McGill

ECSE 421 Lecture 7: More DES; Von Neumann MoC

ESD Chapter 2

Last Time

- Introduction to Discrete Event Simulation
  - VHDL
  - VHDL simulation cycle
- Multi-value Logic

Where Are We?

W	D	Date		Topic	ESD	PES	Out	In	Notes
1	T	12-Jan-2016	L01	Introduction to Embedded System Design	1.1-1.4				
	R	14-Jan-2016		Introduction to Embedded System Design	1.1-1.4				
2	T	19-Jan-2016	L02	Specifying Requirements / MoCs / MSC	2.1-2.3				
	R	21-Jan-2016	L03	CFSMs	2.4				
3	T	26-Jan-2016	L04	Data Flow Modeling	2.5	3.1-5,7	LA1		
	R	28-Jan-2016	L05	Petri Nets	2.6				Thru Slide 21
4	T	2-Feb-2016	L06	Discrete Event Models	2.7	4			G: Zaid Al-bayati
	R	4-Feb-2016	L07	DES / Von Neumann Model of Computation	2.8-2.10	5	LA2	LA1	
5	T	9-Feb-2016	L08	Sensors	3.1-3.2	7.3,12.1-6			
	R	11-Feb-2016	L09	Processing Elements	3.3	12.6-12			
6	T	16-Feb-2016	L10	More Processing Elements / FPGAs			LA3	LA2	
	R	18-Feb-2016	L11	Memories, Communication, Output	3.4-3.6				
7	T	23-Feb-2016	L12	Embedded Operating Systems	4.1				
	R	25-Feb-2016		***Midterm exam: in-class, closed book***			P	LA3	Chapters 1-3
	T	1-Mar-2016		***No class***					Winter break
	R	3-Mar-2016		***No class***					Winter break
8	T	8-Mar-2016	L13	Middleware	4.4-4.5				
	R	10-Mar-2016	L14	Performance Evaluation	5.1-5.2				
9	T	15-Mar-2016	L15	More Evaluation and Validation	5.3-5.8				
	R	17-Mar-2016	L16	Introduction to Scheduling	6.1-6.2.2				
10	T	22-Mar-2016	L17	Scheduling Aperiodic Tasks	6.2.3-6.2.4				
	R	24-Mar-2016	L18	Scheduling Periodic Tasks	6.2.5-6.2.6				

Today

- More Discrete Event Simulation
  - SystemC
  - Verilog, SystemVerilog
  - SpecC
- Imperative (von Neumann) Models
- Comparing Models of Computation
- Mixing Models of Computation
- Levels of Modeling Abstraction

MoCs Considered in 421

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	**VHDL*, Verilog*, SystemC*, …**	**Only experimental systems, e.g. distributed DE in Ptolemy**	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

Other Discrete Event Environments

- SystemC
- Verilog and SystemVerilog
- SpecC

Using C for ES Design: Motivation

- Many standards (*e.g.* the GSM and MPEG) are published as C programs
  - Standards have to be translated if special hardware description languages are to be used
- Many systems implement features with a mix of hardware and software components
  - Hardware simulators and software simulators
  - Simulations require an interface *unless* the same language is used for the description of both hardware and software
- SystemC and others attempt to describe software and hardware in the same language
  - Easier said than implemented!
  - Various C dialects used for hardware description

Drawbacks of a C/C++ Design Flow

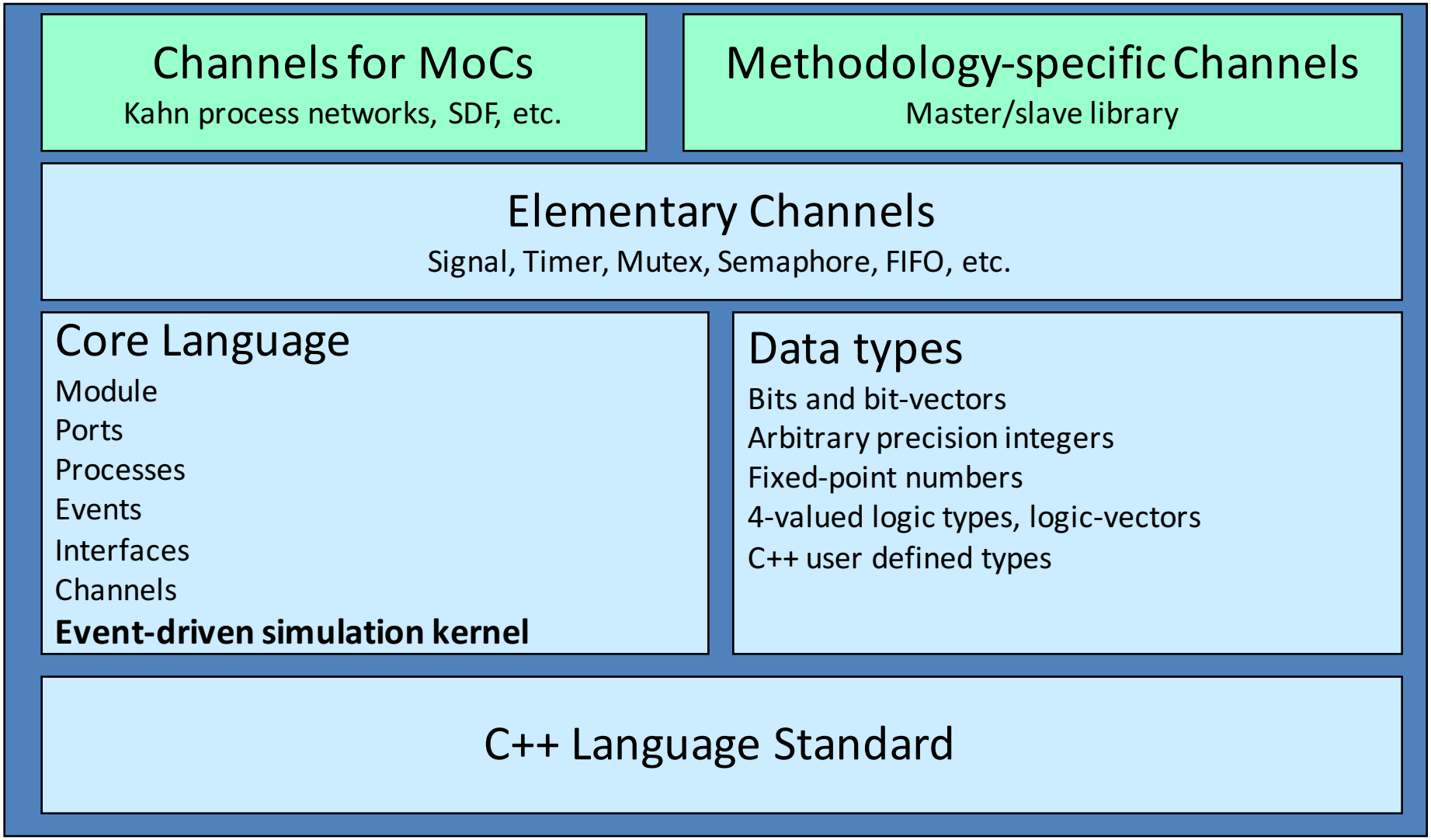
- C/C++ was not created to design hardware
- C/C++ does not support
  - *Hardware style communication*: signals, protocols
  - *Notion of time*: clocks, time sequenced operations
  - *Concurrency*: hardware is concurrent, operates in ||
  - *Reactivity*: hardware is reactive, responds to stimuli, interacts with environment (e.g., exceptions, interrupts)
  - *Hardware data types*: bit, bit-vector, multi-valued logic, signed and unsigned integer, fixed-point
- Missing links to hardware during debugging

SystemC: Required Features

- Solutions for modeling HW in a SW language:
  - C++ class library including required functions
  - *Concurrency*: via processes, controlled by sensitivity lists*
    - and calls to wait primitives
  - *Time*: floating point numbers in SystemC 1.0
    - Integer values in SystemC 2.0
    - Includes units such as ps, ns, µs, etc*
  - *Support of bit-datatypes*
    - Bit vectors of different lengths
    - 2- and 4-valued logic with built-in resolution*
  - *Communication*: plug-and-play (pnp) channel model, allowing easy replacement of intellectual property (IP)
  - Determinate behavior not guaranteed

* Good to know VHDL ☺

SystemC Language Architecture



Transaction-level Modeling

- Separates communication between modules from
  - Module implementation and
  - Communication architecture
- Emphasis is on the functionality of data transfers
  - E.g., buses and FIFOs are modeled as channels
  - Transactions call interface functions which hide the implementation details of the channels
- Eases design space exploration
  - E.g., designers can experiment with different bus architectures without re-developing individual modules
- Facilitates design refinement

Verilog

- HW description language competing with VHDL
- Standardized:
  - IEEE 1364-1995 (Verilog version 1.0)
  - IEEE 1364-2001 (Verilog version 2.0)
  - Features that are similar to VHDL:
    - Designs described as connected entities
    - Bit vectors and time units are supported
  - Features that are different:
    - Built-in support for 4-value, 8-strength logic
    - More features for transistor-level descriptions
    - Less flexible than VHDL
    - More popular in the US (VHDL common in Europe)

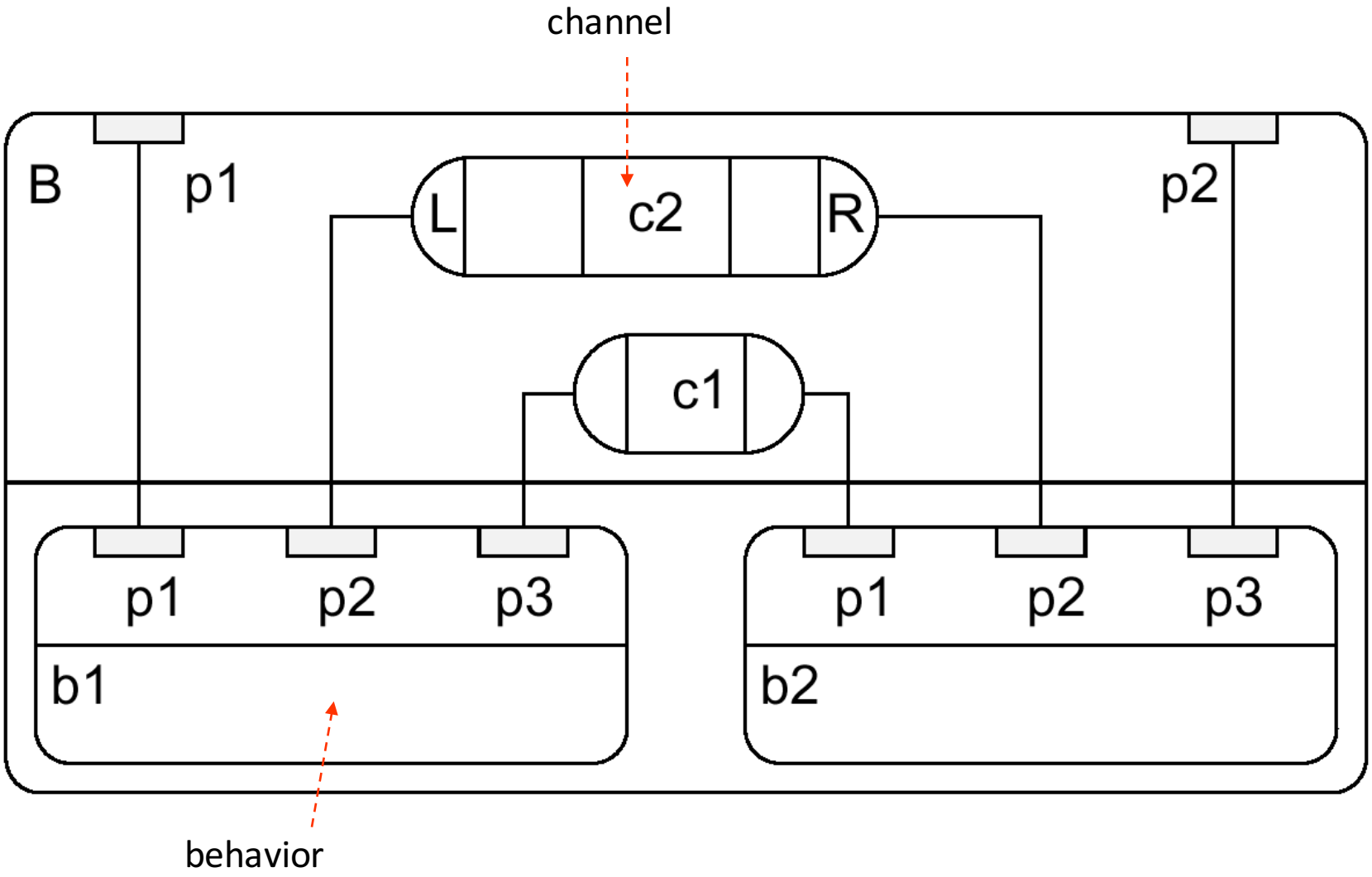
SystemVerilog

- Corresponds to Verilog versions 3.0 and 3.1
- Includes:
  - Additional language elements for modeling behavior
  - C data types such as **`int`**
  - Type definition facilities
  - Definition of interfaces of HW components as entities
  - Mechanism for calling C/C++-functions from Verilog
  - Limited mechanism for calling Verilog functions from C
  - Enhanced features for describing the testbench
  - Dynamic process creation
  - Inter-process communication and synchronization
  - Automatic memory allocation and deallocation
  - Interface for formal verification

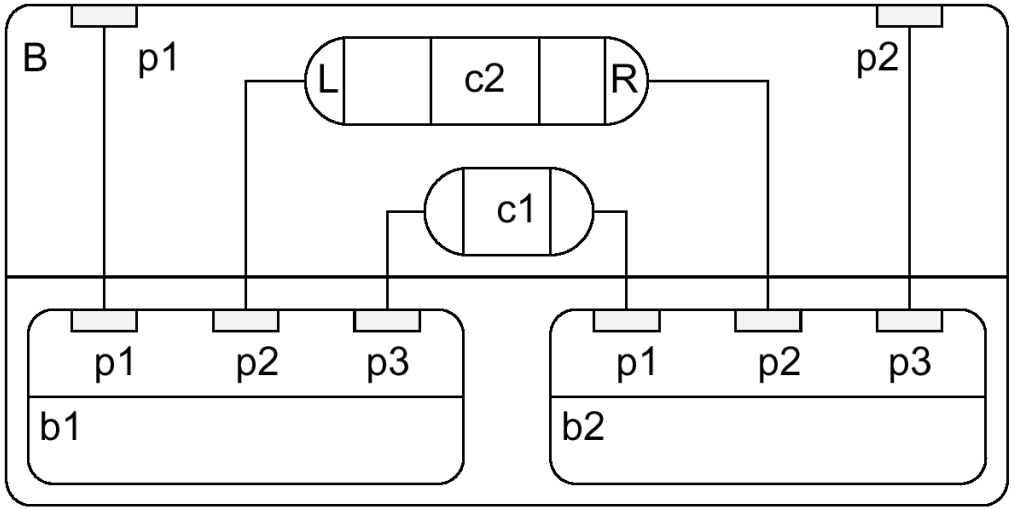
SpecC [Gajski, Dömer *et al.* 2000]

- Separates communication and computation
  - Enables "plug-and-play" for system components
  - Models systems as hierarchical networks of behaviors communicating through channels
- Models consists of behaviors, channels and interfaces
- **Behaviors** include ports, locally instantiated components, private variables and functions and a public main function
- **Channels** encapsulate communication
  - Include variables and functions used for the definition of a communication protocol
- **Interfaces** link behaviors and channels
  - Declare communication protocols (defined in a channel)

Example



SpecC-Example




```
interface L {void Write(int x);};
interface R {int Read (void);};
channel C implements L,R
        {
                int Data; bool Valid;
                void Write(int x)  {Data=x; Valid=true;}
                int Read(void) {while (!Valid) waitfor(10); return (Data);}
        };
behavior B1 (in int p1, L p2, in int p3)
        { void main(void)    {/*...*/ p2.Write(p1);} };
behavior B2 (out int p1, R p2, out int p3)
        { void main(void)   {/*...*/ p3=p2.Read(); } };
behavior B(in int p1, out int p2)
        {
                int c1; C c2; B1 b1(p1,c2,c1); B2 b2 (c1,c2,p2);
                void main (void)
                { par {b1.main();b2.main();} }
        };
```

MoCs Considered in 421

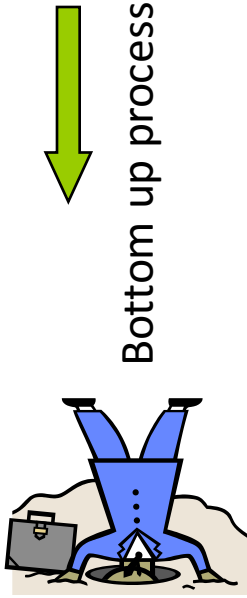
Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	**C, C++, Java**	**C, C++, Java with libraries CSP, ADA |**	

Imperative (von Neumann) Model

- The von Neumann model reflects the principles of operation of standard computers
- Sequential execution of instructions
  - Sequential control flow, fixed sequence of operations
- Possible branches
- Partitioning of applications into threads
- Context switching between threads
  - Frequently based on pre-emption (cooperative multi-tasking or time-triggered context switching is less common)
- Access to shared memory

From Implementation to Programming

- Example Languages
  - Machine languages (binary)
  - Assembly languages (mnemonics)
  - Imperative languages (C, C++, Java, …)
    - Provide a limited abstraction of machine languages



- Threads/Processes
  - Initially managed by the operating system
  - Now available to the programmer as well
  - Languages initially not designed for communication
    - Threads mandate synchronization and communication

Communication via Shared Memory

- Several threads access the same memory
  - Very fast communication technique (no extra copying)
  - Potential race conditions:

```
thread a {
  u = 1;
  if u<5 {u = u + 1; ..}
}
```

```
thread b {
  ..
  u = 5
}
```

- Context switch after the test could result in u == 6.

⇒ Inconsistent results are possible

⇒ *Critical sections* = sections at which exclusive access to resource *r* (*e.g.* shared memory) must be guaranteed

Shared Memory

```
thread a {
  u = 1; ..
  P(S)  //obtain mutex
  if u<5 {u = u + 1; ..}
  // critical section
  V(S)  //release mutex
}
```

```
thread b {
  ..
  P(S)  //obtain mutex
  u = 5
  // critical section
  V(S)  //release mutex
}
```

- S: semaphore
- P(S) grants up to *n* concurrent accesses to resource
  - *n*=1 in this case (mutex/lock)
- V(S) increases number of allowed accesses to resource
- Imperative model should be supported by
  - mutual exclusion for critical sections
  - cache coherency protocols

CSP

- Communicating Sequential Processes [Hoare, 1985]
- Synchronous Message Passing
  - Rendez-vous-based communication
- Example:

```
process A
..
var a ...
  a:=3;
  c!a; -- output
end
```

```
process B
..
var b ...
  ...
  c?b; -- input
end
```

No race conditions (!)

ADA

- After Ada Lovelace
  - Said to be the 1st female programmer
- Synchronous Message Passing
- US Department of Defense (DoD) wanted to avoid using a multitude of programming languages
  - DoD defined their requirements
  - Selected a language from a set of competing designs (selected design based on PASCAL)
- ADA'95 is object-oriented extension of original ADA
- Introduces the task concept

Tasks in ADA

```
procedure example1 is
    task a;
    task b;
    task body a is
      -- local declarations for a
    begin
      -- statements for a
    end a;
```

```
    task body b is
      -- local declarations for b
    begin
      -- statements for b
    end b;
begin
  -- Tasks a and b will start before the first
  -- statement of the body of example1
end;
```

ADA Rendez-vous

```
task screen_out is
    entry call_ch(val:character; x, y: integer);
    entry call_int(z, x, y: integer);
end screen_out;
task body screen_out is
    ...
    select
        accept call_ch ...  do ..
        end call_ch;
    or
        accept call_int ... do ..
        end call_int;
    end select;
```

```
Sending  a message:
begin
 screen_out.call_ch('Z',10,20);
 exception
  when tasking_error =>
            (exception handling)
end;
```

Communication and Synchronization

- Communication libraries
  - Can add blocking, or non-blocking communication to von-Neumann languages like C, C++, Java, …
  - Examples will be presented in Chapter 4
    - System Software

Other Imperative Embedded Languages

- **Pearl**
  - Designed in Germany for process control applications
  - Dating back to the 70s
  - Used to be popular in Europe
  - Pearl News still exists
    (in German, see http://www.real-time.de/)
- **Chill**
  - Designed for telephone exchange stations
  - Based on PASCAL
  - http://psc.informatik.uni-jena.de/languages/chill/chill.htm

Java

- Potential benefits:
  - Clean and safe language
  - Supports currency with multi-threading
  - Platform independent
- Problems:
  - Large run-time libraries
    - Large memory requirements
  - No direct access to I/O (for safety reasons)
  - Non-deterministic garbage collection time
  - Non-deterministic thread dispatcher
  - Less efficient than C
  - No abstraction of time

Overview of Java 2 Editions

"J2ME … addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers, all the way up to the TV set-top box …"

Based on http://java.sun.com/products/cldc/wp/KVMwp.pdf

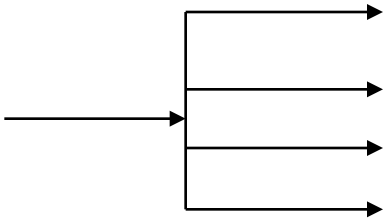
Deadlock

- Four conditions that lead to deadlock [Coffman, 1971]
  - **Mutual exclusion**: a shared resource that can't be used by more than one thread at a time
  - **Hold and wait**: a thread holding resources may request more
  - **No preemption**: resource cannot be forcibly removed from threads, they can be released only by the holding threads
  - **Circular wait**: *e.g.*, thread A waits for a resource held by B, which is waiting for a resource which is held by A
- Safety-critical hardware
  - No guaranteed way to ensure one of the above is false
- Non-safety-critical software
  - Infrequent deadlock is sufficient

Mutual Exclusion in Java

"The Observer pattern defines a one-to-many dependency between a subject object and

- *any number of observer objects*
- *so that when the subject object changes state,*
- *all its observer objects are notified and updated automatically."*



Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Addision-Wesley, 1995

Example: Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue);
    }
}
```

Will this work in a multithreaded context?

Thanks to Mark S. Miller for the details of this example.

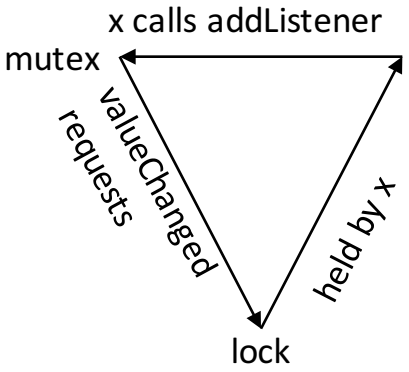
Adding Mutual Exclusion

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue);
    }
}
```

Javasoft recommends against this.
What's wrong with it?

Mutexes Using Monitors Are Minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue);
    }
}
```



valueChanged() may attempt to acquire a lock on some other object and stall.
If the holder of that lock calls addListener(): deadlock!

Not-So-Simple Observer Pattern

```
public synchronized void addListener(listener) {…}
public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make a copy of listeners to avoid race conditions

notify each listener outside of the synchronized block to avoid deadlock

This still isn't right. What's wrong with it?

How to Make it Right?

```
public synchronized void addListener(listener) {...}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object.

However, listeners may be notified in the opposite order. The listeners may be alerted to the value-changes in the wrong order!

Lee's Conclusion

- *Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans. …*
- **Succinct Problem Statement**
  - Threads are wildly non-deterministic
  - The programmer's job: *eliminate* non-determinism
    - By imposing constraints on execution order (*e.g.*, mutexes)
  - In the end, should programmers *improve threads*?
  - Or *replace them*?

[Edward Lee (UC Berkeley), Artemis Conference, Graz, 2007]

Imperative Languages, Shared Memory

- Potential for deadlock
- Specification of total order is over-specification
  - A partial order would be sufficient
  - Total ordering reduces the potential for optimizations
- Timing cannot be specified
- Access to shared memory leads to anomalies
  - Prune them away with mutexes, semaphores, monitors
- Access to shared resources leads to priority inversion
  - Chapter 4
- Termination in general undecidable
- Preemptions at any time complicate timing analysis

Comparing Models of Computation

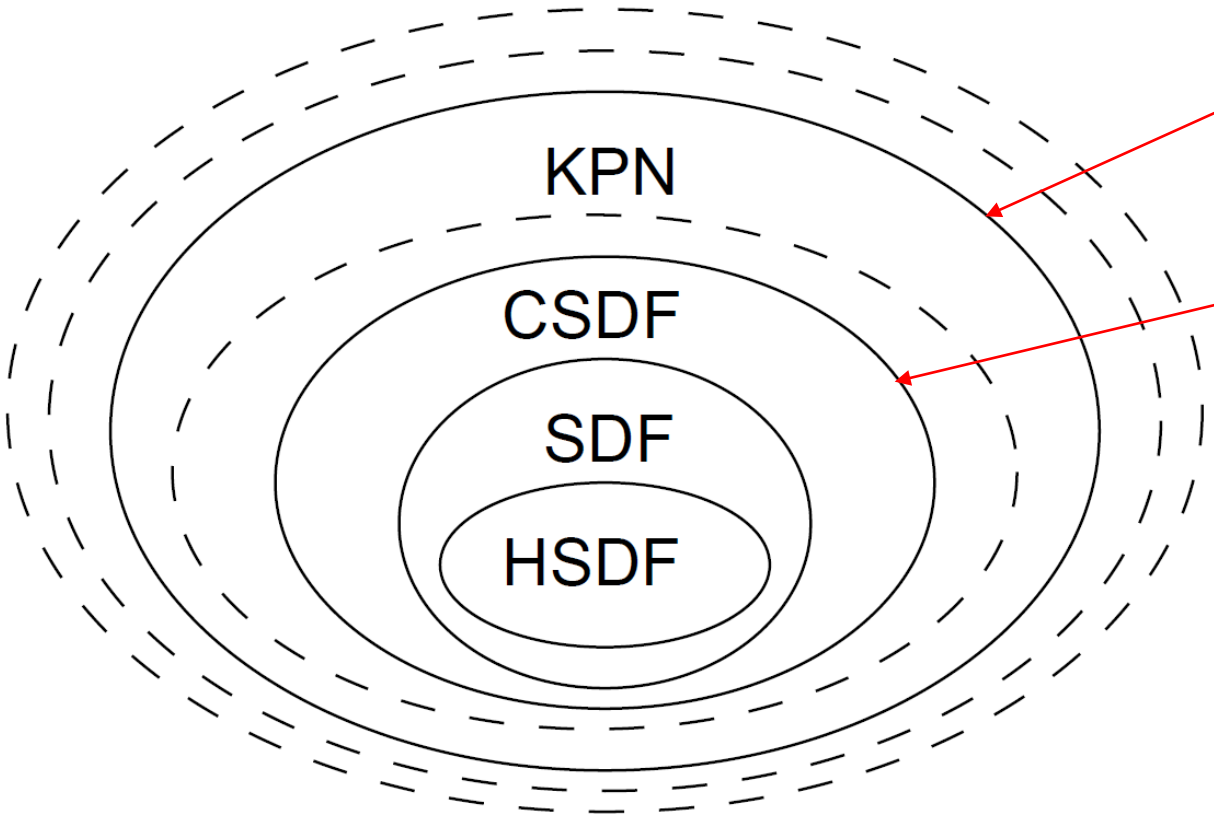
Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Comparison Criteria (1)

- Classification by Stuijk
- **Expressiveness** and succinctness indicate which systems can be modeled, and how compactly
- **Analyzability** relates to scheduling algorithms and the need for run-time support
- **Implementation** efficiency is influenced by the required scheduling policy and the code size

Expressiveness of Data Flow Models

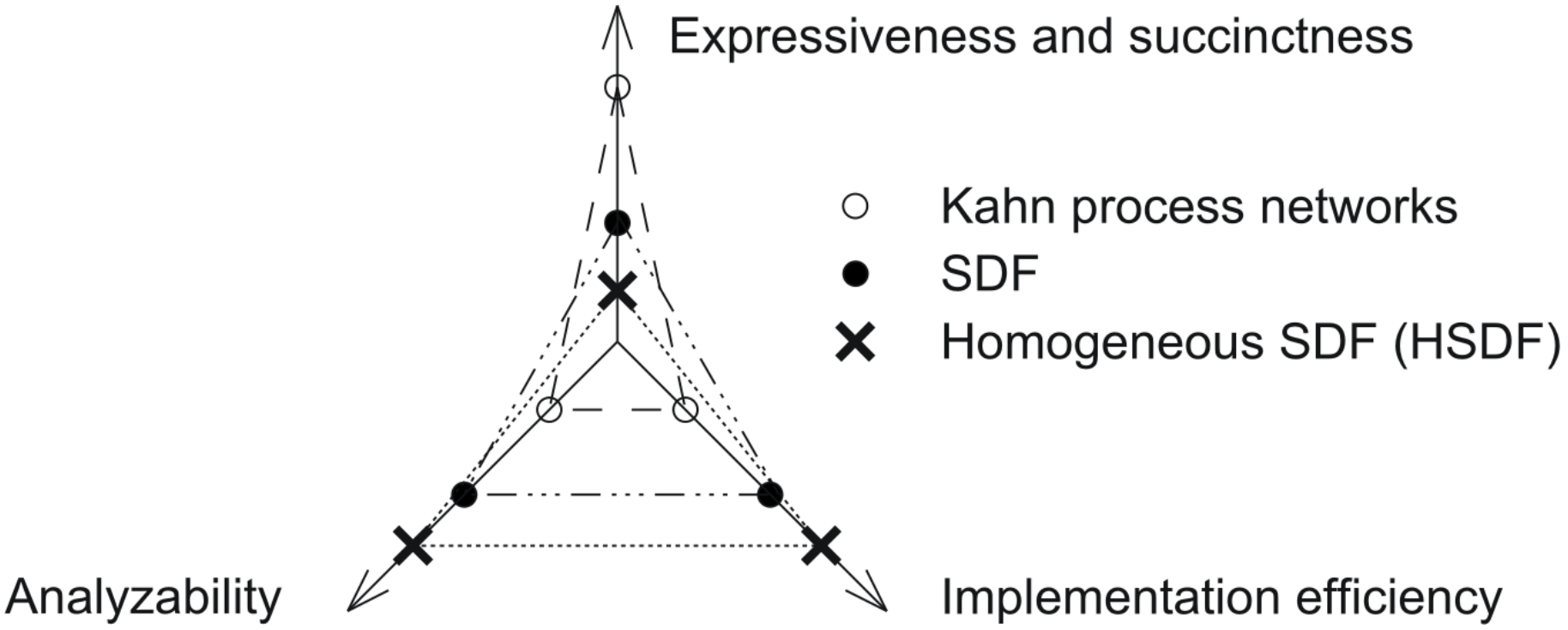


Turing complete

Not Turing complete

[S. Stuijk, 2007]

Classification by Stuijk



- KPN very expressive, but difficult to analyze

[S. Stuijk, 2007]

Comparison Criteria (2)

- Support for processes
- Number of processes
  - **Static** or **dynamic**
- Nesting
  - Can processes be nested?

```
process {
  process {
    process {
}}}
```

  - Or must they all declared at the same level?

```
process { … }
process { … }
process { … }
```

Comparison Criteria (3)

- Different techniques for **process creation**
  - Elaboration in the source (c.f. ADA)
    ```
    declare
       process P1 …
    ```
  - explicit fork and join (c.f. Unix)
    ```
    id = fork();
    ```
  - process creation calls
    ```
    id = create_process(P1);
    ```
- *E.g.*: StateCharts
  - A static number of processes
  - Nested process declaration
  - Process creation through elaboration in the source.

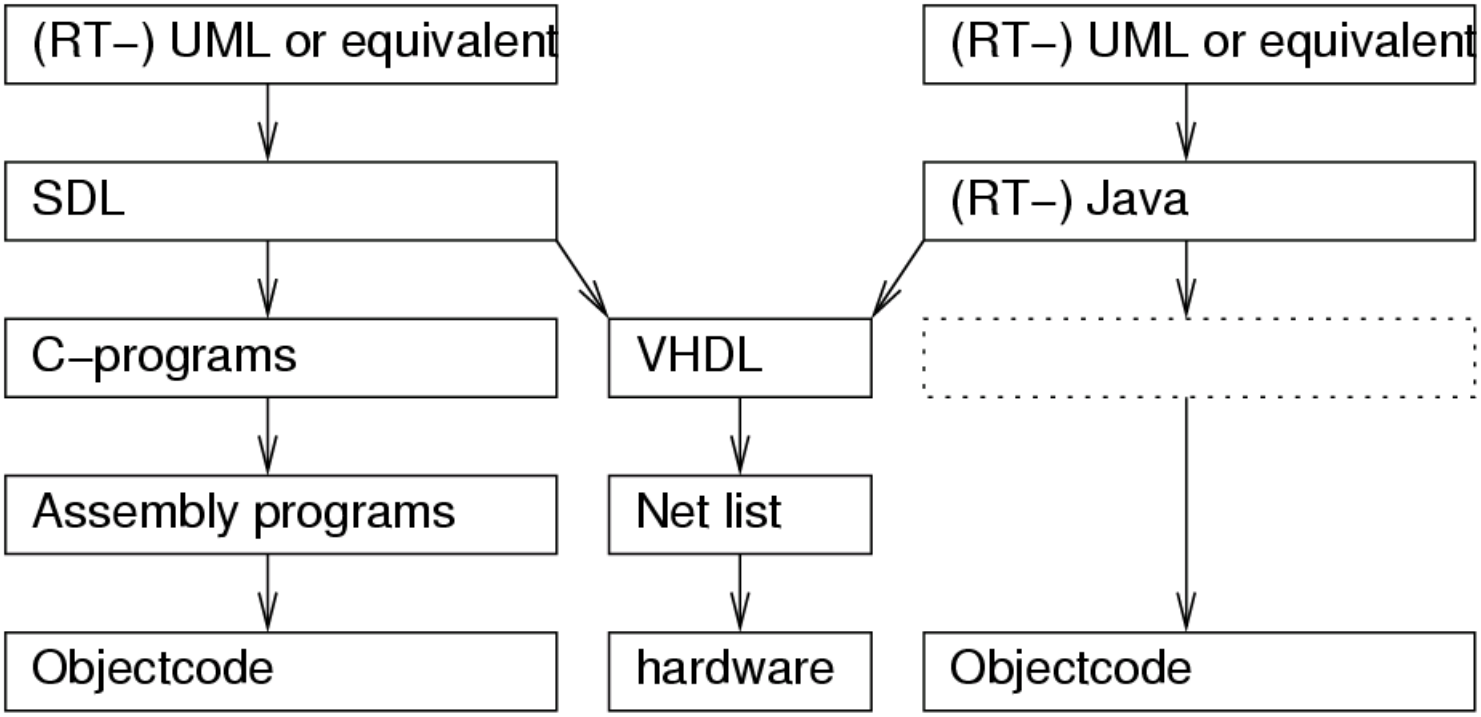
Language Comparison

Language	Behavioral Hierarchy	Structural Hierarchy	Programming Language Elements	Exceptions Supported	Dynamic Process Creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SDL	+-	+-	+-	-	+
Petri Nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	+	+
ADA	+	-	+	+	+

- SpecC and SystemC meet all listed requirements, but not others
  - E.g., precise specification of deadlines

So What Now?

- How to cope with MoC and language problems?
  - Mixed model approaches



Mixing models may require formal models of MoCs: *meta-modeling*!

Mixing Models of Computation: Ptolemy

- Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation
  - http://ptolemy.berkeley.edu/
  - Available examples are restricted to a subset of the supported models of computation

Mixing MoCs: Ptolemy

(Focus on executable models; "mature" models only)

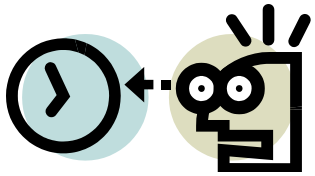
Communication/ local computations	**Shared memory**	**Message passing Synchronous | Asynchronous**
Communicating finite state machines	FSM, synchronous/reactive MoC	
Data flow	(*Not useful*)	Kahn networks, SDF, dynamic dataflow, discrete time
Petri nets		
Discrete event (DE) model	DE	Experimental distributed DE
Von Neumann model		CSP |
Wireless	Special model for wireless communication	
Continuous time	Partial differential equations	

Mixing Models of Computation: UML

(Focus on support of early design phases)

Communication/ local computations	**Shared memory**	**Message passing** Synchronous | Asynchronous	
Undefined components	*use cases* | *sequence charts, timing diagrams*		
Communicating finite state machines	State diagrams		
Data flow	(*Not useful*)	Data flow	
Petri nets		activity charts	
Discrete event (DE) model	-	-	
Von Neumann model	-	-	

UML for Embedded Systems?



- Initially not designed for real-time
- Initially lacking features
  - Partitioning of software into tasks and processes
  - specifying timing
  - specification of hardware components
- Projects to define profiles for E/RT systems
  - Schedulability, Performance and Timing Analysis
  - SysML (System Modeling Language)
  - UML Profile for SoC
  - Modeling and Analysis of Real-Time Embedded Systems
  - UML/SystemC, …
- Profiles may be incompatible

Example: Annotated Activity Diagrams

[http://www.omg.org/cgi-bin/doc?ptc/2002-03-02]

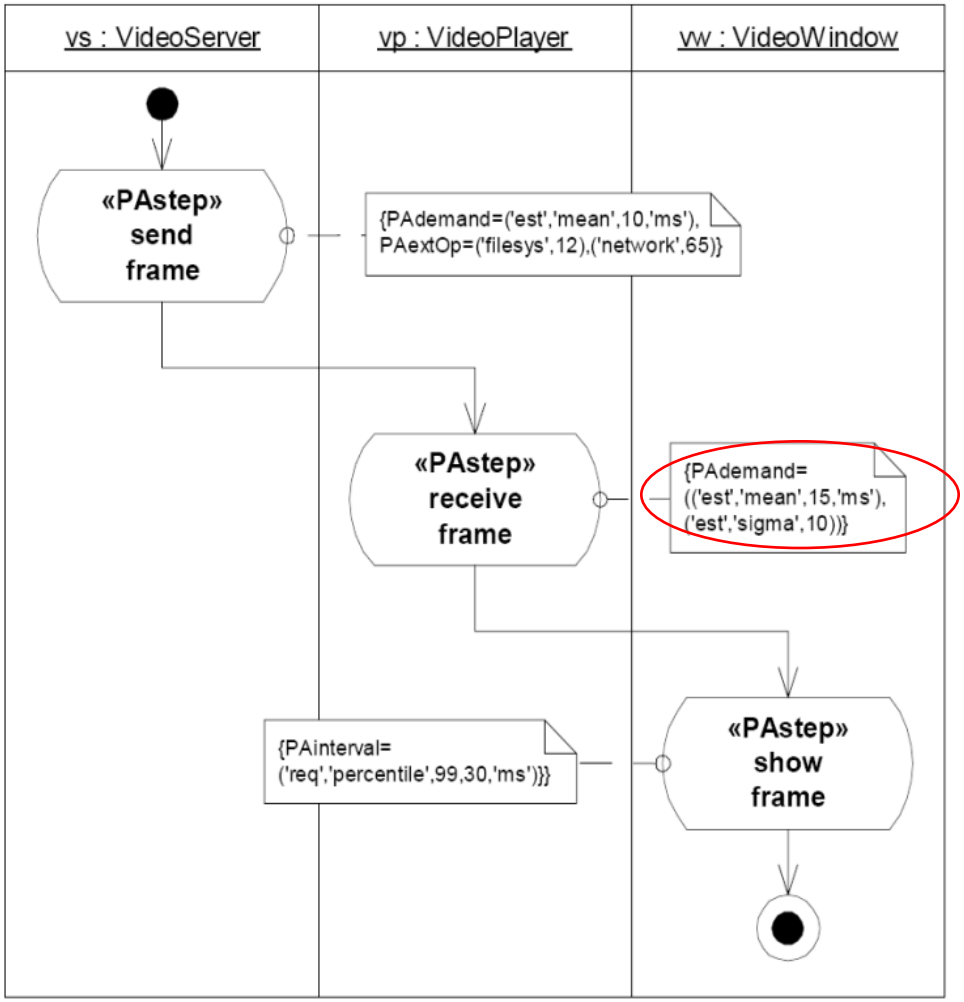


Figure 8-10 Details of the "send video" subactivity with performance annotations

See also W. Müller et al.: UML for SoC, http://jerry.c-lab.de/uml-soc/

Levels of Modeling Abstraction

- Levels, at which modeling can be done:
  - System level
  - Algorithmic level: just the algorithm
  - Processor/memory/switch (PMS) level
  - Instruction set architecture (ISA) level: function only
  - Transaction level modeling (TML)
    - Reads and writes are just "transactions" (not cycle accurate)
  - Register-transfer level: registers, muxes, adders, … (cycle accurate, bit accurate)
  - Gate-level: gates
  - Layout level: polygons

Tradeoff between accuracy and simulation speed

System Level Models

- "System level" not clearly defined
- Here: the entire cyber-physical/embedded system
  - System into which information processing is embedded
  - Possibly also the environment
- Models may include
  - Information processing
  - Physical mechanics
- May be difficult to find appropriate simulators
  - Solutions: VHDL-AMS, SystemC or MATLAB
  - MATLAB+VHDL-AMS support partial differential equations
- Automatic synthesis is challenging from this level

Algorithmic Level

- Simulating the algorithms implemented by the cy-phy or ES
- No reference to processors or instruction sets
- Data types may support higher precision than the final design
- Model is *bit-true* if
  - Every bit in the model corresponds to exactly one bit in the final design
- Single process or sets of cooperating processes

Segment of MPEG-4

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3||y3<0||48<y3)
        then_block_1;  else else_block_1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2;  else else_block_2;
}}}}}}
```

Instruction Set Architecture (ISA)

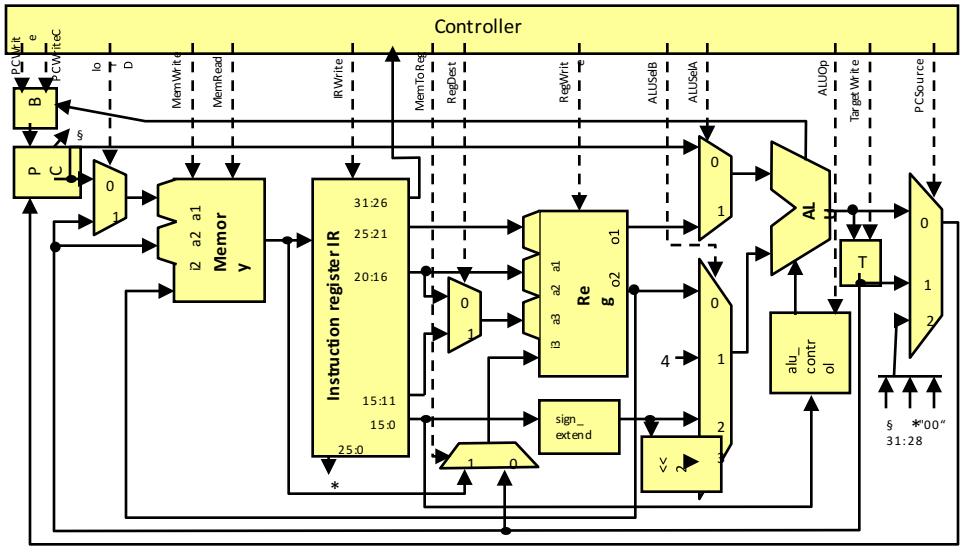
- Algorithms already compiled for the target ISA
- Model allows counting the executed # of instructions

Assembler (MIPS)	Simulated semantics
`and $1,$2,$3`	`Reg[1]:=Reg[2] ∧ Reg[3]`
`or $1,$2,$3`	`Reg[1]:=Reg[2] ∨ Reg[3]`
`andi $1,$2,100`	`Reg[1]:=Reg[2] ∧ 100`

- Variations:
  - Simulation only of the effect of instructions (not timing)
  - Transaction-level modeling: each read/write is one transaction, instead of a set of signal assignments
  - Cycle-true simulations: exact number of cycles
  - Bit-true simulations: simulations using exactly the correct number of bits

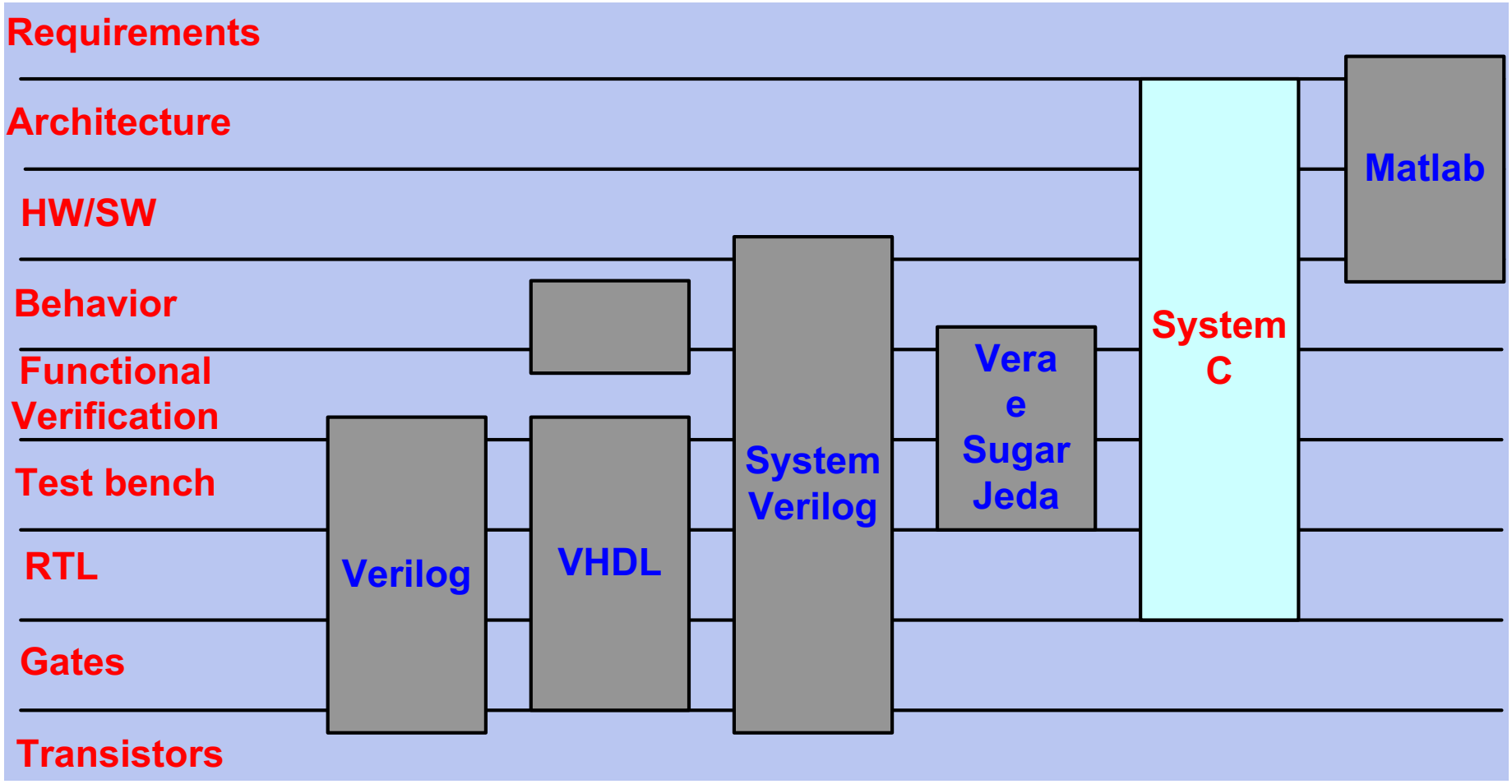
Register Transfer Level (RTL)

- Modeling of all components at the register-transfer level, including
  - arithmetic/logic units (ALUs),
  - registers,
  - memories,
  - muxes and
  - decoders.



- Models at this level are always cycle-true
- Automatic synthesis is frequently feasible

Languages and Levels Covered



What's the Bottom Line?

- Good embedded software requires proper modeling
  - However, there is no ideal modeling technique!
  - The choice of the technique depends on the application
- In the field, you should
  - Check code generation from non-imperative models
  - Note the tradeoff between expressiveness and analyzability
- **It may be necessary to combine modeling techniques**
  - Think about the model before you write down your spec!
  - Be aware of pitfalls
- You may be forced to use imperative models
  - However, you can still implement FSMs or KPNs in Java

Summary

- Imperative Von-Neumann models
  - Problems resulting from access to shared resources and mutual exclusion (*e.g.* deadlock)
  - Communication: built-in or facilitated by libraries
- Comparison of models
  - Expressiveness vs. analyzability
  - Process creation
  - Mixing models of computation (Ptolemy & UML)
- Modeling levels

Want more MoC? Look into COMP-522: Modeling and Simulation

Next Time

- Embedded System Hardware
  - Sensors
  - Chapter 3.1-2