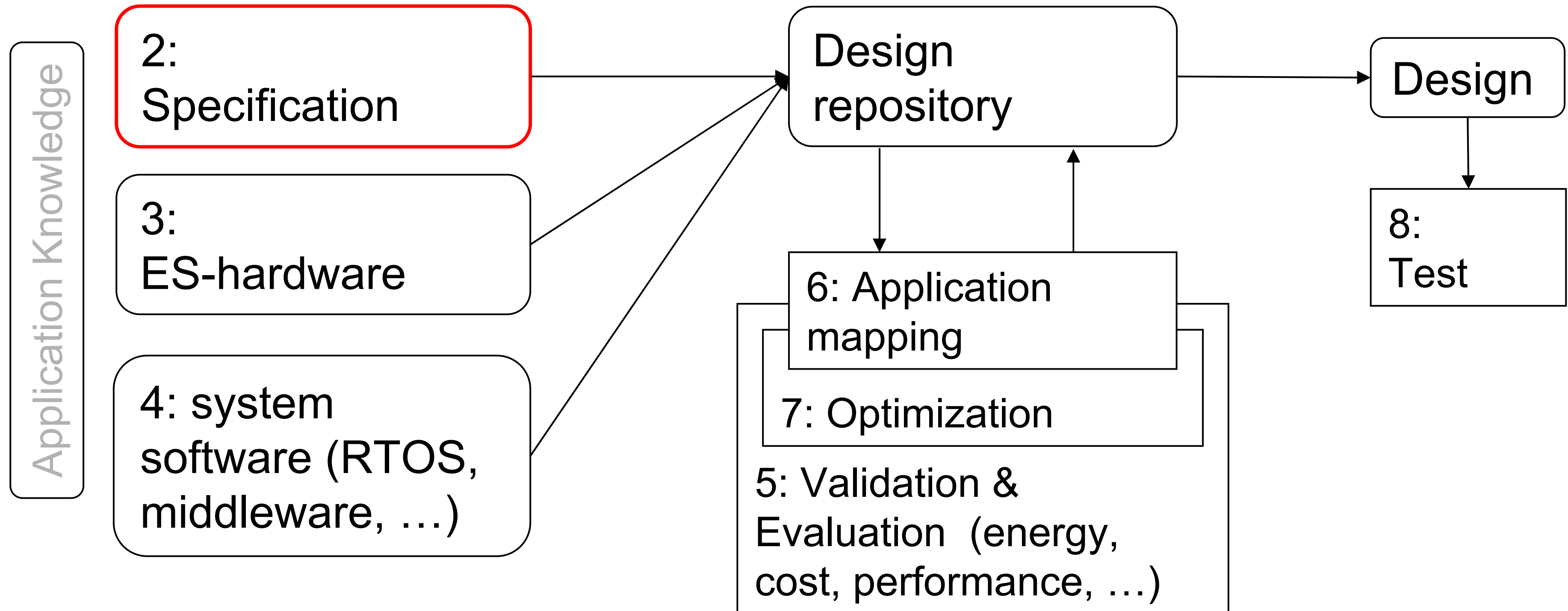
Specifications and Modeling

Peter Marwedel
TU Dortmund,
Informatik 12

2010/10/09

Graphics: © Alexandra Nolte, Gesine Marwedel, 2003

Structure of this course



Numbers denote sequence of chapters

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.
- Typically, we work with **models** of the **system under design** (SUD)

☞ What is a *model* anyway?

Models

Definition: *A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task*.

[Jantsch, 2004]:

Which requirements do we have for our models?

Requirements for specification techniques: Hierarchy

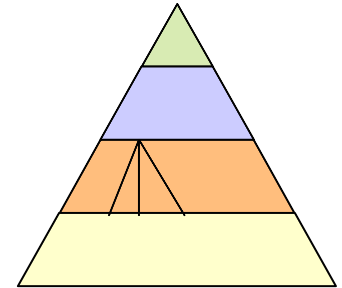
Hierarchy

Humans not capable to understand systems containing more than ~5 objects.

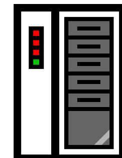
Most actual systems require more objects

☞ Hierarchy

- Behavioral hierarchy
  Examples: states, processes, procedures.
- Structural hierarchy
  Examples: processors, racks,
  printed circuit boards



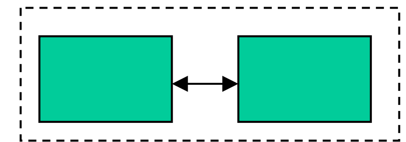
proc
proc
proc



Requirements for specification techniques (2): Component-based design

- Systems must be designed from components
- Must be “easy” to derive behavior from behavior of subsystems

☞Work of Sifakis, Thiele, Ernst, …

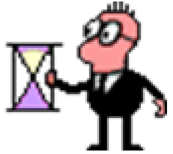


Requirements for specification techniques (3): Concurrency, Synchronization and communication

- Concurrency
- Synchronization
  and communication

© p. marwedel, informatik 12, 2009

Requirements for specification techniques (4): Timing



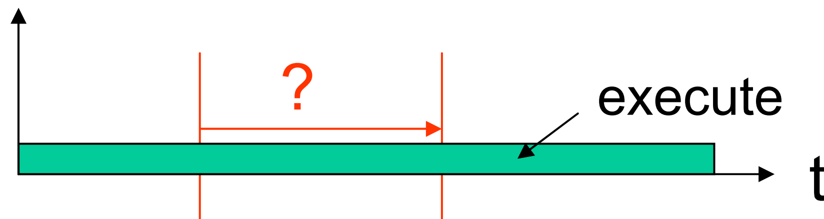
- **Timing behavior**
  Essential for embedded and cy-phy systems!
  - **Additional information (periods, dependences, scenarios, use cases) welcome**
  - **Also, the speed of the underlying platform must be known**
  - **Far-reaching consequences for design processes!**

The lack of timing in the core abstraction (of computer science) *is a flaw, from the perspective of embedded software* [Lee, 2005]

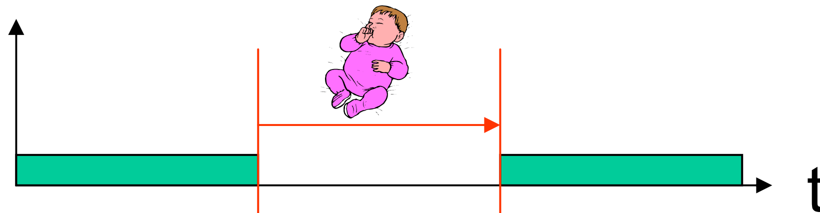
Requirements for specification techniques (4): Timing (2)

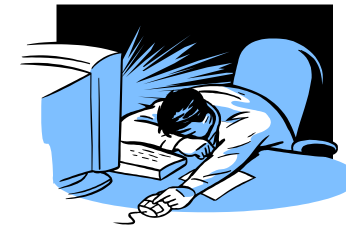
4 types of timing specs required, according to Burns, 1990:

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



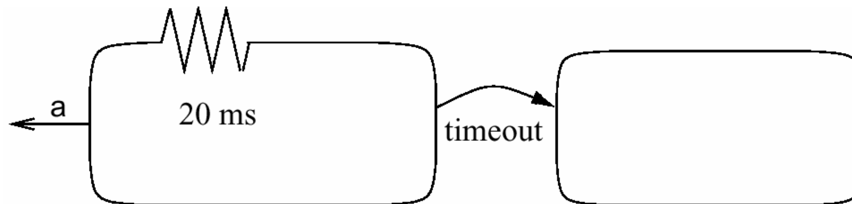
2. Means for delaying processes

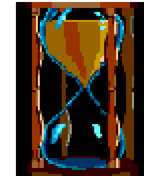




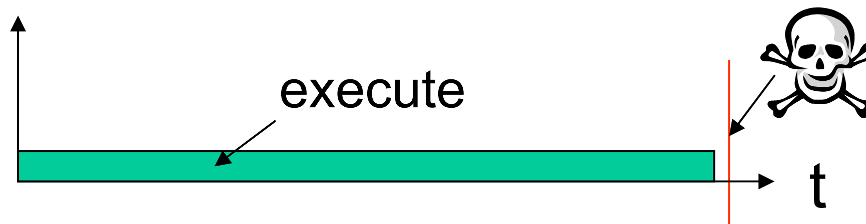
Requirements for specification techniques (4) Timing (3)

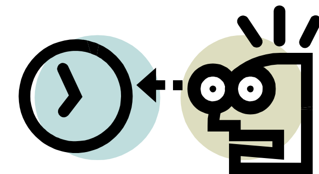
3. Possibility to specify timeouts
   Stay in a certain state a maximum time.





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

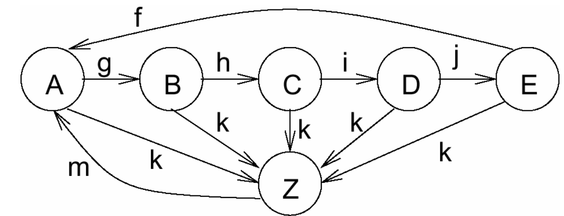




© p. marwedel, informatik 12, 2009

Specification of embedded systems (5): Support for designing reactive systems

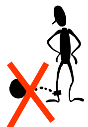
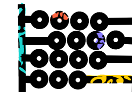
- **State-oriented behavior**
  Required for reactive systems; classical automata insufficient.
- **Event-handling**
  (external or internal events)
- **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state



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

Requirements for specification techniques (6)

- **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**

Appropriate model of computation (MoC):

Von Neumann languages do not match well with requirements for embedded system design:

- lack of facilities for describing timing
- are a source of problems:
  - potential deadlocks,
  - undecidability of termination,
  - poor communication facilities
  - …

☞ let's have a look at an example!

Problems with von Neumann Computing

- Thread-based multiprocessing may access global variables
- We know from the theory of operating systems that
  - access to global variables might lead to race conditions
  - To avoid these, we need to use mutual exclusion.
  - Mutual exclusion may lead to deadlocks.
  - Avoiding deadlocks is possible only if we accept performance penalties.

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*."

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.

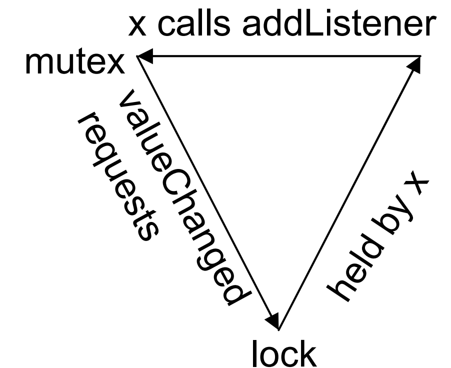
Example: Observer Pattern with Mutual Exclusion (mutexes)

```
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!

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."*

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

Problems with classical CS theory and von Neumann computing

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.

Edward A. Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

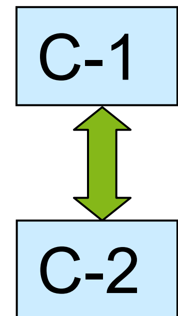
☞ Search for non-thread-based, non-von-Neumann MoCs; which are the requirements for specification techniques?

Models of computation
- Definition -

What does it mean, "to compute"?

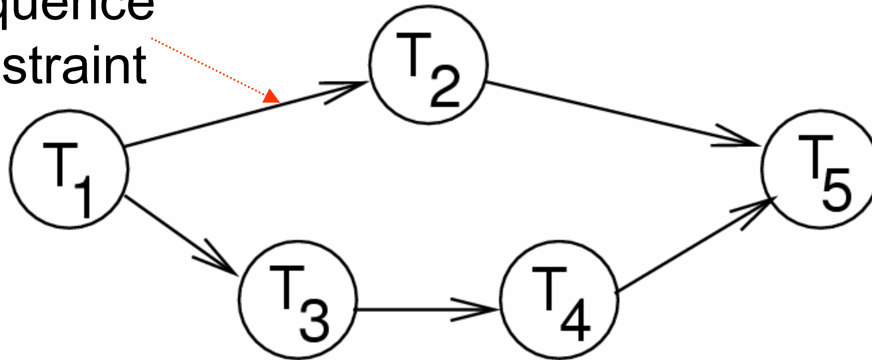
Models of computation define:

- Components and an execution model for computations for each component
- Communication model for exchange of information between components.



Dependence graph
Definition

Sequence constraint

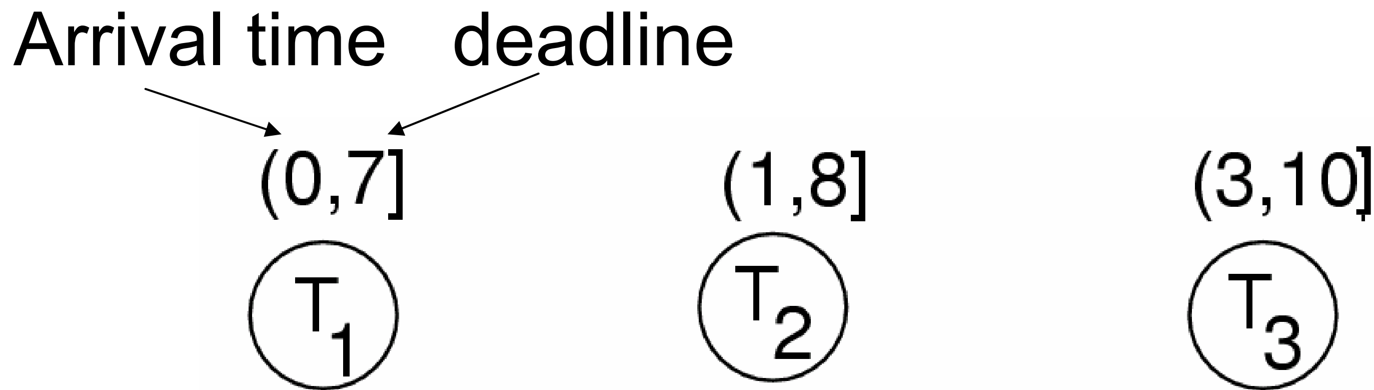


Nodes could be programs or simple operations

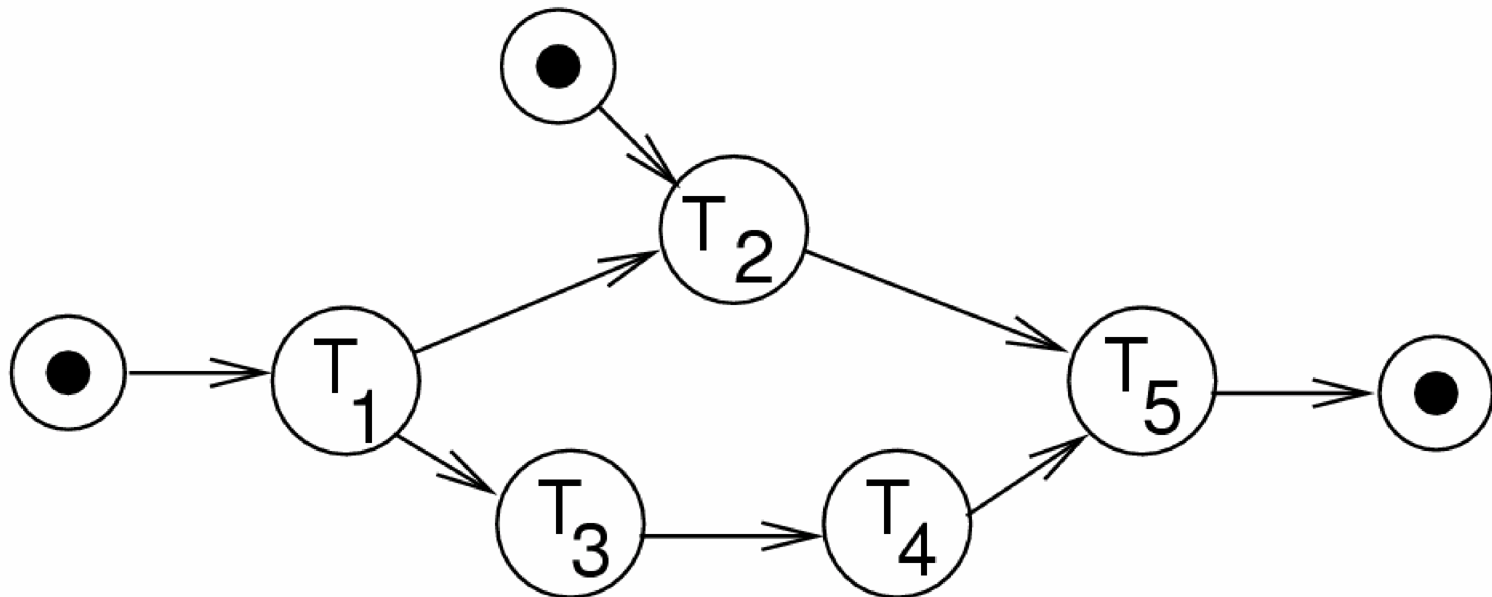
Def.: A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.
If $(v1, v2) \in E$, then $v1$ is called an **immediate predecessor** of $v2$ and $v2$ is called an **immediate successor** of $v1$.
Suppose E^* is the transitive closure of E.
If $(v1, v2) \in E^*$, then $v1$ is called a **predecessor** of $v2$ and $v2$ is called a **successor** of $v1$.

Dependence graph
Timing information

Dependence graphs may contain additional information, for example: Timing information

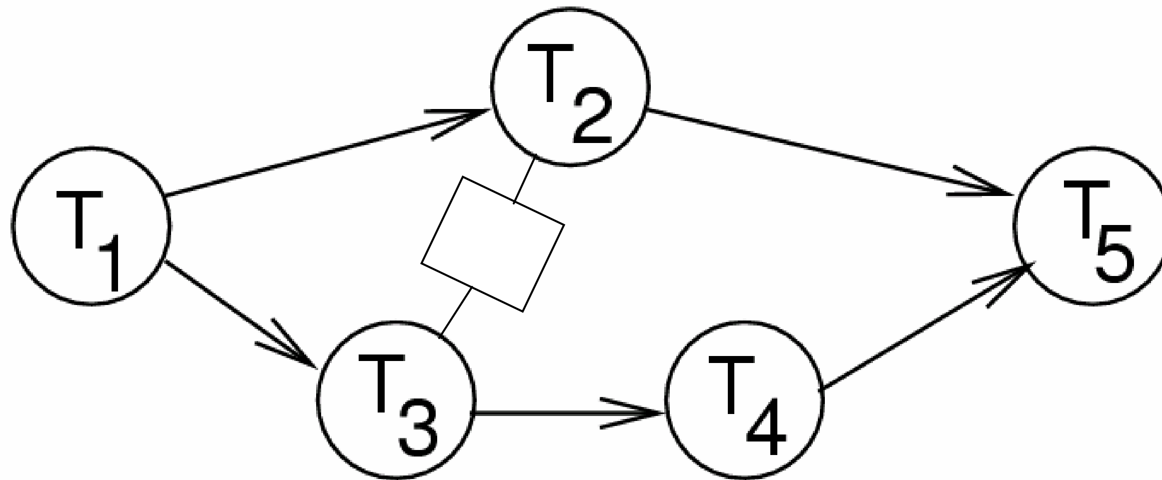


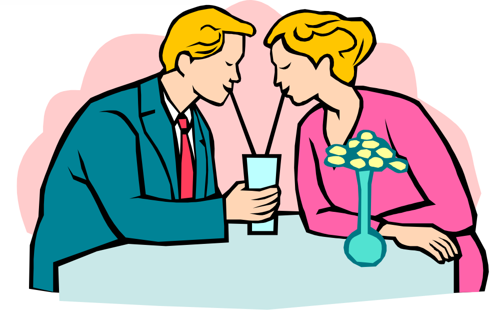
Dependence graph
I/O-information



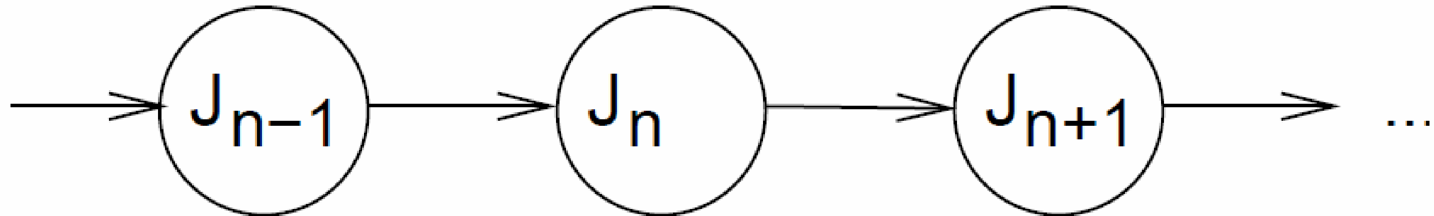
Dependence graph

Shared resources





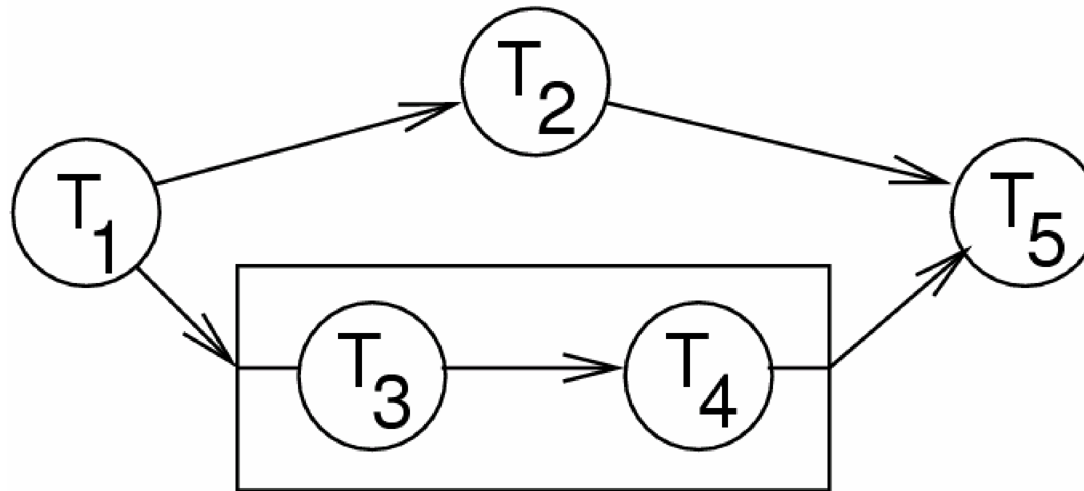
Dependence graph
Periodic schedules



- A **job** is single execution of the dependence graph
- Periodic dependence graphs are infinite

Dependence graph

Hierarchical task graphs



© p. marwedel, informatik 12, 2009

Models of 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
}
```

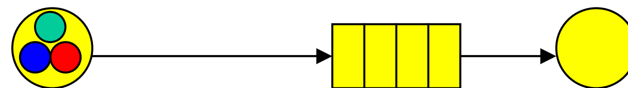
Race-free access to shared memory protected by S possible

This model may be supported by:

- mutual exclusion for critical sections
- cache coherency protocols

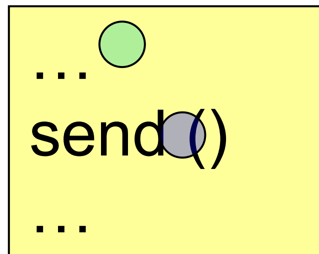
Non-blocking/asynchronous message passing

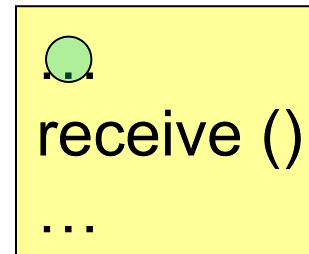
Sender does not have to wait until message has arrived; potential problem: buffer overflow

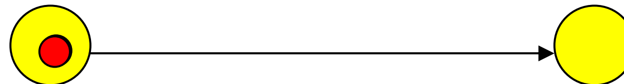


Blocking/synchronous message passing *rendez-vous*

Sender will wait until receiver has received message

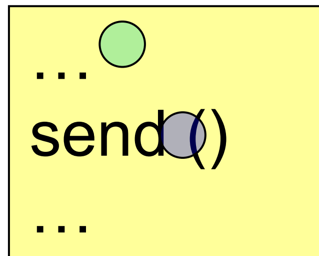


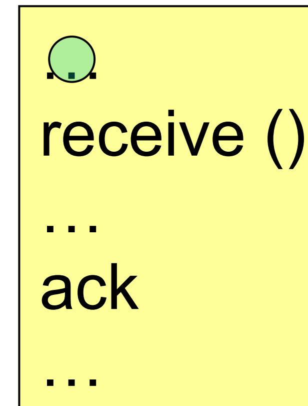




Extended *rendez-vous*

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



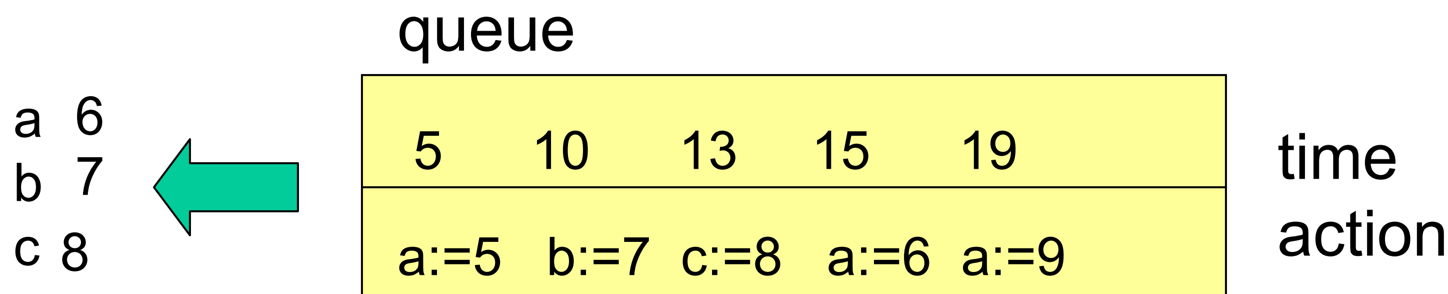


Organization of computations within the components (1)

- Von Neumann model

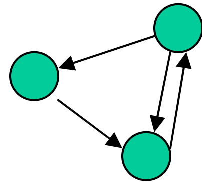
  Sequential execution, program memory etc.

- Discrete event model



Organization of computations within the components (2)

- Finite state machines



- Differential equations

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

Models of computation considered in this course

<table>
<tr><th rowspan="2">Communication/
local computations</th><th rowspan="2">Shared
memory</th><th colspan="2">Message passing</th></tr>
<tr><th>Synchronous</th><th>Asynchronous</th></tr>
<tr><td>Undefined
components</td><td colspan="3">Plain text, use cases
| (Message) sequence charts</td></tr>
<tr><td>Communicating finite
state machines</td><td>StateCharts</td><td></td><td>SDL</td></tr>
<tr><td>Data flow</td><td>(Not useful)</td><td></td><td>Kahn networks,
SDF</td></tr>
<tr><td>Petri nets</td><td></td><td colspan="2">C/E nets, P/T nets, …</td></tr>
<tr><td>Discrete event (DE)
model</td><td>VHDL,
Verilog,
SystemC, …</td><td colspan="2">Only experimental systems, e.g.
distributed DE in Ptolemy</td></tr>
<tr><td>Von Neumann model</td><td>C, C++,
Java</td><td colspan="2">C, C++, Java with libraries
CSP, ADA |</td></tr>
</table>

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 Edward A. Lee, UCB
- Axel Jantsch: Modeling Embedded Systems and Soc's: Concurrency and Time in Models of Computation, Morgan-Kaufman, 2004

Ptolemy

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

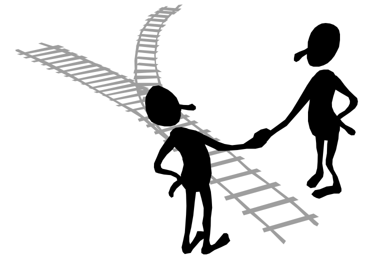
http://ptolemy.berkeley.edu/

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

Search for other models of computation =

- models of components
  - finite state machines (FSMs)
  - data flow, ….
- models for communication
  - Shared memory
  - Message passing