vector
sc_lv	arbitrary length 4-value vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

2. Native C/C++ types
   - Integer types :: char, unsigned char, short, unsigned short, int, unsigned int long, unsigned long
   - Floating point types :: float, double, long double

Modeling

Modeling Using SystemC – Data Types

1. sc_bit :: 2-value single bit type :: '0' = false, '1' = true

2. sc_logic :: 4-value single bit type :: '0' = false, '1' = true, 'X' = unknown / indeterminate value, 'Z' = high-impedence / floating value

3. SystemC allows for mixed use of operand types sc_bit and sc_logic. Use of character literals for constant assignments is allowed.

sc_bit / sc_logic operators

Bitwise	**&** (and)	**|** (or)	**^** (xor)	**~** (not)
Assignment	**=**	**&=**	**|=**	**^=**
Equality	**==**	**!=**		

Modeling

Modeling Using SystemC – Data Types

1. sc_int$< n >$ :: Signed Fixed point integer type :: $(n : wordLength, 1 \leq n \leq 64)$
2. sc_uint$< n >$ :: UnSigned Fixed point integer type :: $(n : wordLength, 1 \leq n \leq 64)$
3. sc_bigint$< n >$ :: Signed Fixed point integer type :: $(n : wordLength, n \geq 64)$
4. sc_biguint$< n >$ :: UnSigned Fixed point integer type :: $(n : wordLength, n \geq 64)$
5. Features ::
   - Mixed use of operand types sc_int, sc_uint, sc_bigint, sc_biguint and C++ integer types
   - Truncation and / or sign extension can be done if required
   - 2ś complement representation

sc_int / sc_uint / sc_bigint / sc_biguint operators									
Bitwise	&	|	^	~	>>	<<			
Arithmetic	+	-	*	/	%				
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Auto-Ink/Dek	++	--							
Bit/Part Select	[]	range()							
Concatenation	(,)								

Modeling

Modeling Using SystemC – Data Types

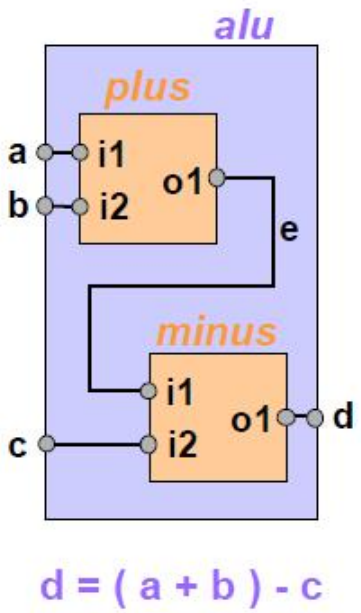
1. sc_bv$< n >$ :: Aribitrary length bit vector :: $(n : vectorLength)$
2. sc_lv$< n >$ :: Aribitrary length logic vector :: $(n : vectorLength)$
3. Features ::
   - Assignment between sc_bv and sc_lv
   - Use of string literals for vector constant assignments
   - Conversions between sc_bv/ sc_lv ans SystemC integer types
   - No arithmetic operation available

sc_bv / sc_lv								
Bitwise	&	|	^	~	>>	<<		
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=						
Bit/Part Select	[]	range()						
Concatenation	(,)							
Reduction	and_reduction()	or_reduction()	xor_reduction()					
Conversion	to_string()							

Modeling

Modeling Using SystemC – Modules and Hierarchies

1. It is possible to have a module containing sub-modules to form an hierarchical structure.




```
SC_MODULE( plus ) {
  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;
  .....
};

SC_MODULE( minus ) {
  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;
  .....
};
```

```
SC_MODULE( alu ) {
  sc_in<int>  a;
  sc_in<int>  b;
  sc_in<int>  c;
  sc_out<int> d;

  plus  *p;
  minus *m;

  sc_signal<int> e;

  SC_CTOR( alu ) {
    p = new plus ( "PLUS" );
    p->i1 (a);
    p->i2 (b);
    p->o1 (e);

    m = new minus ( "MINUS" );
    (*m) (e,c,d);
  }
};
```

Modeling

Modeling Using SystemC – Processes

1. Process
   - Encapsulates functionality
   - Basic unit of concurrent execution
   - Not Hierarchical
2. Process Activation
   - Processes have sensitivity lists
   - Processes are triggered by events on sensitive signals
3. Process Types
   - Method (SC_METHOD) :: asynchronous block, like a sequential function
   - Thread (SC_THREAD) :: asynchronous process
   - Clocked Thread (SC_CTHREAD) :: synchronous process

	SC_METHOD	SC_THREAD	SC_CTHREAD
triggered	by signal events	by signal events	by clock edge
infinite loop	no	yes	yes
execution suspend	no	yes	yes
suspend & resume	-	wait()	wait() wait_until()

construct & sentisize method	SC_METHOD(*p*);
	
	sensitive(*s*);
	sensitive_pos(*s*);
	sensitive_neg(*s*);
		SC_THREAD(*p*);
		
		sensitive(*s*);
		sensitive_pos(*s*);
		sensitive_neg(*s*);
			SC_CTHREAD (*p*,*clock*.pos());
			
			SC_CTHREAD (*p*,*clock*.neg());

modeling example (hardware)	combinational logic	sequential logic at RT level (asynchronous reset, etc.)	sequential logic at higher design levels

Modeling

Modeling Using SystemC – Processes

Example: SC_METHOD

```
SC_MODULE( plus ) {
  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus ) {
    SC_METHOD( do_plus );
    sensitive << i1 << i2;
  }
};
```

```
void plus::do_plus()  {
  int arg1;
  int arg2;
  int sum;

  arg1 = i1.read();
  arg2 = i2.read();
  sum = arg1 + arg2;
  o1.write(sum);
}
```

```
void plus::do_plus()  {
  o1 = i1 + i2;
}
```

Modeling

Modeling Using SystemC – Processes

Example: SC_THREAD

```
SC_MODULE( plus ) {
  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus ) {
    SC_THREAD( do_plus );
    sensitive << i1 << i2;
  }
};
```

```
void plus::do_plus() {
  int arg1;
  int arg2;
  int sum;

  while ( true ) {
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);

    wait();
  }
}
```

Modeling

Modeling Using SystemC – Processes

Example: SC_CTHREAD

```
SC_MODULE( plus ) {
  sc_in_clk  clk;

  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus ) {
    SC_CTHREAD( do_plus, clk.pos() );
  }
};
```

```
void do_plus() {
  int arg1;
  int arg2;
  int sum;

  while ( true ) {
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);

    wait();
  }
}
```

Modeling

Modeling Using SystemC – Waiting and Watching

1. Suspend / reactivate process execution :: `wait()` :: SC_THREAD, SC_CTHREAD
2. Halt process execution until an event occurs :: `wait_until()` :: SC_CTHREAD ONLY
3. Transfer control to a special code sequence if a specific condition occurs ::
   `watching(reset.delayed() == true)`

Modeling

Modeling Using SystemC – Simulation Kernel Scheduler Steps

Step 1: All clock signals that change their valueat the current time are assigned their new value.

Step2: All *SC_METHOD* / *SC_THREAD* processes with inputs that have changed are executed. The entire bodies of *SC_METHOD* processes are executed. *SC_THREAD* processes are executed until the next *wait()* statement suspends execution. *SC_METHOD* / *SC_THREAD* processes are not executed in a fixed order.

Step3: All *SC_CTHREAD* processes that are triggered have their outputs updated and are saved in a queue to be executed in step 5. All outputs of *SC_METHOD* / *SC_THREAD* processes that were executed in step 1 are also updated.

Step 4: Step 2 and step 3 are repeated until no signal changes ist value.

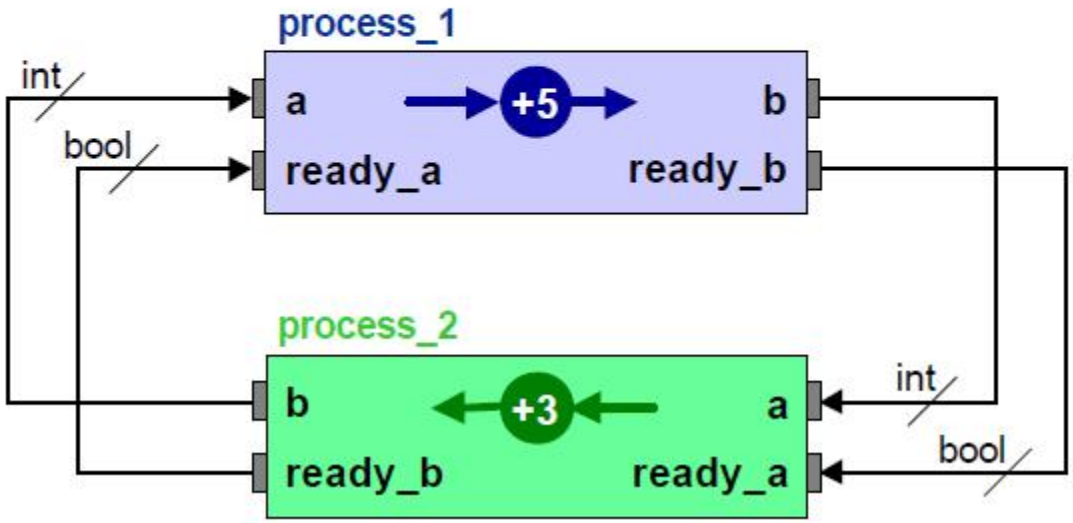
Step 5: All *SC_CTHREAD* processes that were triggered and queued in step 3 are executed. There is no fixed execution order of these processes. Their outputs are updated at the next active edge (when step 3 is executed), and therefore are saved internally.

Step6: Simulation time is advanced to the next clock edge and the scheduler goes back to step 1.

Modeling

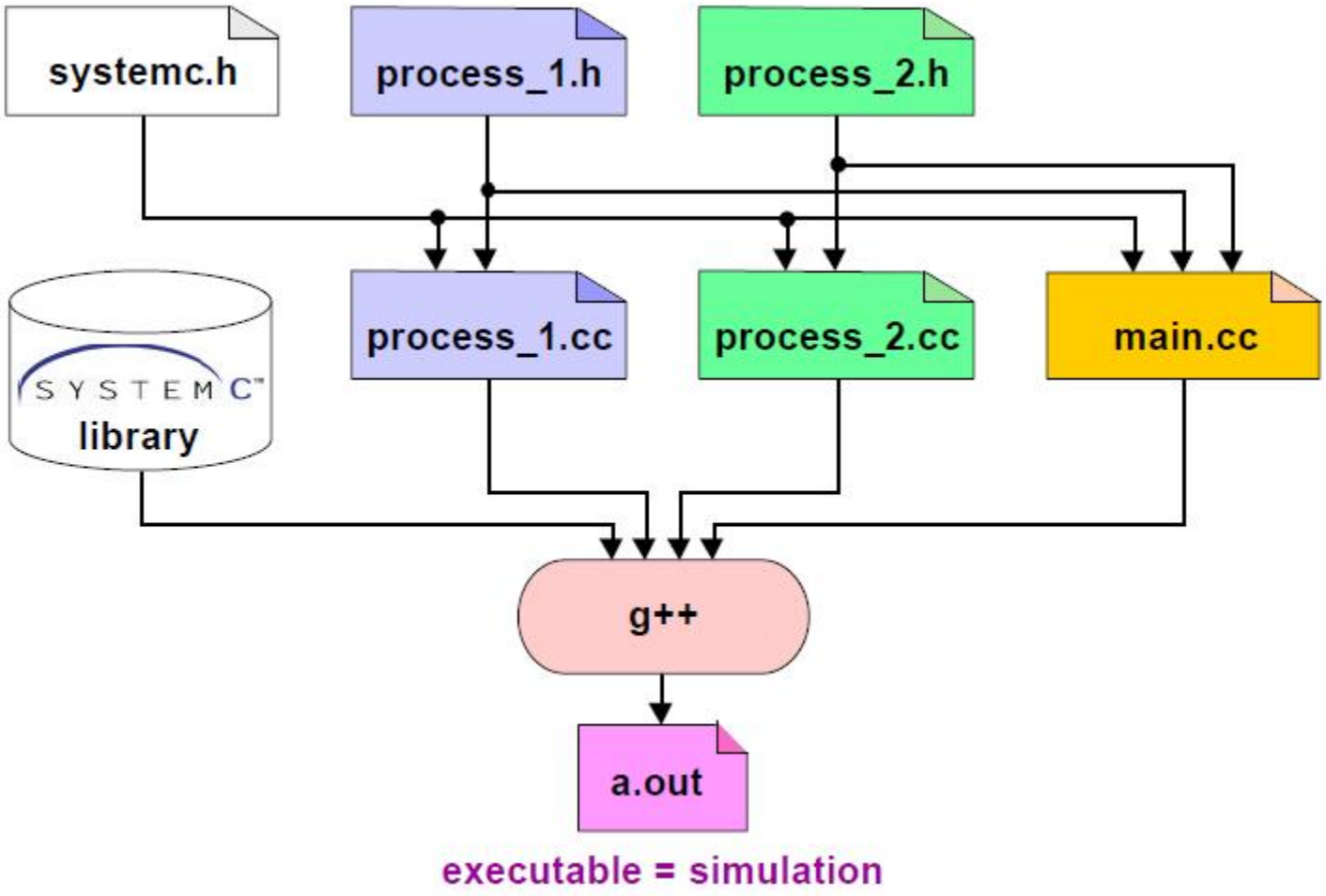
Modeling Using SystemC – Example

Two processes (process_1 and process_2) alternately incrementing an integer value



Modeling

Modeling Using SystemC – Example – Typical File Structure



Modeling

Modeling Using SystemC – Example

```
// header file: process_1.h

SC_MODULE( process_1 ) {

    // Ports
    sc_in_clk clk;
    sc_in<int> a;
    sc_in<bool> ready_a;
    sc_out<int> b;
    sc_out<bool> ready_b;

    // Process functionality
    void do_process_1();

    // Constructor
    SC_CTOR( process_1 ) {
        SC_CTHREAD( do_process_1 , clk.ps() );
    }

};
```

Module: process_1

```
// implementation file: process_1.cc

#include "systemc.h"
#include "process_1.h"

void process_1::do_process_1()
{
    int v;

    while ( true )
    {
        wait_until( ready_a.delayed() == true );
        v = a.read();
        v += 5;
        cout << "P1: v = " << v << endl;
        b.write( v );

        ready_b.write( true );
        wait();
        ready_b.write( false );
    }
}
```

Modeling

Modeling Using SystemC – Example

```
// header file: process_2.h

SC_MODULE( process_2 ) {

    // Ports
    sc_in_clk clk;
    sc_in<int> a;
    sc_in<bool> ready_a;
    sc_out<int> b;
    sc_out<bool> ready_b;

    // Process functionality
    void do_process_2();

    // Constructor
    SC_CTOR( process_2 ) {
        SC_CTHREAD( do_process_2 , clk.ps() );
    }

};
```

Module: process_2

```
// implementation file: process_2.cc

#include "systemc.h"
#include "process_2.h"

void process_2::do_process_2()
{
    int v;

    while ( true )
    {
        wait_until( ready_a.delayed() == true );
        v = a.read();
        v += 3;
        cout << "P2: v = " << v << endl;
        b.write( v );

        ready_b.write( true );
        wait();
        ready_b.write( false );
    }
}
```

Modeling

Modeling Using SystemC – Example

```
// implementation file: main.cc

#include "systemc.h"
#include "process_1.h"
#include "process_2.h"

int  sc_main  (int ac,char *av[])
{
    sc_signal<int> s1 ( "Signal-1" );
    sc_signal<int> s2 ( "Signal-2" );
    sc_signal<bool> ready_s1 ( "Ready-1" );
    sc_signal<bool> ready_s2 ( "Ready-2" );

    sc_clock clock( "Clock" , 20 , 0.5 , 0.0 );

    process_1 p1 ( "P1" );
    p1.clk( clock );
    p1.a( s1 );
    p1.ready_a( ready_s1 );
    p1.b( s2 );
    p1.ready_b( ready_s2 );
```

Top-Level Module: main

```
    process_2 p2 ( "P2" );
    p2.clk( clock );
    p2.a( s2 );
    p2.ready_a( ready_s2 );
    p2.b( s1 );
    p2.ready_b( ready_s1 );

    s1.write(0);
    s2.write(0);
    ready_s1.write(true);
    ready_s2.write(false);

    sc_start(100000);

    return 0;
}
```

Modeling

Acknowledgements

1. Hardware / Software Co-Design - Principles and Practice :: J. Staunstrup & W. Wolf :: Ch 1.3

2. Lecture Notes :: http://www.facweb.iitkgp.ernet.in/ãnupam/language_ab.ppt

3. Lecture Notes :: Joachim Gerlach :: System-on-Chip Design with System C :: University of Tubingen (Dept of Computer Engg.)

4. IEEE Standard SystemC Language Reference Manual :: 1666^{TM} - 2005