technische universität
dortmund

fakultät für informatik
informatik 12

Imperative model of computation

Peter Marwedel
Informatik 12
TU Dortmund
Germany

2012年 11 月 07 日

Models of computation considered in this course

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	Scoreboarding + Tomasulo Algorithm (☞ Comp.Archict.)		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	
Imperative (Von Neumann) model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on semantic model

Imperative (von-Neumann) model

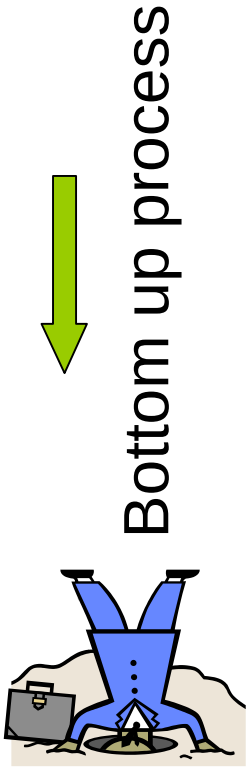
The von-Neumann model reflects the principles of operation of standard computers:

- Sequential execution of instructions (total order of instructions)

- Possible branches
- Visibility of memory locations and addresses

Example languages

- Machine languages (binary)
- Assembly languages (mnemonics)
- Imperative languages providing limited abstraction of machine languages (C, C++, Java, ….)



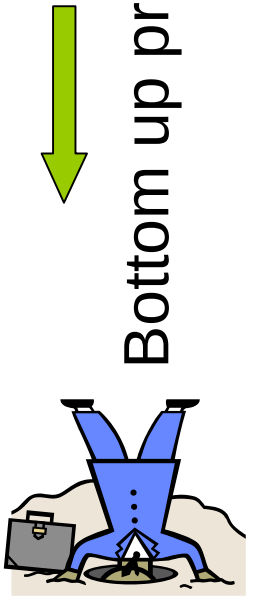
Threads/processes

Threads/processes

- Initially available only as entities managed by OS
- In most cases:
  - Context switching between threads/processes, frequently based on pre-emption
    (cooperative multi-tasking or time-triggered system rare)
- Made available to programmer as well

☞ Partitioning of applications into threads (same address space)

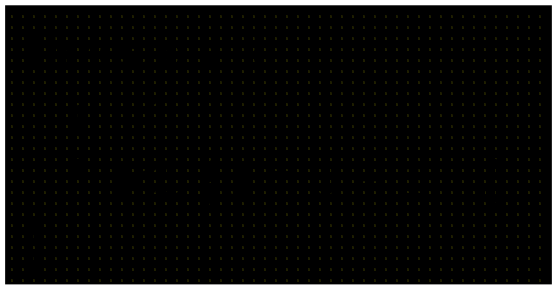
- Languages initially not designed for communication, but synchronization and communication is needed!
  - ☞ Access to shared memory



Communication via shared memory

Several threads access the same memory

- Very fast communication technique (no extra copying)
- Potential race conditions:



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

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

☞inconsistent results possible

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

Shared memory

```
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

Deadlocks

Deadlocks can happen, if the following 4 conditions are met [Coffman, 1971]:

- **Mutual exclusion**: a resource that cannot be used by >1 thread at a time
- **Hold and wait**: thread already holding resources may request new resources
- **No preemption**: Resource cannot be forcibly removed from threads, they can be released only by the holding threads
- **Circular wait**: ≥ 2 threads form a circular chain where each thread waits for a resource that the next thread in the chain holds

Techniques for turning one of these conditions false degrade performance/ increase resource requirements seriously

In non-safety-critical software, it is “ok” to ensure that deadlocks are “sufficiently” infrequent.

Problems with imperative languages and shared memory

- Specification of total order is an over-specification.
  A partial order would be sufficient.
  The total order reduces the potential for optimizations
- Preemptions at any time complicate timing analysis
- Access to shared memory leads to anomalies, that have to be pruned away by mutexes, semaphores, monitors
- Potential deadlocks

- Access to shared, protected resources leads to priority inversion (☞ chapter 4)
- Termination in general undecidable
- Timing cannot be specified and not guaranteed

Synchronous message passing: CSP

- CSP (communicating sequential processes)
  [Hoare, 1985],
  Rendez-vous-based communication:
  Example:

```
process A                    process B
..                           ..
var a ...                    var b ...
  a:=3;                        ...
  c!a; -- output               c?b; -- input
end                          end
```

No race conditions (!)

Synchronous message passing: 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;
```

Synchronous message passing: Ada

After Ada Lovelace (said to be the 1st female programmer).

US Department of Defense (DoD) wanted to avoid multitude of programming languages

☞Definition of requirements

☞Selection of a language from a set of competing designs (selected design based on PASCAL)

Ada'95 is object-oriented extension of original Ada.

Salient: task concept

Synchronous message passing: Using of 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;
```

Communication/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

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

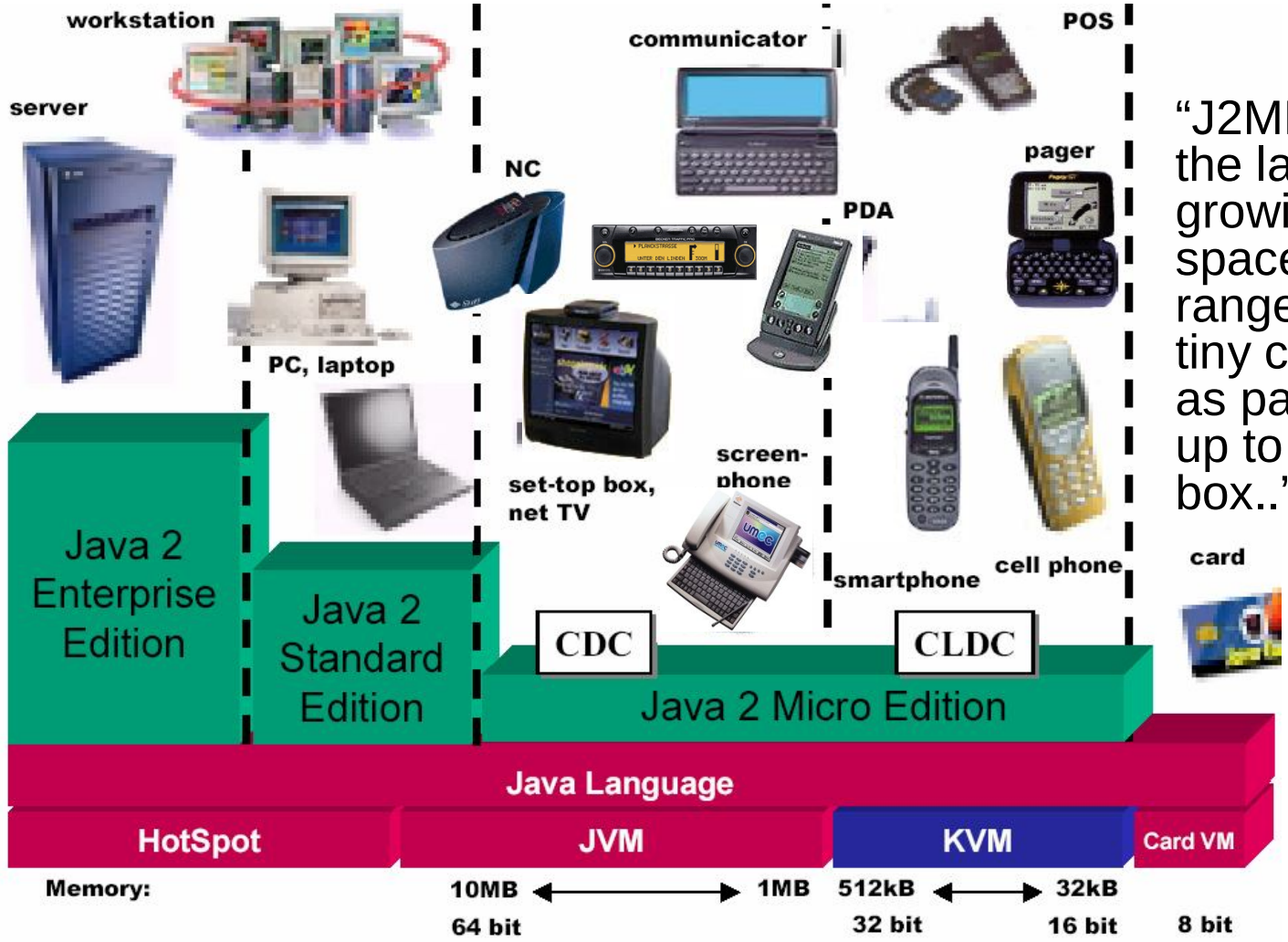
Potential benefits:

- Clean and safe language
- Supports multi-threading (no OS required?)
- Platform independence (relevant for telecommunications)

Problems:

- Size of Java run-time libraries? Memory requirements.
- Access to special hardware features
- Garbage collection time
- Non-deterministic dispatcher
- Performance problems
- Checking of real-time constraints

Overview over 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

Lee's conclusion

Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

….

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes).

Improve threads?

Or replace them?

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

technische universität
dortmund

fakultät für informatik
informatik 12

Comparison of models

Peter Marwedel
Informatik 12,
TU Dortmund,
Germany

2012年 11 月 07 日

Models of computation considered in this course

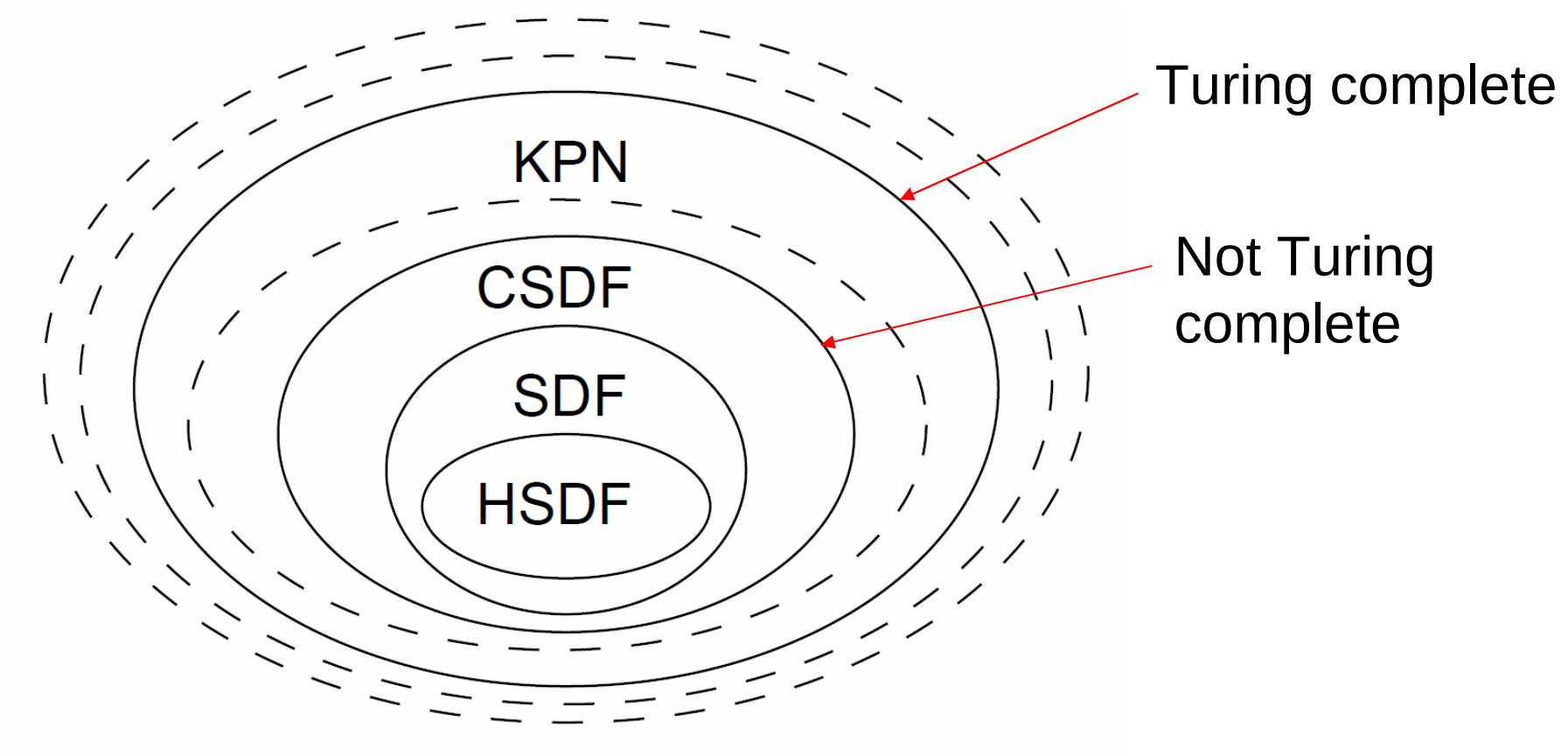
Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorithm (☞ Comp.Archict.)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative (Von Neumann) model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on semantic model

Classification by Stuijk

- **Expressiveness** and **succinctness** indicate, which systems can be modeled and how compact the are.
- **Analyzability** relates to the availability of 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



[S. Stuijk, 2007]

Properties of processes/threads (1)

- **Number of processes/threads**
  static;
  dynamic (dynamically changed hardware architecture?)

- **Nesting:**
  - Nested declaration of processes
    process {
      process {
        process {
    }}}
  - or all declared at the same level
    process { … }
    process { … }
    process { … }

Properties of processes/threads (2)

- 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);
    ```

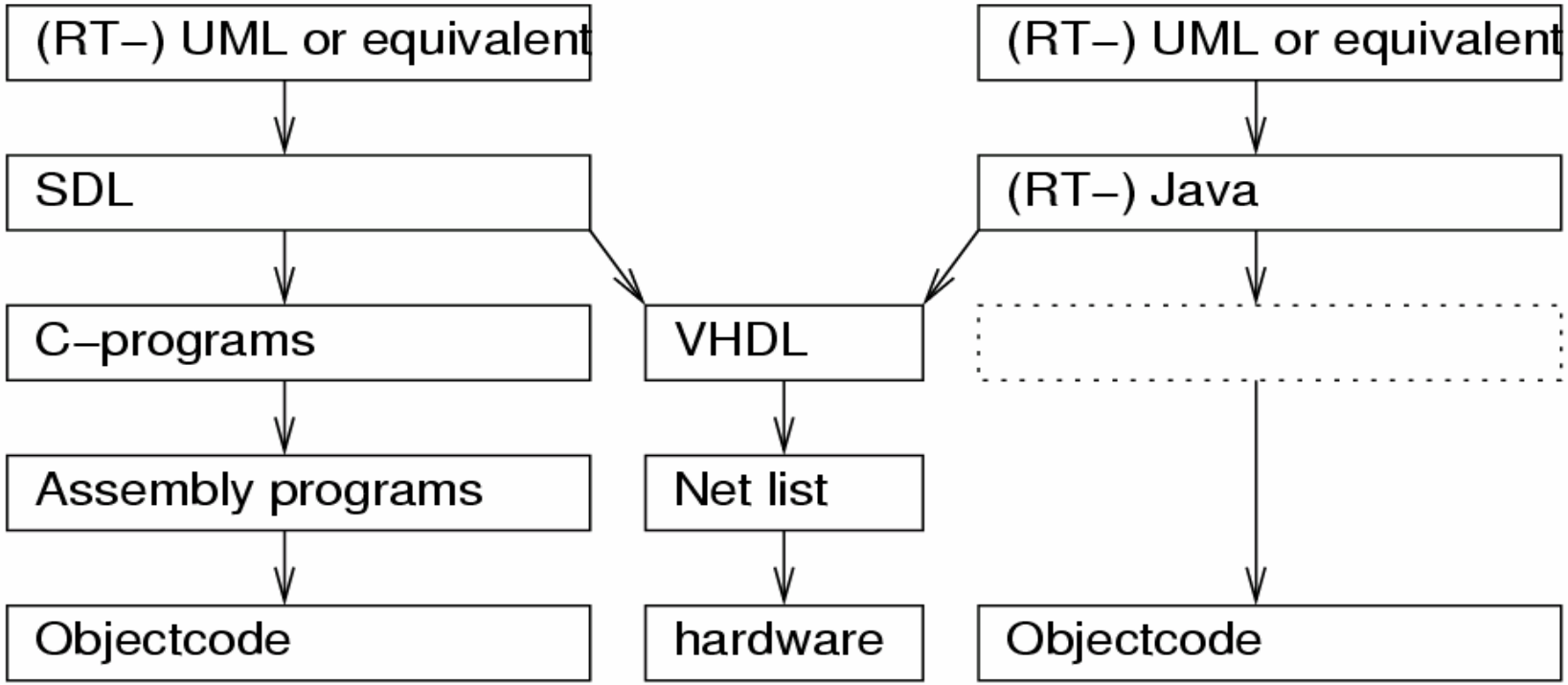
E.g.: StateCharts comprises a static number of processes, nested declaration of processes, and process creation through elaboration in the source.

Language comparison

Language	Behavioral Hierarchy	Structural Hierarchy	Programming Language Elements	Exceptions Supported	Dynamic Process Creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SpecCharts	+	-	+	+	-
SDL	+-	+-	+-	-	+
Petri nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	- (2.0)	- (2.0)
ADA	+	-	+	+	+

How to cope with MoC and language problems in practice?

Mixed approaches:



Transformations between models

- Transformations between models are possible, e.g.
  - Frequent transformation into sequential code
  - Transformations between restricted Petri nets and SDF
  - Transformations between VHDL and C
- Nevertheless, it is best to specify in the most convenient model
- Transformations should be based on the precise description of the semantics
  (e.g. Chen, Sztipanovits et al., DATE, 2007)
  (☞ an advanced course would be nice)

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.

Newton‘s cradle

Mixing MoCs: Ptolemy

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

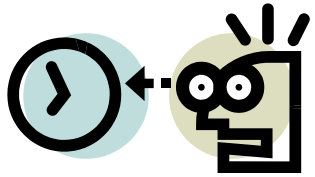
Communication/ local computations	**Shared memory**	**Message passing Synchronous | Asynchronous**
Undefined components		
Communicating finite state machines	FSM, synchronous/reactive MoC	
Data flow		Kahn networks, SDF, dynamic dataflow, discrete time
Petri nets		
Discrete event (DE) model	DE	Experimental 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:

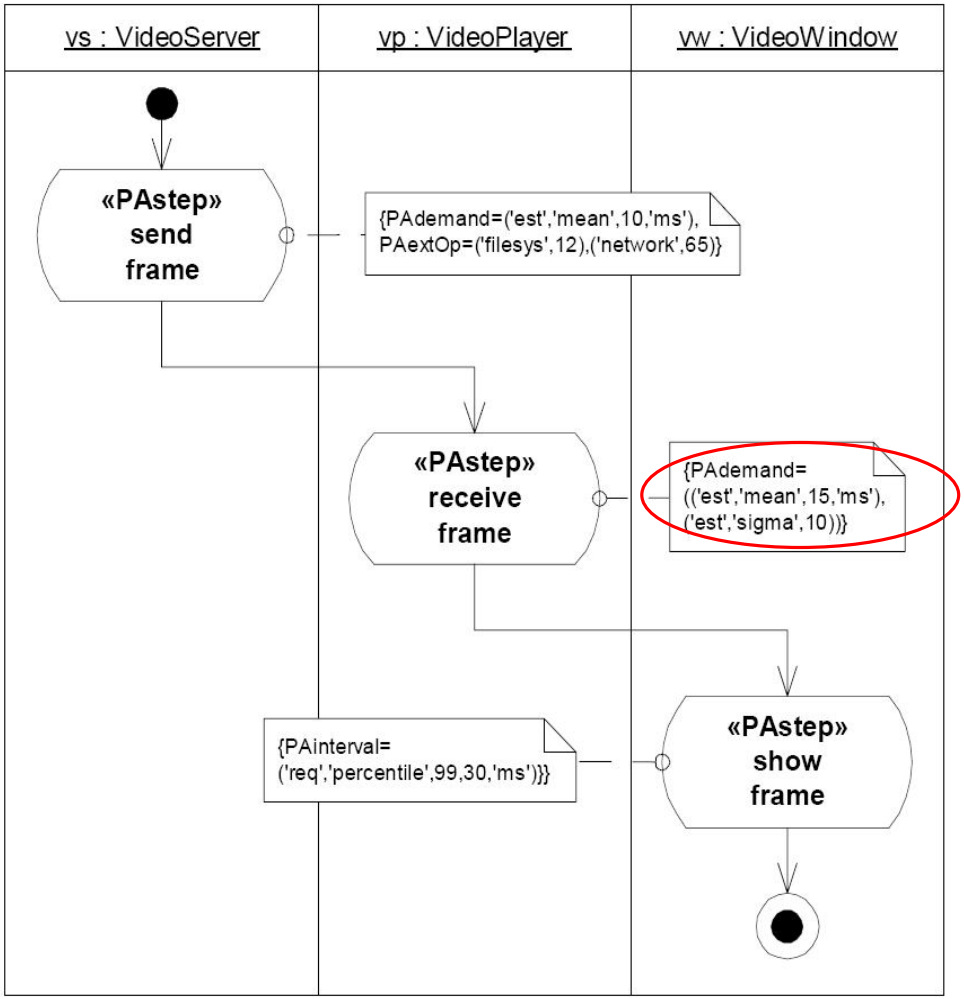
- Partitioning of software into tasks and processes
- specifying timing
- specification of hardware components

Projects on defining profiles for embedded/real-time 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: Activity diagram with annotations



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

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

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

Modeling levels

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): memory reads & writes are just “transactions“ (not cycle accurate)
- Register-transfer level: registers, muxes, adders, .. (cycle accurate, bit accurate)
- Gate-level: gates
- Layout level

Tradeoff between accuracy and simulation speed

Example: System level

- Term not clearly defined.
- Here: denotes the entire cyber-physical/embedded system, system into which information processing is embedded, and possibly also the environment.
- Models may include mechanics + information processing.
  May be difficult to find appropriate simulators.
  Solutions: VHDL-AMS, SystemC or MATLAB.
  MATLAB+VHDL-AMS support partial differential equations.
- Challenge to model information processing parts of the system such that the simulation model can be used for the synthesis of the embedded system.

Example: Algorithmic level

- Simulating the algorithms envisioned for the the cyber-physical/embedded system.
- No reference to processors or instruction sets.
- Data types may still allow a higher precision than the final implementation.
- Model is **bit-true** if data types selected such that every bit corresponds to exactly one bit in the final implementation
- 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 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`

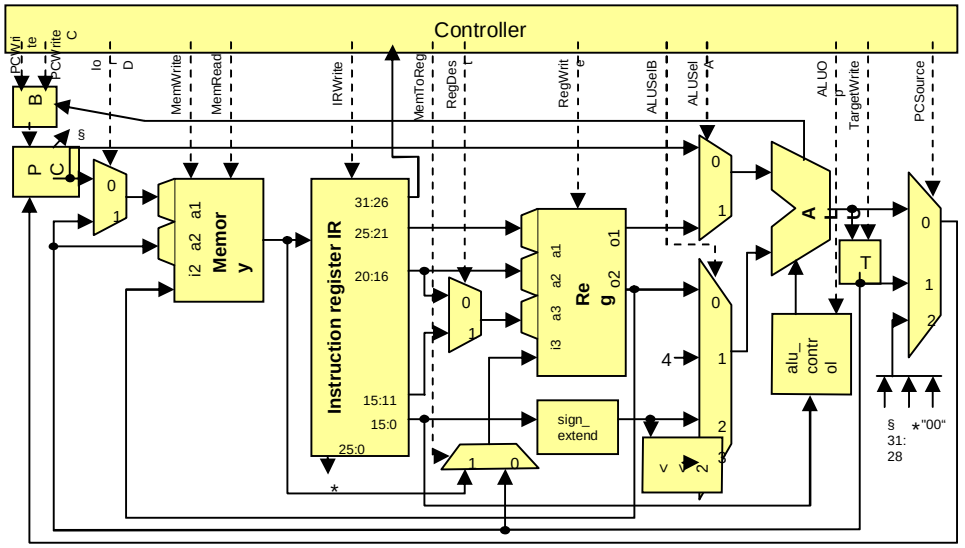
Variations:

- Simulating only of the effect of instructions
- **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

Example: Register transfer level (RTL)

Modeling of all components at the register-transfer level, including

- arithmetic/logic units (ALUs),
- registers,
- memories,
- muxes and
- decoders.

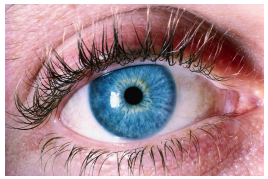


Models at this level are always cycle-true.

Automatic synthesis from such models is frequently feasible.

What‘s the bottom line?

- The prevailing technique for writing embedded SW has inherent problems; some of the difficulties of writing embedded SW are not resulting from design constraints, but from the modeling.
- However, there is no ideal modeling technique.
- The choice of the technique depends on the application.
- Check code generation from non-imperative models
- There is a tradeoff between the power of a modeling technique and its analyzability.
- It may be necessary to combine modeling techniques.
- **In any case, open your eyes & think about the model before you write down your spec! Be aware of pitfalls.**



- You may be forced, to use imperative models, but you can still implement, for example, finite state machines or KPNs in Java.

Summary

- Imperative Von-Neumann models
  - Problems resulting from access to shared resources and mutual exclusion (e.g. potential deadlock)
  - Communication built-in or by libraries
- Comparison of models
  - Expressiveness vs. analyzability
  - Process creation
  - Mixing models of computation
    - Ptolemy & UML
    - Using FSM and KPN models in imperative languages, etc.