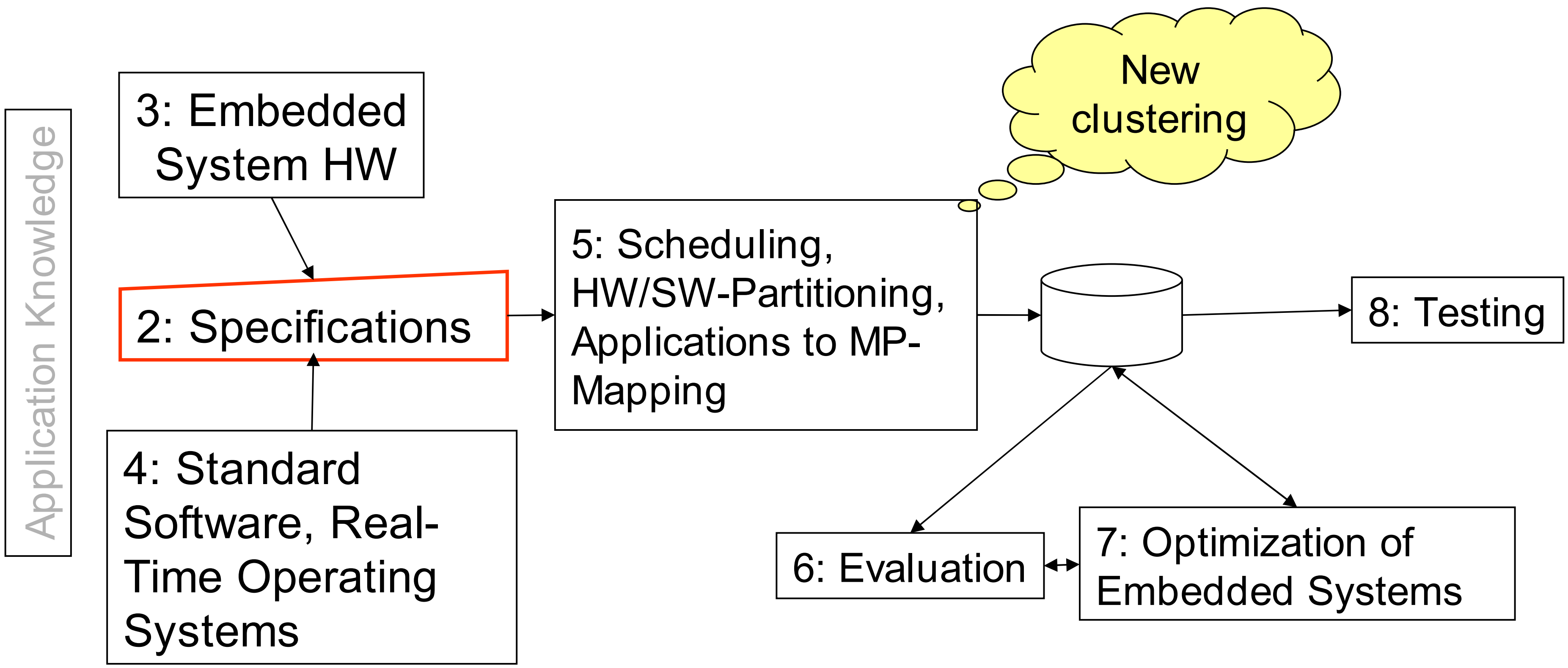
Specifications

Peter Marwedel
TU Dortmund,
Informatik 12

2008/11/15

Structure of this course



Motivation for considering specs

- Why considering specs?
- If something is wrong with the specs, then it will be difficult to get the design right, potentially wasting a lot of time.
- Why not just use standard languages like Java, C++ etc?

☞ Example demonstrating weakness

Consider a Simple Example

"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."

Eric Gamman 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.

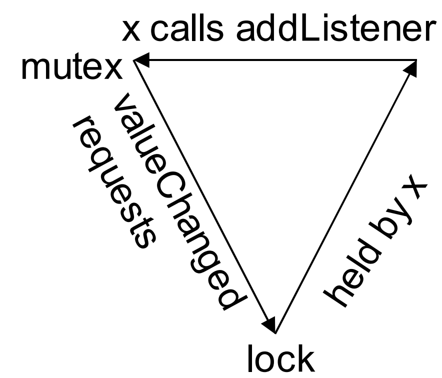
Example: Observer Pattern with Mutual Exclusion (mutexes)

```
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!

Simple Observer Pattern Becomes not so simple

```
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?

Simple Observer Pattern: 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, but listeners may be notified in the opposite order. The listeners may be alerted to the value-changes in the wrong order!

What it Feels Like to Use the *synchronized* Keyword in Java

Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, *Scientific American*, September 1999.

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).

A stake in the ground …

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

Problems with thread-based concurrency

*"… **threads as a concurrency model are a poor match for embedded systems**. … they work well only … where best-effort scheduling policies are sufficient."*

Ed Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

Problems with classical CS theory and von Neumann computing (1)

"The lack of timing in the core abstraction is a flaw, from the perspective of embedded software, …"

Ed Lee: Absolutely Positively on Time,
IEEE Computer, July, 2005

"Timing is everything"

Frank Vahid, WESE 2008

Problems with classical CS theory and von Neumann computing (2)

Even the core … notion of "computable" is at odds with the requirements of embedded software.

In this notion, useful computation terminates, but termination is undecidable.

In embedded software, termination is failure, and yet to get predictable timing, subcomputations must decidably terminate.

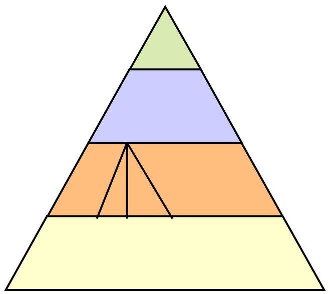
What is needed is nearly a reinvention of computer science.

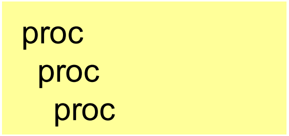
Ed Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

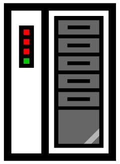
☞ Search for non-thread-based, non-von-Neumann MoCs; which are the requirements for specification techniques?

Specification of embedded systems: Requirements for specification techniques (1)

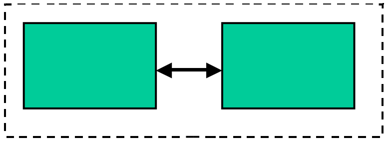
- **Hierarchy**
  Humans not capable to understand systems containing more than ~5 objects.
  Most actual systems require more objects
  ☞ Hierarchy
  - Behavioral hierarchy
    Examples: states, processes, procedures.
  - Structural hierarchy
    Examples: processors, racks, printed circuit boards







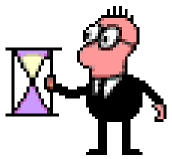
- **Compositional behavior**
  Must be "easy" to derive behavior from behavior of subsystems



- **Concurrency, Synchronization and communication**

Requirements for specification techniques (2)
Timing

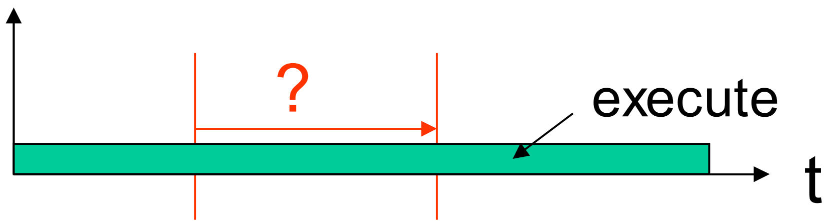
- **Timing behavior**
  Essential for connecting to physical environment



  - **Additional information (periods, dependences, scenarios, use cases) welcome**
  - **Also, the speed of the underlying platform must be known**
  - **Far-reaching consequences for design processes!**

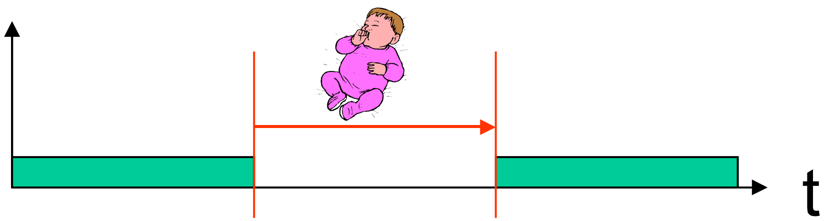
4 types of timing specs required, according to Burns 1990]:

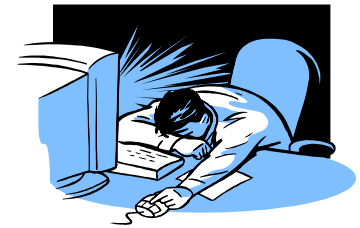
1. Measure elapsed time
   Check, how much time has elapsed since last call



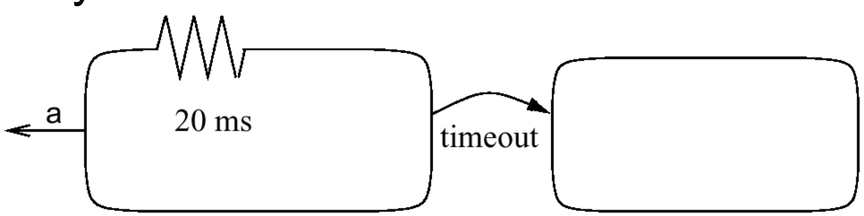
Requirements for specification techniques (3) Timing (2)

1. Means for delaying processes

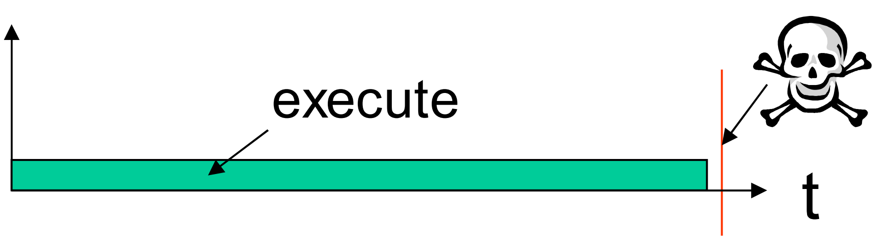


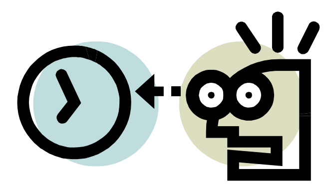


1. Possibility to specify timeouts
   Stay in a certain state a maximum time.



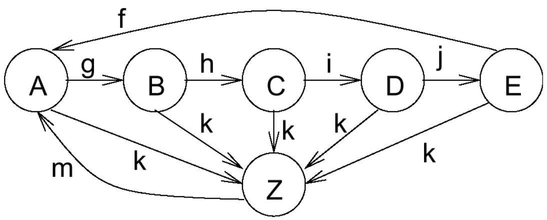
1. Methods for specifying deadlines
   Not available or in separate control file.





Specification of embedded systems (4): Support for designing reactive systems

- **State-oriented behavior**
  Required for reactive systems; classical automata insufficient.
- **Event-handling**
  (external or internal events)
- **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state



We will see, how all the arrows labeled k can be replaced by a single one.

Requirements for specification techniques (5)

- **Presence of programming elements**
- **Executability** (no algebraic specification)
- **Support for the design of large systems** (☞ OO)
- **Domain-specific support**
- **Readability**

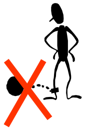
- **Portability and flexibility**

- **Termination**

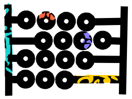
- **Support for non-standard I/O devices**

- **Non-functional properties**

- **Support for the design of dependable systems**
  Unambiguous semantics, ...
- **No obstacles for efficient implementation**



- **Adequate model of computation**

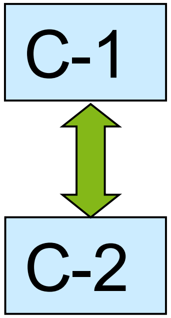


Models of computation
- Definition -

What does it mean, "to compute"?

Models of computation define:

- Components and an execution model for computations for each component
- Communication model for exchange of information between components.
  - Shared memory
  - Message passing
  - …



Communication

- **Shared memory**

Variables accessible to several tasks.

Model is useful only for local systems.

Shared memory

Potential race conditions (☞inconsistent results possible)
☞ Critical sections = sections at which exclusive access to resource *r* (e.g. shared memory) must be guaranteed.

```
process a {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

```
process b {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

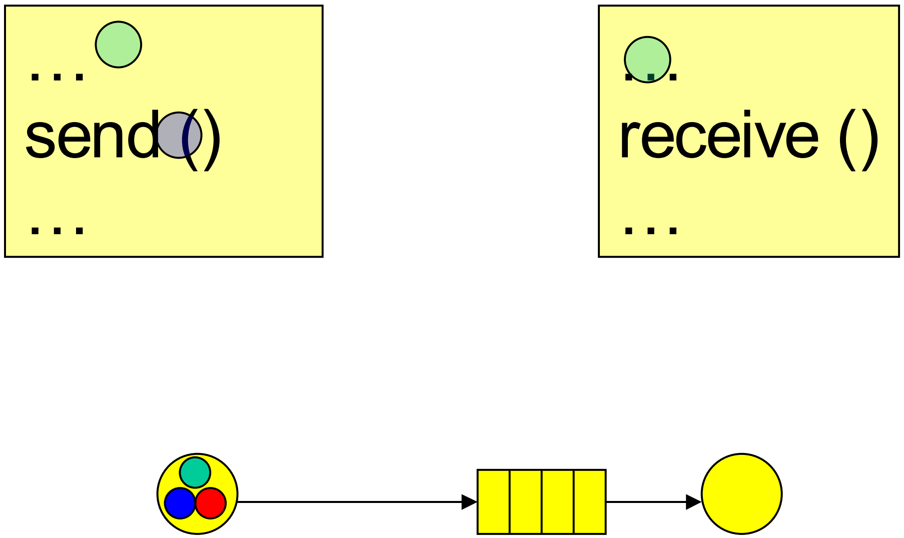
Race-free access to shared memory protected by S possible

This model may be supported by:

- mutual exclusion for critical sections
- cache coherency protocols

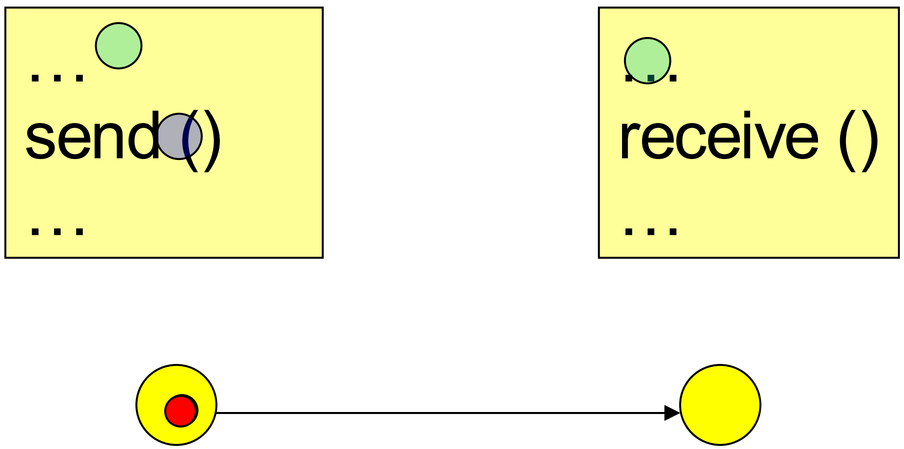
Non-blocking/asynchronous message passing

Sender does not have to wait until message has arrived;
potential problem: buffer overflow



Blocking/synchronous message passing *rendez-vous*

Sender will wait until receiver has received message



Extended *rendez-vous*

Explicit acknowledge from receiver required. Receiver can do checking before sending acknowledgement.

… send () …	… receive () … ack …

Components (1)

- Von Neumann model

  Sequential execution, program memory etc.

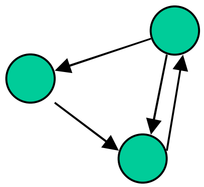
- Discrete event model

queue

a 6	←	5	10	13	15	19	time
b 7							
c 8		a:=5	b:=7	c:=8	a:=6	a:=9	action

Components (2)

- Finite state machines



- Differential equations

$$\frac{\partial^2 x}{\partial t^2} = b$$

Combined models

- languages presented later in this chapter -

- SDL
  FSM+asynchronous message passing
- StateCharts
  FSM+shared memory
- CSP, ADA
  von Neumann execution+synchronous message passing
- ….

See also

- Work by Ed Lee, UCB
- Axel Jantsch: Modeling Embedded Systems and Soc's: Concurrency and Time in Models of Computation, Morgan-Kaufman, 2004

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Communicating finite state machines	StateCharts		SDL
Data flow model ⊂ Computational graphs	Not useful	Simulink Sequence dia-gram, Petri nets	Kahn process networks, SDF
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	
Discrete event (DE) model	VHDL, …	Only experimental systems, e.g. distributed DE in Ptolemy	

Ptolemy

Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation.

http://ptolemy.berkeley.edu/

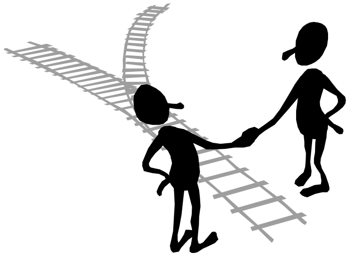
Available examples are restricted to a subset of the supported models of computation.

Newton's craddle

Facing reality

No language that meets all language requirements

☞ using compromises



Summary

- Non-deterministic thread-based concurrency results in problems

☞ Search for other models of computation =
  - models of components
    - finite state machines (FSMs)
    - data flow, ….
  - + models for communication
    - Shared memory
    - Message passing