EEE3096S

# Specifications and Modelling

Lecture

# Embedded Systems II

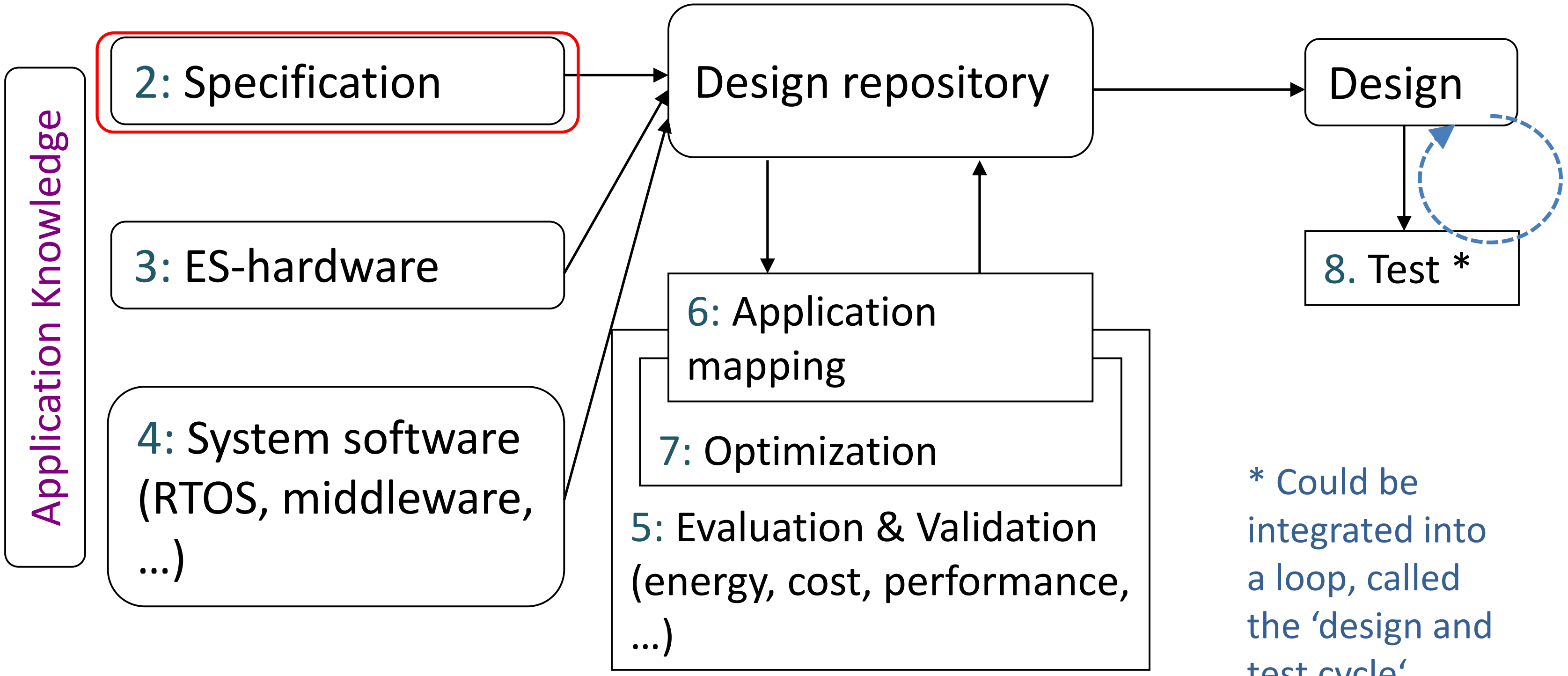
Dr Simon Winberg

Electrical Engineering
University of Cape Town

## Outline

- Specifications and Requirements
- Models of computation
- Customer tour
- Selection of problems
  - Discussion from sect 1.8.

# Hypothetical design flow



* Could be integrated into a loop, called the 'design and test cycle' (D&T cycle)

Generic ES design cycle

The number in the diagram denote chapters of the textbook

# Requirements and Specification

## Requirements =

Requirements can be totally abstract things, sometimes a bit vague. They are simply things or operations that the user (or stakeholder) requires.

## Specifications =

*(sometimes called "requirements specifications")*

This is more specific. A clear description of what the system needs to do, much more precise. May include issues such as specific timing limits.
The (initial) requirements drives the specifications.
The specifications drives the design.

*The specifications document for a project is generally a refinement of the requirements (e.g. indicating physical constaints) together with more detail on other behavioural and structural aspects.*

## Example

Req.

Make me an MP3 player!

Req.

Do you want that with earphones and memory card reader?

It must be a DINN connector, because it makes me think of “Dinner”. Must be PCMCIA because my people say it is more expensive in the shops.

Team

oned)

Spec.

Do you agree to a DINN connector and PCMCIA card reader? You realize those standards are poorly supported nowadays?

So what are the requirements and what are the specifications? …

Hopefully that rather simplistic example will help you remember the difference

## Motivation for considering specs & models

– Why considering specs and models in detail?

– 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* of an ES 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*.

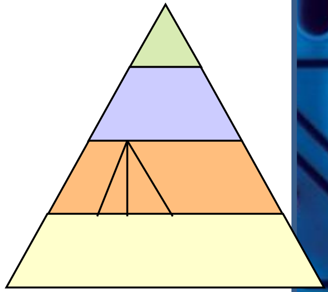
[Jantsch, 2004]:

Which requirements do we have for our models? …

(i.e. for creating out models not for creating the final ES product)

## Requirements for specification & modeling techniques: Hierarchy

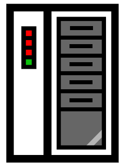
- **Hierarchy**
  Humans unable of (easily) understand systems containing more than ~5 objects.
  Most actual systems require more objects
  ☞ Hierarchy (+ Abstraction)



– **Behavioral hierarchy**
  Examples: states, processes, procedures.

proc
  proc
    proc

– **Structural hierarchy**
  Examples: processors, racks, printed circuit boards



## Requirements for Component-based design

- Systems must be designed from components*
- Must be “easy” to derive behaviour of system from behaviour of its subsystems

(Detailed further by Sifakis, Thiele, Ernst, …)

**Needs to support**

- Concurrency
  components able to work together potentially at the same time
- Synchronization and communication
  able to synchronize component / talk or control it from another component

* i.e. parts whose functionality and structure can be explained without needing to know much about the containing system.

## Requirements for Timing

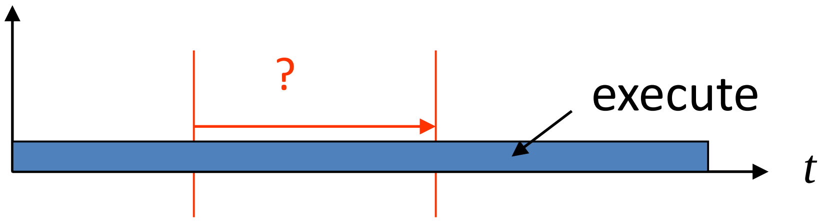
- Timing behaviour
  *Essential for embedded / CPS systems!*
  - Additional information (periods, dependences, scenarios, use cases)
  - Speed of the underlying platform must be known
  - Far-reaching consequences for design processes! Can take much time and effort to get the timing right!

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

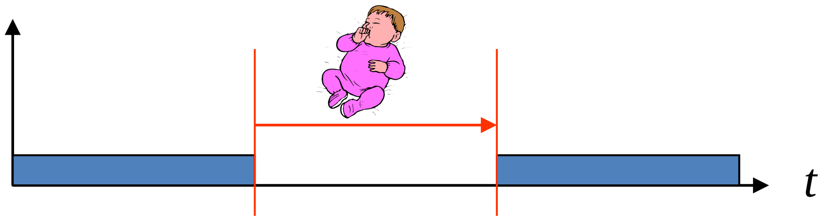
## Requirements for Timing

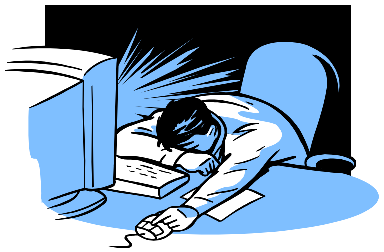
Four types of timing specs required (Burns, 1990):

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



2. Means for delaying processes



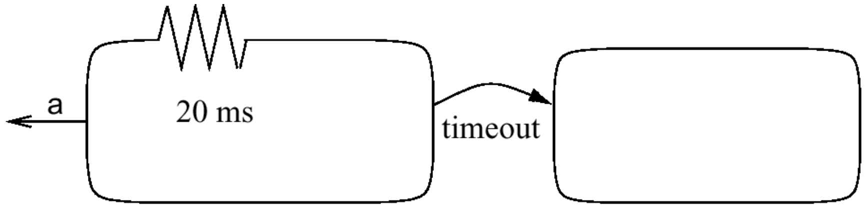


Concept of ‘wall clock time’ (WCT) i.e. the real time that tasks take, as opposed to simulation time or ‘cpu time’  (cpu time would be *at best* 100% of WCT).

## Requirements for Timing

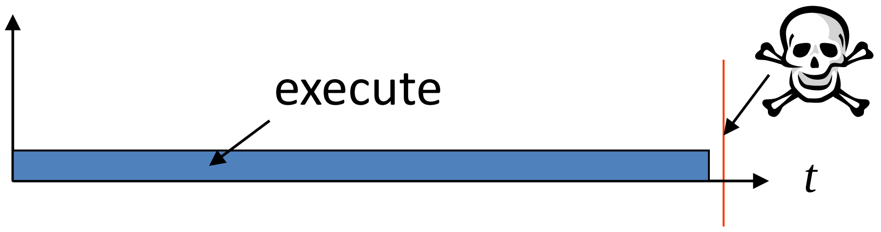
3. Possibility to specify timeouts

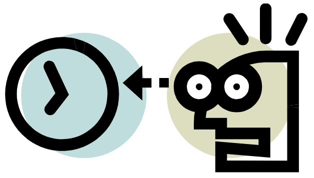
   Stay in a certain state a maximum time.



4. Methods for specifying deadlines

   Not available or in separate control file.





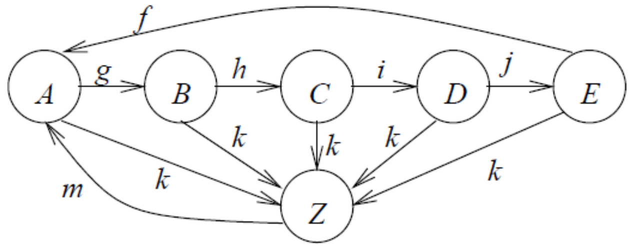
# Support for designing reactive systems

– **State-oriented behavior**
  Required for reactive systems; classical automata insufficient.

  *Why…?*

– **Event-handling**
  (external or internal events)

– **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state

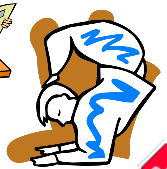


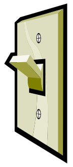
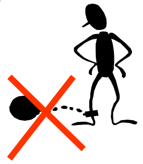
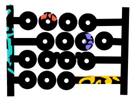
*Because a reactive system needs to respond (within a particular time) from any state it happens to be in. Hence the mass of k connections above.*

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

# Specification & modeling techniques

- **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**
- **No obstacles for efficient implementation**
- **Adequate model of computation**
  What does it mean "to compute" and to be computable? …



## Computable

Def.: Computable

A. The problem can be solved.

[✓ B.] There exists an algorithm to solve the problem.

C. The method to solve the problem can be represented as a turning machine that will terminate.

*Any guesses?*

## Specification & modeling techniques

- Even the core notion of “computable” is at odds with the requirements of embedded software[1].
  - In the CS notion, useful computation ‘terminates’, but termination is undecidable
- In embedded software, termination is failure (usually), and yet to get predictable timing, sub-computations must decidably terminate(!)
- *What is needed is nearly a reinvention of computer science.*

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

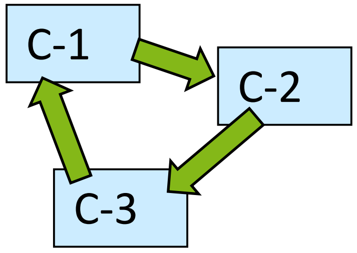
## Models of Computation (MoCs)

# Models of Computation

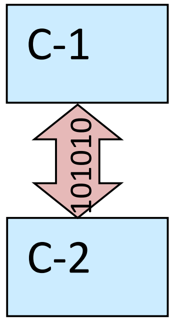
**What does it mean "to compute"?**

**Models of computation define**:

Components and an execution model for computations *for each* component



Communication model for exchange of information between components.

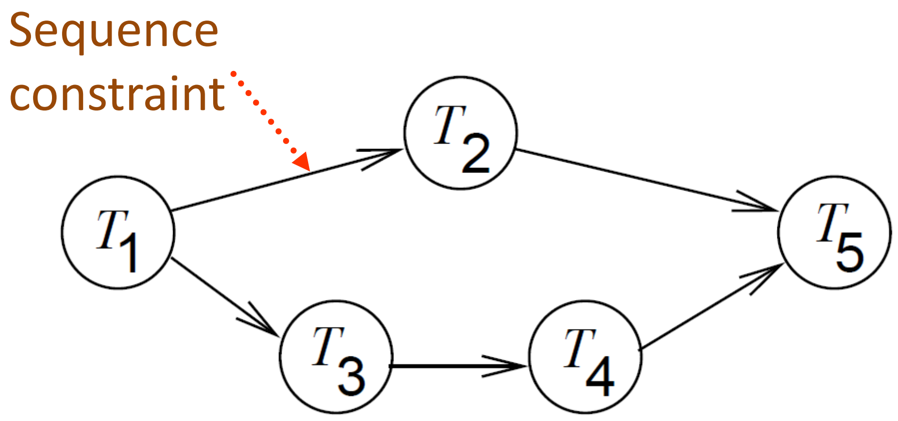


*This leads to notion of 'spatial computing' rather than traditional 'temporal computing' and component based development (CBD)*

*Spatial computing is a new trend for thinking about parallel computing*

Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing.

# Dependence graph: Definition



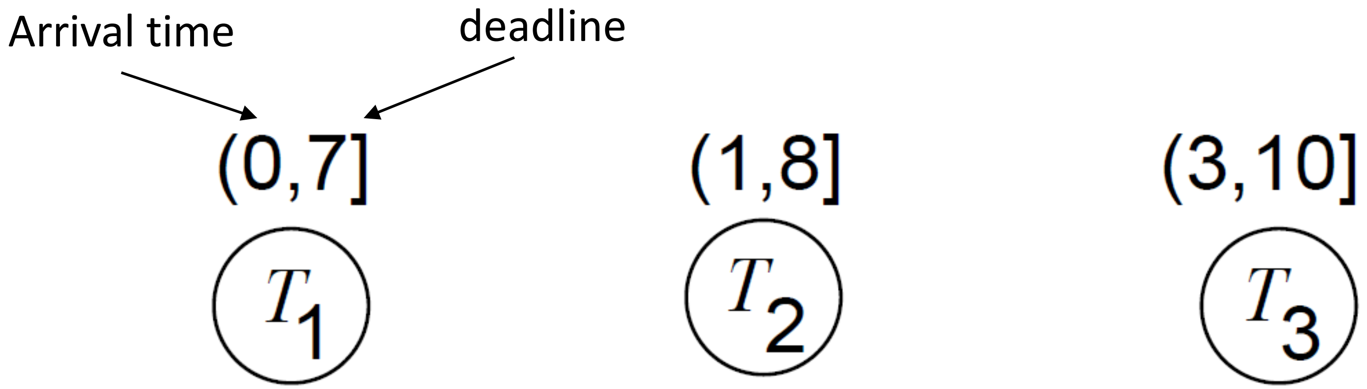
Nodes could be programs or simple operations

- **Def.:** A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a relation.

- If $(v_1, v_2) \in E$, then $v_1$ is called an **immediate predecessor** of $v_2$ and $v_2$ is called an **immediate successor** of $v_1$.
- Suppose $E^*$ is the transitive closure of $E$.
  If $(v_1, v_2) \in E^*$, then $v_1$ is called a **predecessor** of $v_2$ and $v_2$ is called a **successor** of $v_1$.

# Dependence graph: Timing information

Dependence graphs may contain additional information, for example: Timing information for arrival time, deadline, period, execution time

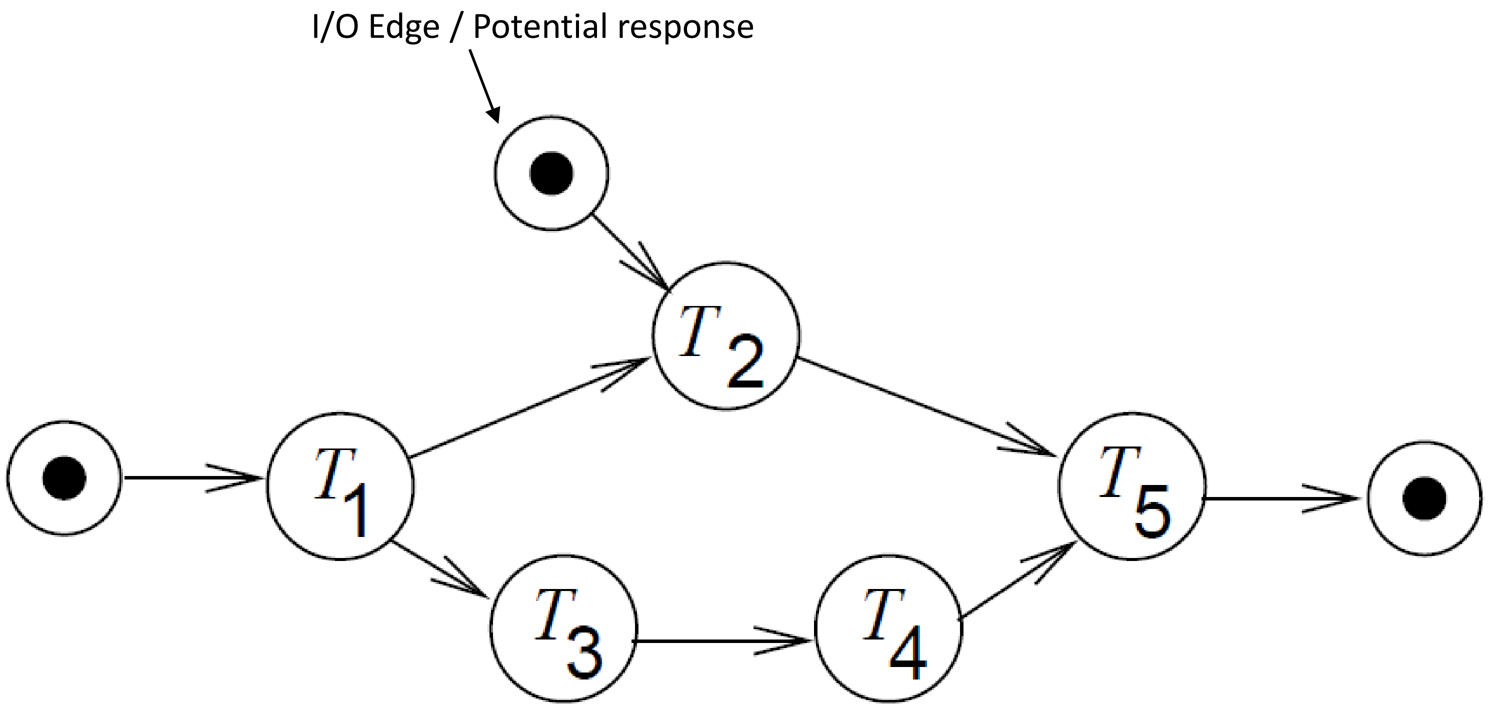


( = soft [ = hard

(A,B] = can start round A but must complete at or before B

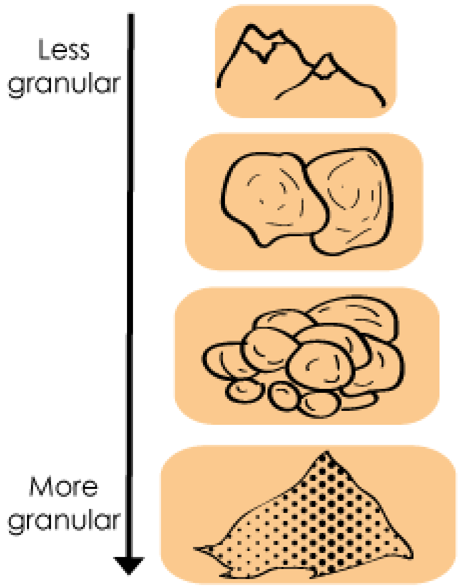
[A,B] = must start at A and must end at B (this is more difficult to schedule)

## Dependence graph: I/O-information

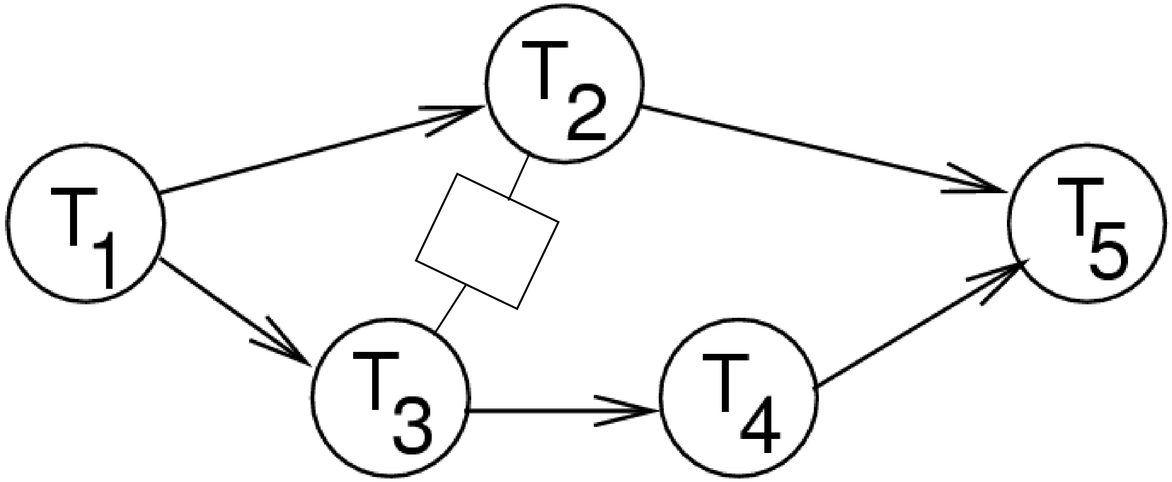


## Graph node complexity

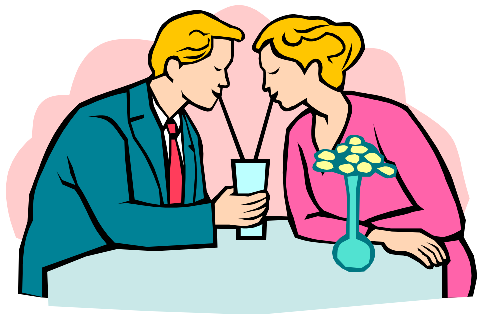
- Each node (or task) may represent just one line of code, or possibly thousands (e.g. whole subprograms)
- This is called “Graph node complexity”
  - Another name for this is node granularity



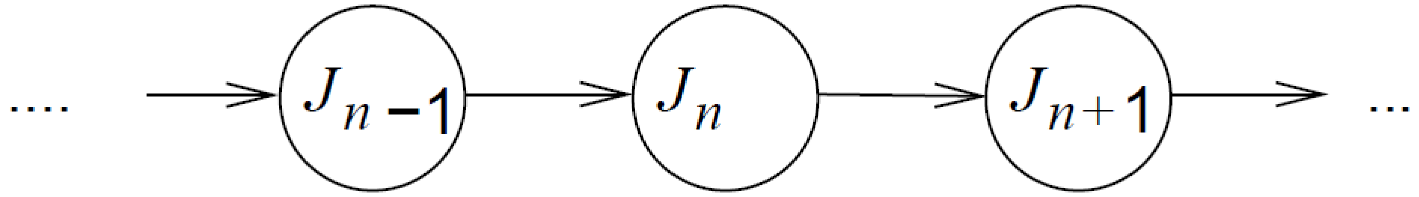
# Dependence graph: Shared resources



You don’t need to indicate the specific type of shared resource, this can indicate potential contention over a resource that may at times be needed by another tasks that maybe can’t be shared, or only allow limited sharing.

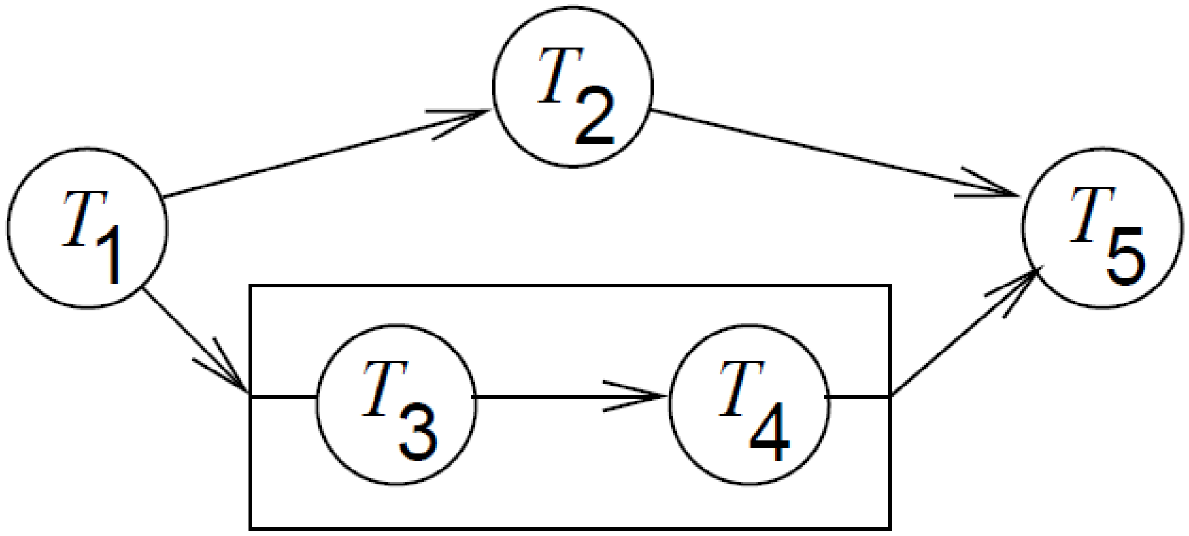


## Dependence graph: Periodic schedules



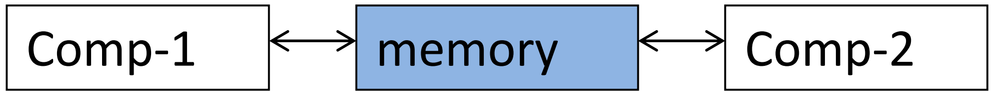
- A **job** is single execution of the dependence graph
- Periodic dependence graphs are infinite
  (i.e. they run again and again, could go on forever)

## Dependence graph: Hierarchical task graphs



## Communication by shared memory

**Shared memory**



Variables accessible to several components/tasks.

This model is mostly restricted to local systems (single computer) – *but not necessarily*, e.g. when using Distributed Shared Memory (DSM) which may be implemented as a hardware architecture or virtualized via the OS.

## 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
}
```

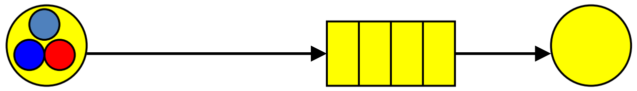
- Unexpected u=6 possible if P(S) and V(S) guards is not used (i.e. double context switch before execution of {u = u+1})
- S: semaphore
- P(S) grants up to *n* concurrent accesses to resource
- *n*=1 in this case (mutex/lock)
- V(S) increases number of allowed accesses to resource
- Thread-based (imperative) model should be supported by mutual exclusion for critical sections

# Non-blocking/asynchronous message passing

Sender does not have to wait until message has arrived;

```
…
send ()
…
```

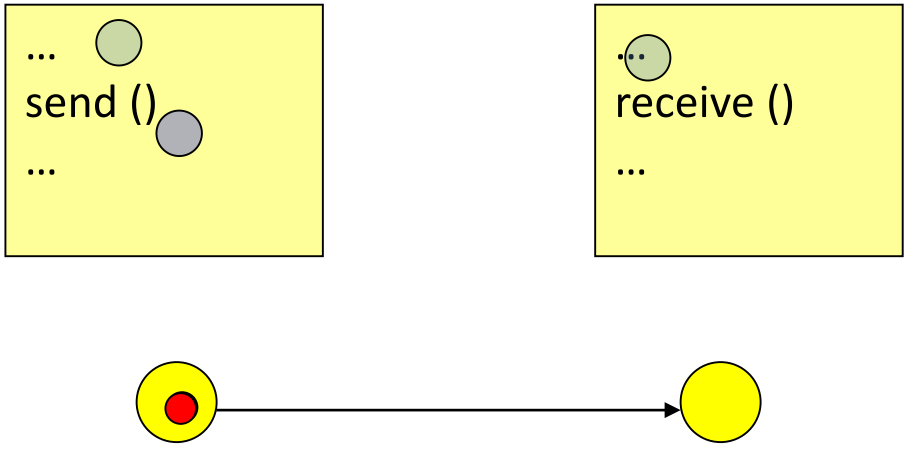
```
…
receive ()
…
```



*Potential problem: buffer overflow*

## Blocking/synchronous message passing - *rendez-vous*

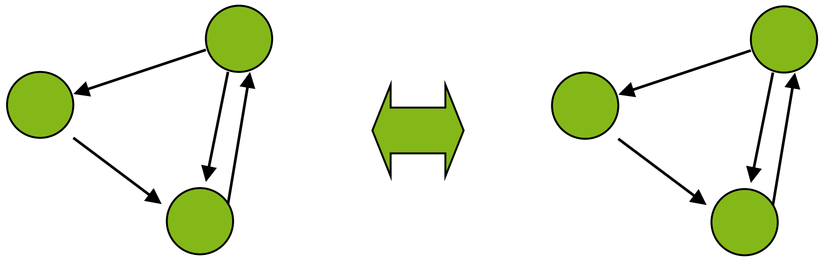
Sender will wait until receiver has received message



*No buffer overflow, but reduced performance.*

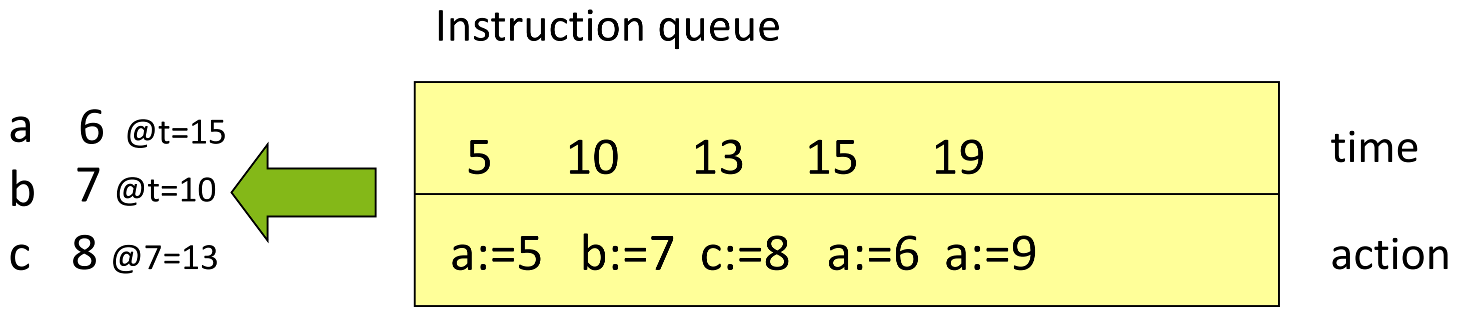
## Organization of computations within the components

Finite state machines



## Organization of computations within the components

### Discrete event model



### Von Neumann model

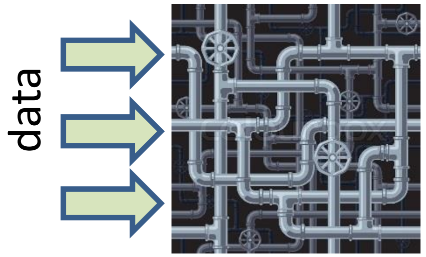
Sequential execution, program memory etc.

## Organization of computations within components

- Differential equations
  (e.g. rate of change of inputs and outputs of the component)

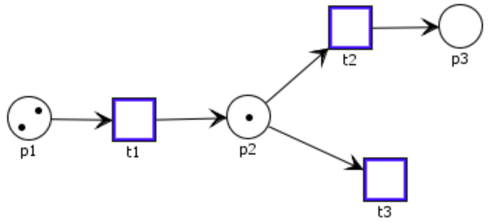
$$\frac{\partial^2 x}{\partial t^2} = b$$

- Data flow
  (models flow of data in a distributed system)



- Petri nets
  (models synchronization in a distributed system)

(how much Petri nets have you all done already?)



# Models of computation 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 (Ptolemy only discussed briefly) | |
| Von Neumann model | C, C++, Java | C, C++, Java with libraries<br>CSP, ADA | |

* Classification based on implementation with centralized data structures
SystemC will not be delved into detail. Only brief flavour of VHDL and Verilog given

## Characteristics of Embedded Systems

End of Lecture

## The Next Episode…

## Lecture P02

P02: Operating systems & introduction to the embedded Linux design. (Part 1)

## References and Acknowledgements

This presentation partly based on slides from the textbook’s companion website. Used with permission by the author Prof. Peter Marwedel

Textbook:
Embedded System Design:
Embedded Systems Foundations of Cyber-Physical Systems
By Peter Marwedel, TU Dortmund