

Specification Languages for Embedded Systems

Peter Marwedel

University of Dortmund, Germany

Specification of embedded systems:

Requirements for specification techniques (1)

- **Hierarchy**

Humans not capable of understanding systems containing more than ~5 objects.

Most actual systems require more objects

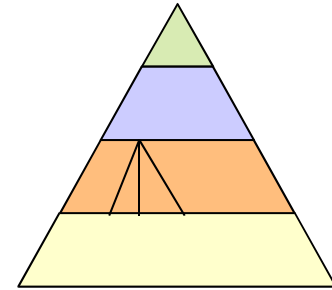
☞ Hierarchy

- Behavioral hierarchy

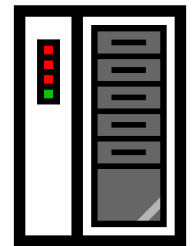
Examples: states, processes, procedures.

- Structural hierarchy

Examples: processors, racks, printed circuit boards

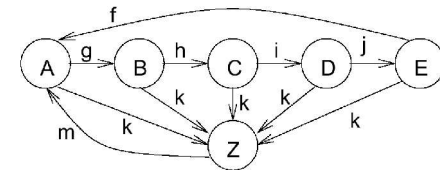


proc
proc
proc



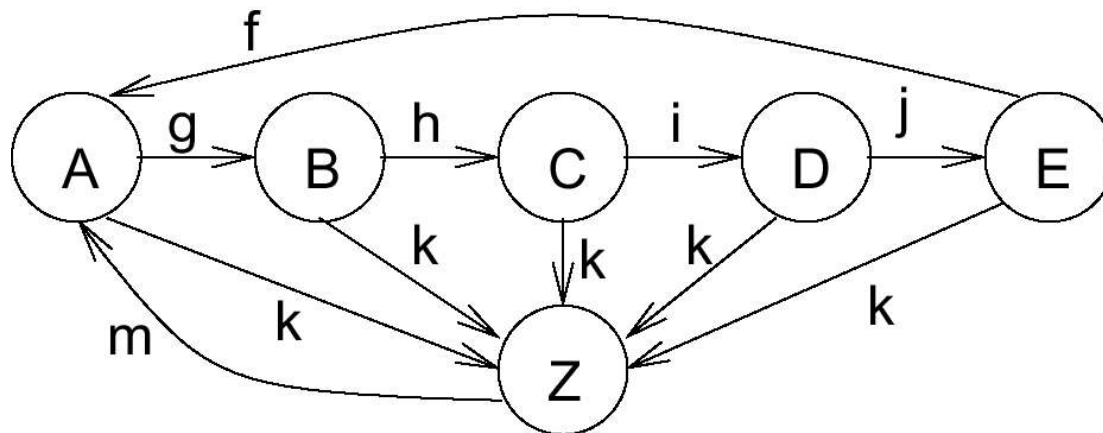
Specification of embedded systems: Requirements for specification techniques (2)

- **Timing behavior.**
- **State-oriented behavior**
Required for reactive systems;
classical automata insufficient.
- **Event-handling**
(external or internal events)
- **No obstacles for efficient implementation**



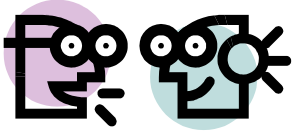
Requirements for specification techniques (3)

- **Support for the design of dependable systems**
Unambiguous semantics, ...
- **Exception-oriented behavior**
Not acceptable to describe exceptions for every state.



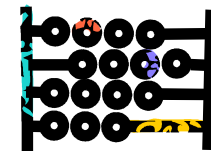
We will see, how all the arrows labeled k can be replaced by a single one.

Requirements for specification techniques (4)

- **Concurrency**
Real-life systems are concurrent
- **Synchronization and communication**
Components have to communicate! 
- **Presence of programming elements**
For example, arithmetic operations, loops, and function calls should be available
- **Executability** (no algebraic specification)
- **Support for the design of large systems** (☞ OO)
- **Domain-specific support**

Requirements for specification techniques (5)

- **Readability**
- **Portability and flexibility**
- **Termination**
It should be clear, at which time all computations are completed
- **Support for non-standard I/O devices**
Direct access to switches, displays, ...
- **Non-functional properties**
fault-tolerance, disposability, EMC-properties, weight, size, user friendliness, extendibility, expected life time, power consumption...
- **Adequate model of computation**

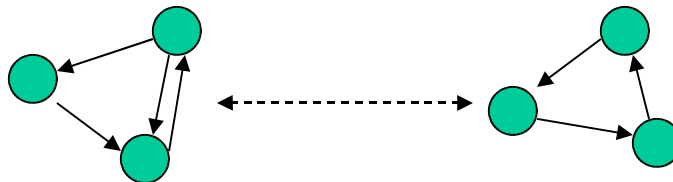


Models of computation

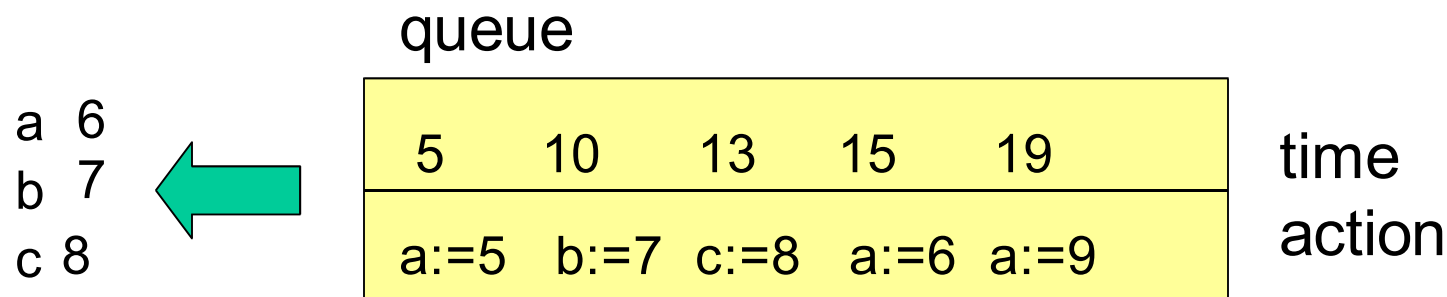
- Examples (1) -

Von Neumann model not good for embedded systems

- Communicating finite state machines (CFSMs):



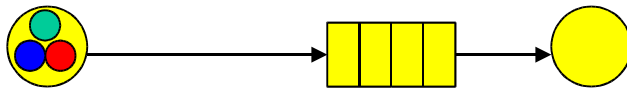
- Discrete event model



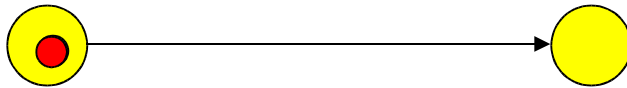
Models of computation

- Examples (2) -

- Process networks with asynchronous message passing



- Process networks with synchronous message passing



- Continuous time modeling with differential equations

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



Facing reality

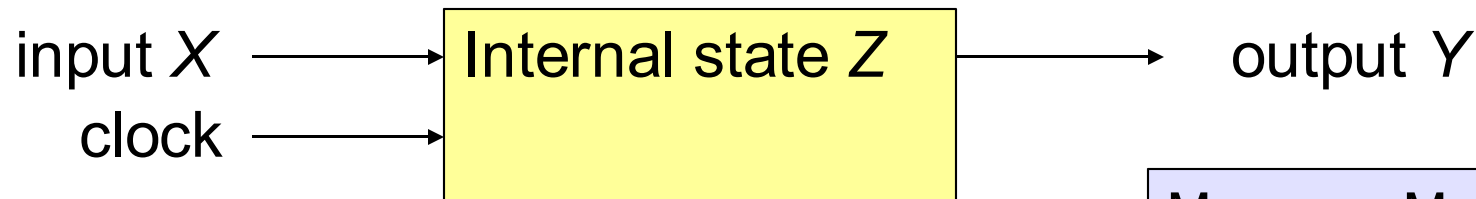
No language that meets all language requirements

- ☞ using compromises

- ☞ StateCharts, based on shared memory communication.

StateCharts: recap of classical automata

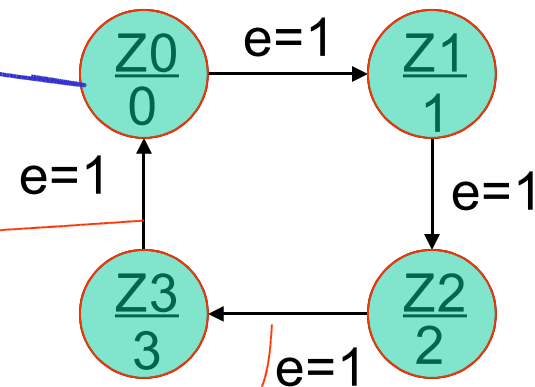
Classical
automata:



Next state Z^+ computed by function δ
Output computed by function λ

Moore- + Mealy
automata=finite state
machines (FSMs)

- Moore-automata:
 $Y = \lambda (Z); \quad Z^+ = \delta (X, Z)$
- Mealy-automata
 $Y = \lambda (X, Z); \quad Z^+ = \delta (X, Z)$



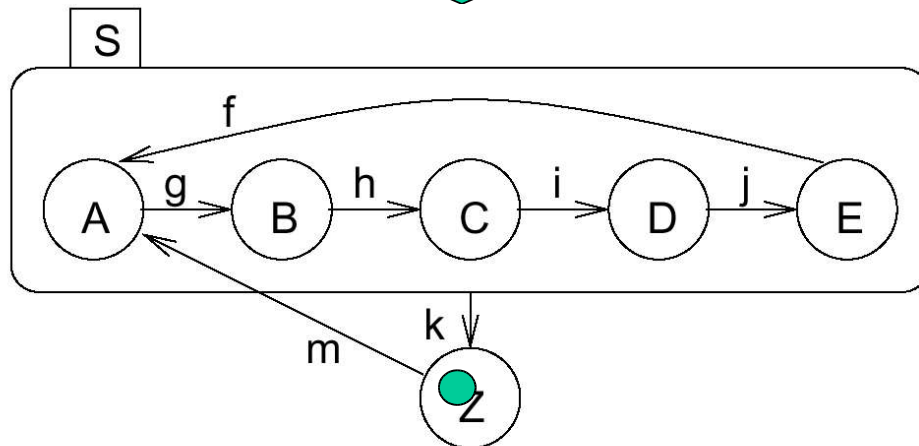
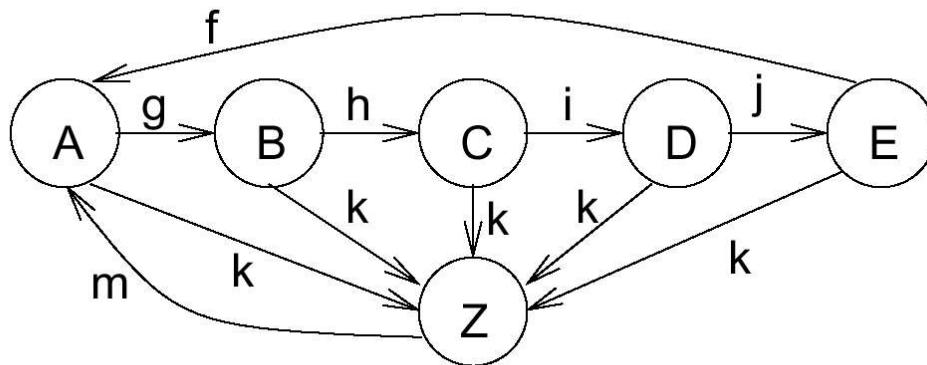
StateCharts

Classical automata not useful for complex systems
(complex graphs cannot be understood by humans).

☞ Introduction of hierarchy ☞ StateCharts [Harel, 1987]

StateChart = *the only unused combination of*
„flow“ or „state“ with „diagram“ or „chart“

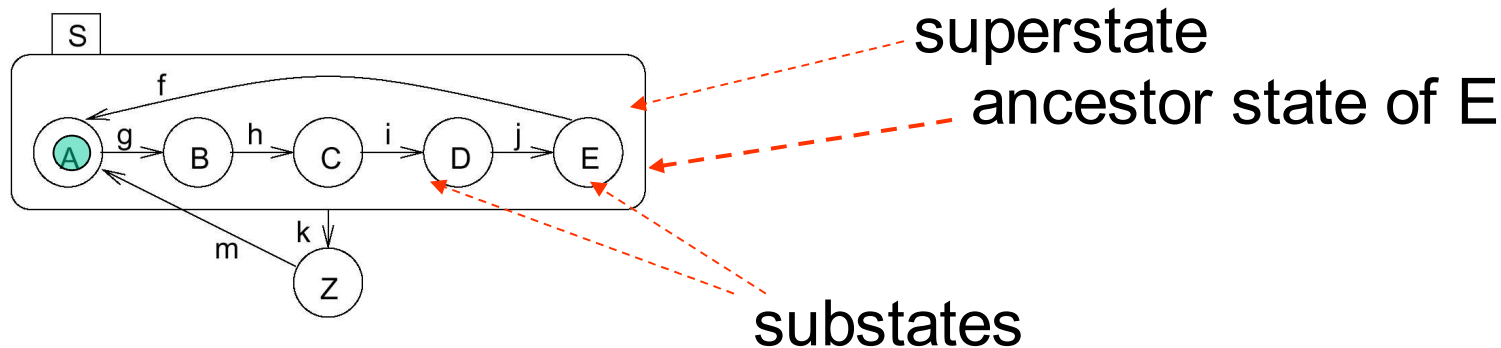
Introducing hierarchy



FSM will be **in** exactly one of the substates of S if S is **active** (either in A or in B or ..)

Definitions

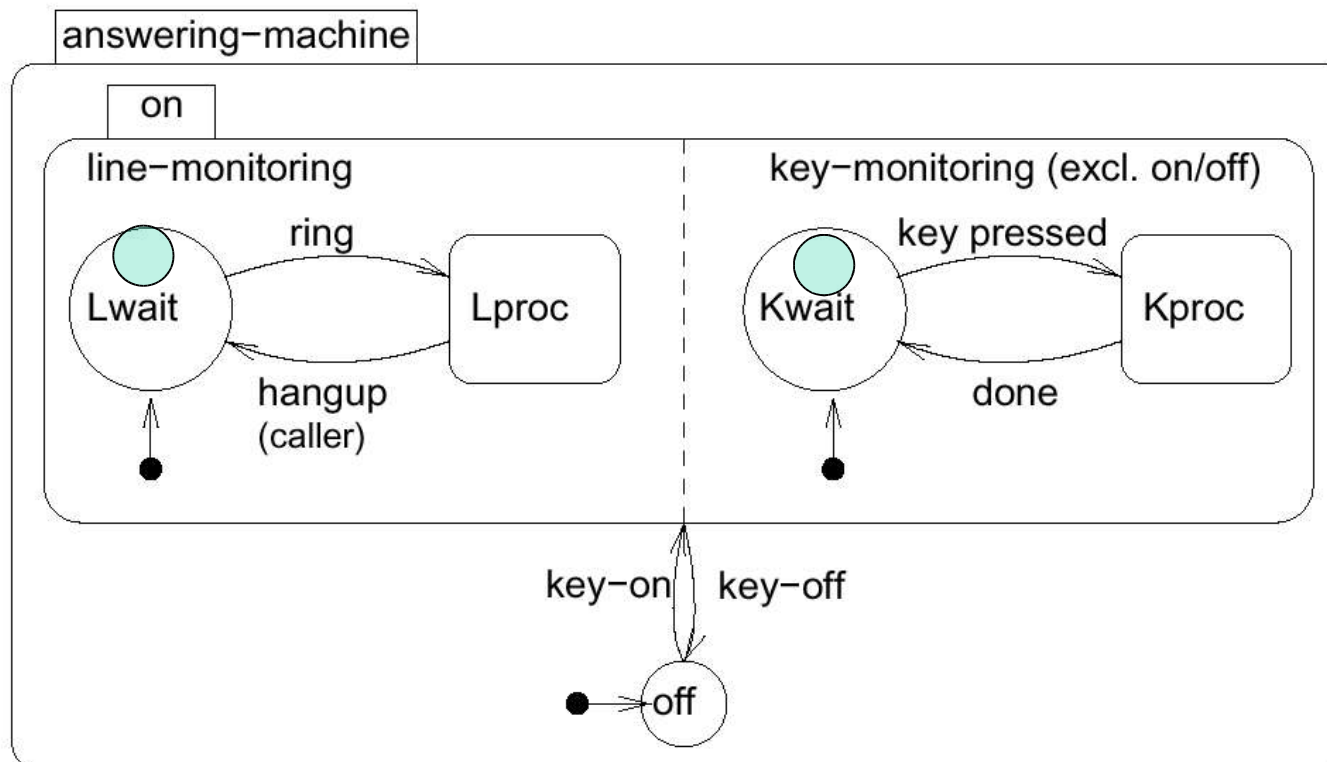
- Current states of FSMs are also called **active** states.
- States which are not composed of other states are called **basic states**.
- States containing other states are called **super-states**.
- For each basic state s , the super-states containing s are called **ancestor states**.
- Super-states S are called **OR-super-states**, if exactly one of the sub-states of S is active whenever S is active.



Concurrency

Convenient ways of describing concurrency are required.

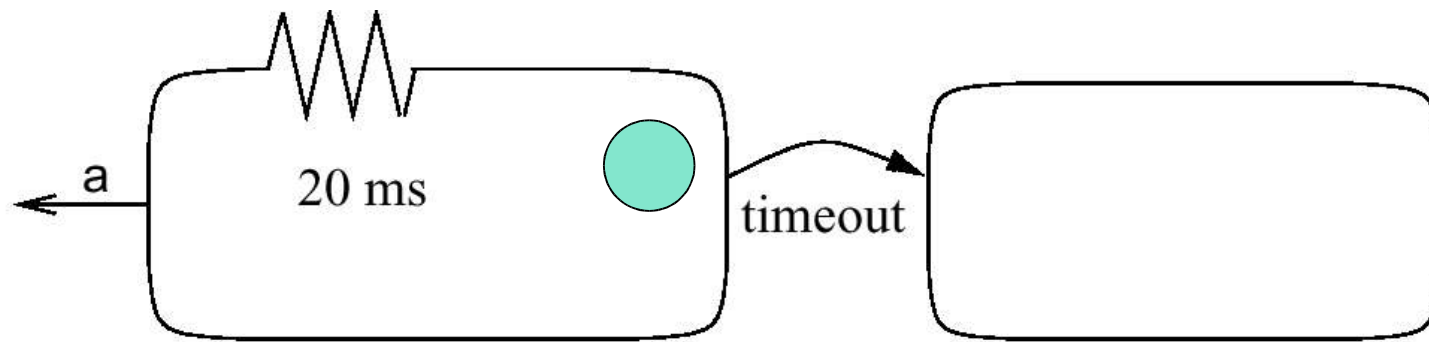
AND-super-states: FSM is in **all** (immediate) sub-states of a super-state; Example:



Timers

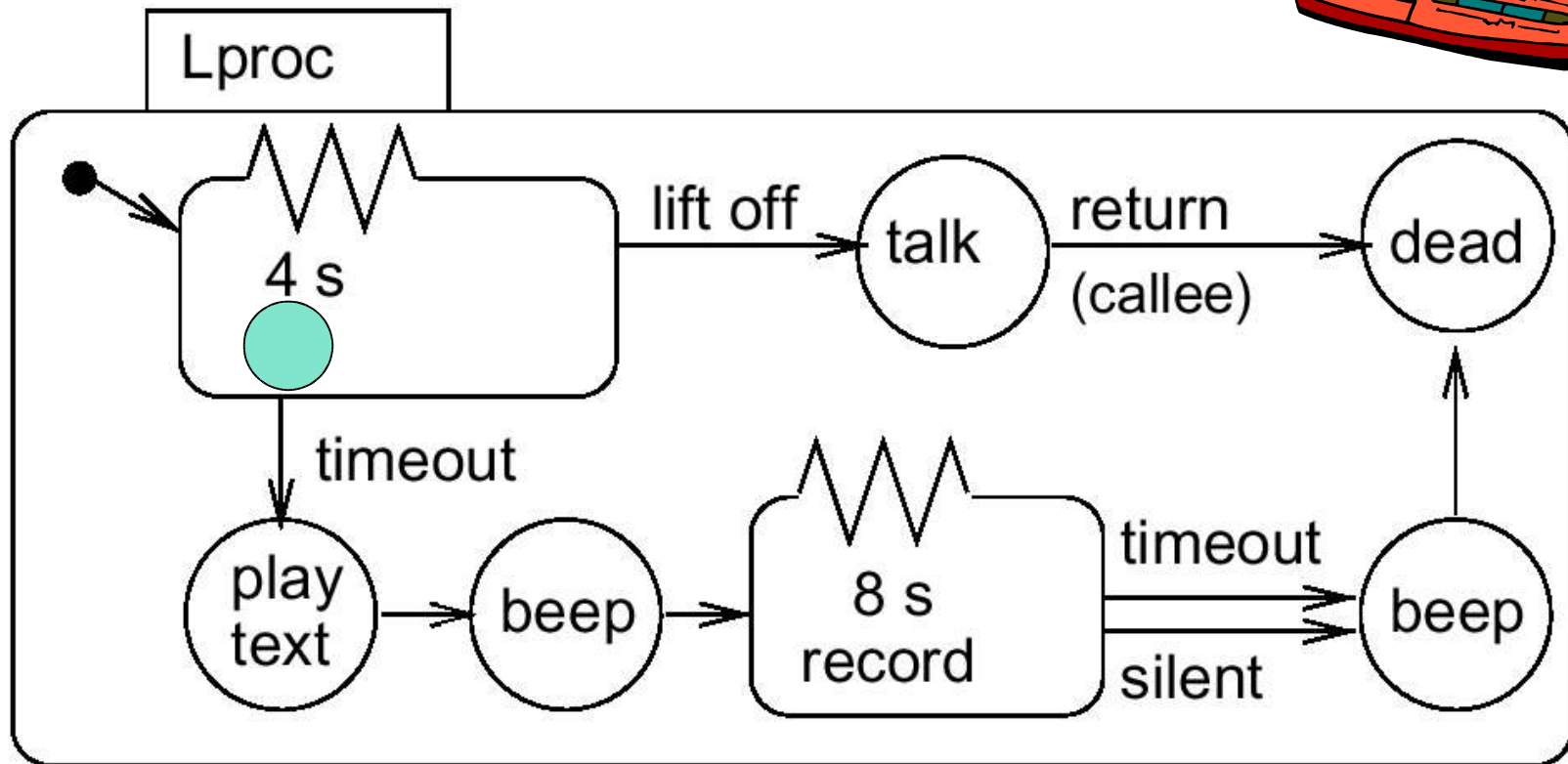
Since time needs to be modeled in embedded systems, timers need to be modeled.

In StateCharts, special edges can be used for timeouts.



If event *a* does not happen while the system is in the left state for 20 ms, a timeout will take place.

Using timers in an answering machine





Broadcast mechanism

Values of variables are visible to all parts of the StateChart model.

New values become effective after each evaluation cycle.

- ☞ StateCharts implicitly assumes a **broadcast** mechanism for variables.
- ☞ StateCharts is **appropriate for local control systems** (☺), but not for distributed applications for which updating variables might take some time (☹).

Evaluation of StateCharts (1)

Pros:

- Hierarchy allows arbitrary nesting of AND- and OR-super states.
- (StateMate-) Semantics defined in a follow-up paper to original paper.
- Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
- Available „back-ends“ translate StateCharts into C or VHDL, thus enabling software or hardware implementations.

Evaluation of StateCharts (2)

Cons:

- Generated C programs frequently inefficient,
- **Not useful for distributed applications,**
- No program constructs,
- No description of non-functional behavior,
- No object-orientation,
- No description of structural hierarchy.

Extensions:

- Module charts for description of structural hierarchy.

What to use for distributed applications?

Petri nets

Introduced in 1962 by Carl Adam Petri in his PhD thesis.

Focus on modeling causal dependencies;

No global synchronization assumed (*message passing only*).

Key elements:

- **Conditions**

Either met or no met.

- **Events**

May take place if certain conditions are met.

- **Flow relation**

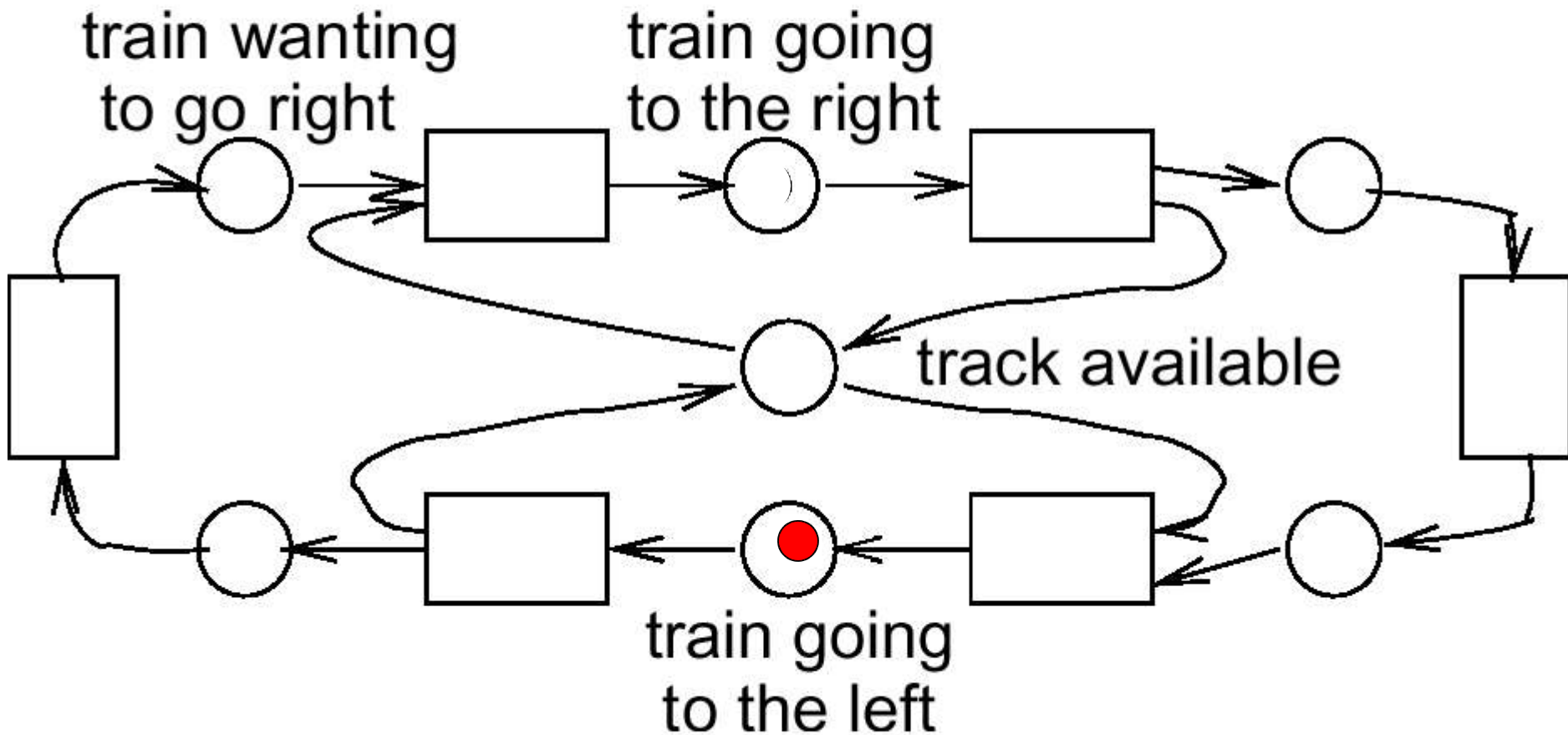
Relates conditions and events.

Conditions, events and the flow relation form a **bipartite graph** (graph with two kinds of nodes).

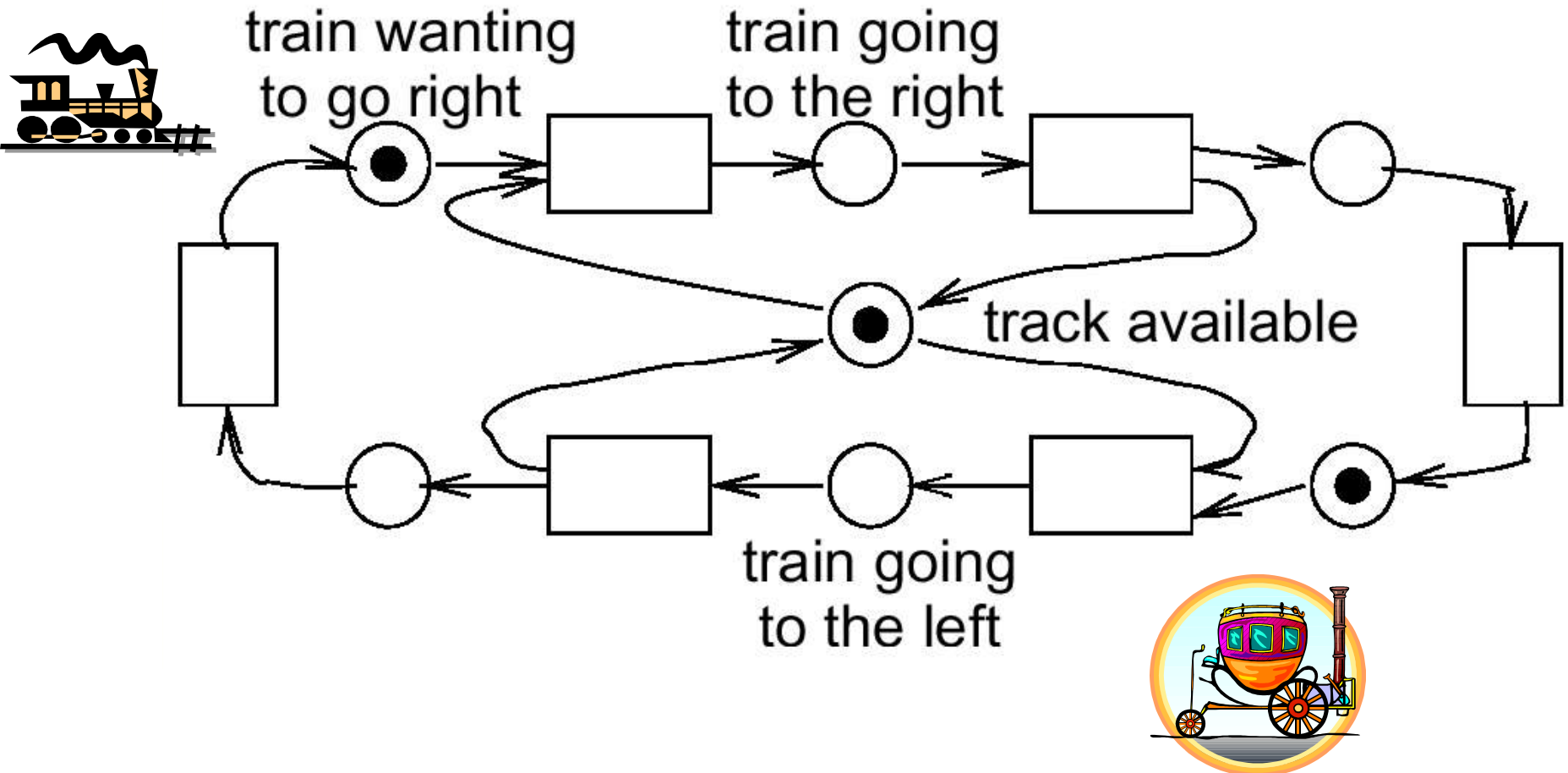
© P. Marwedel, Univ. Dortmund/Informatik 12 + ICD/ES, 2006 Tue1 - 21 -



Playing the „token game“



Conflict for resource „track“



More complex example (1)

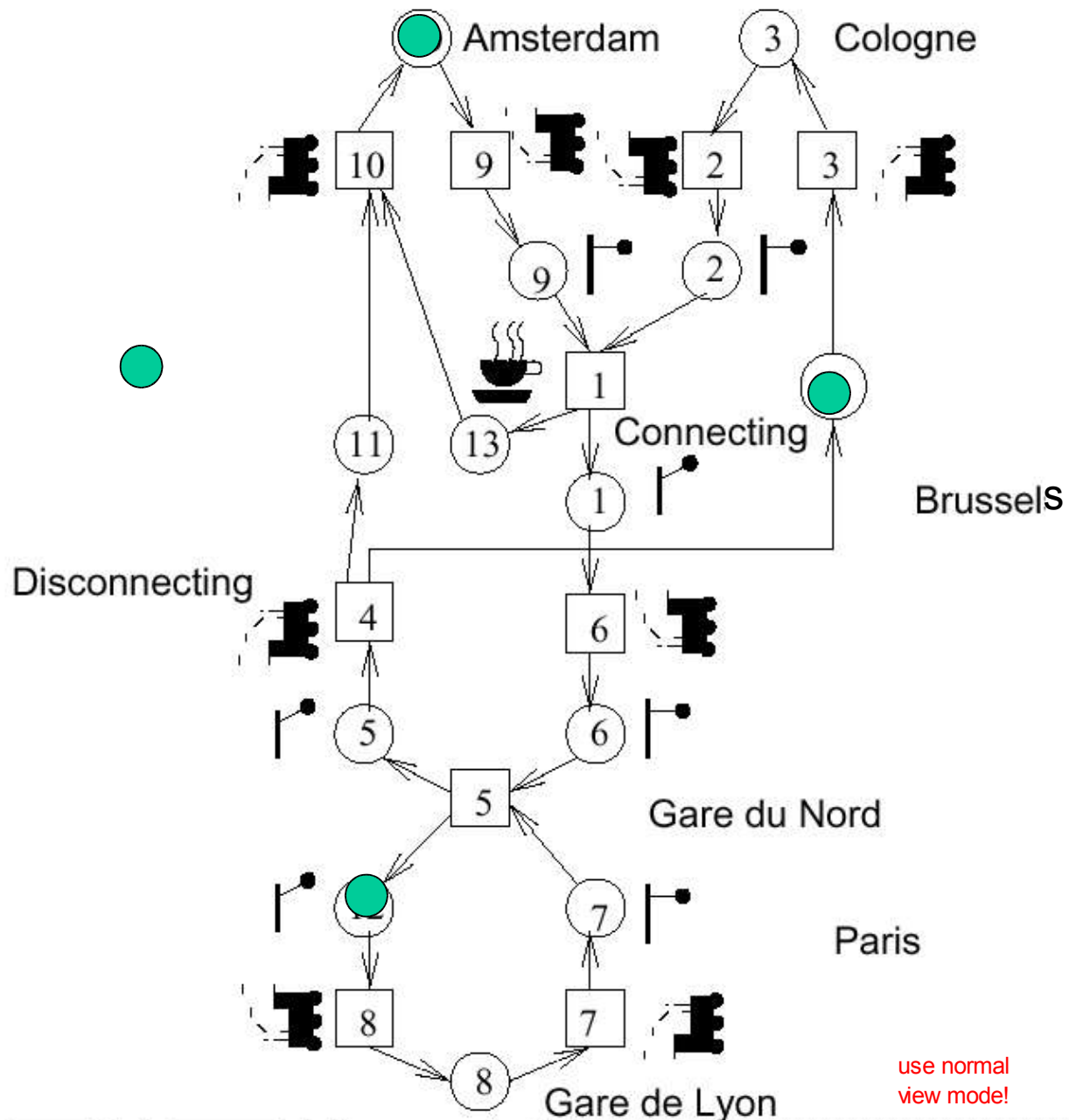
Thalys trains between
Cologne, Amsterdam,
Brussels and Paris.



[<http://www.thalys.com/be/en>]

More complex example (2)

Slightly simplified:
Synchronization at
Brussels and
Paris,
using stations
“Gare du Nord”
and “Gare de
Lyon” at Paris



use normal
view mode!

Evaluation

Pros:

- Appropriate for distributed applications,
- Well-known theory for formally proving properties,
- Initially a quite bizarre topic, but now accepted due to increasing number of distributed applications.

Cons (for the nets presented) :

- no programming elements,
- no hierarchy,
- problems with modeling timing,

Extensions:

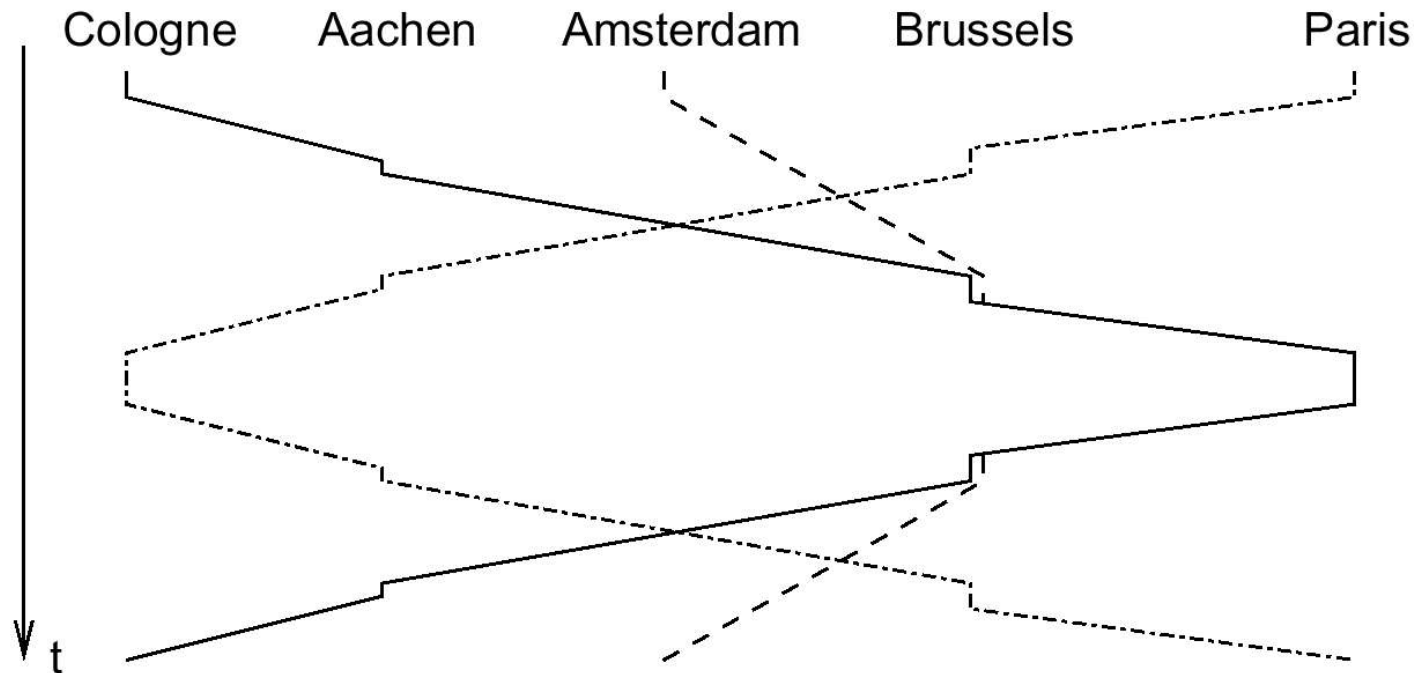
- Enormous amounts of efforts on removing limitations.

How to model time?

[back to full
screen mode](#)

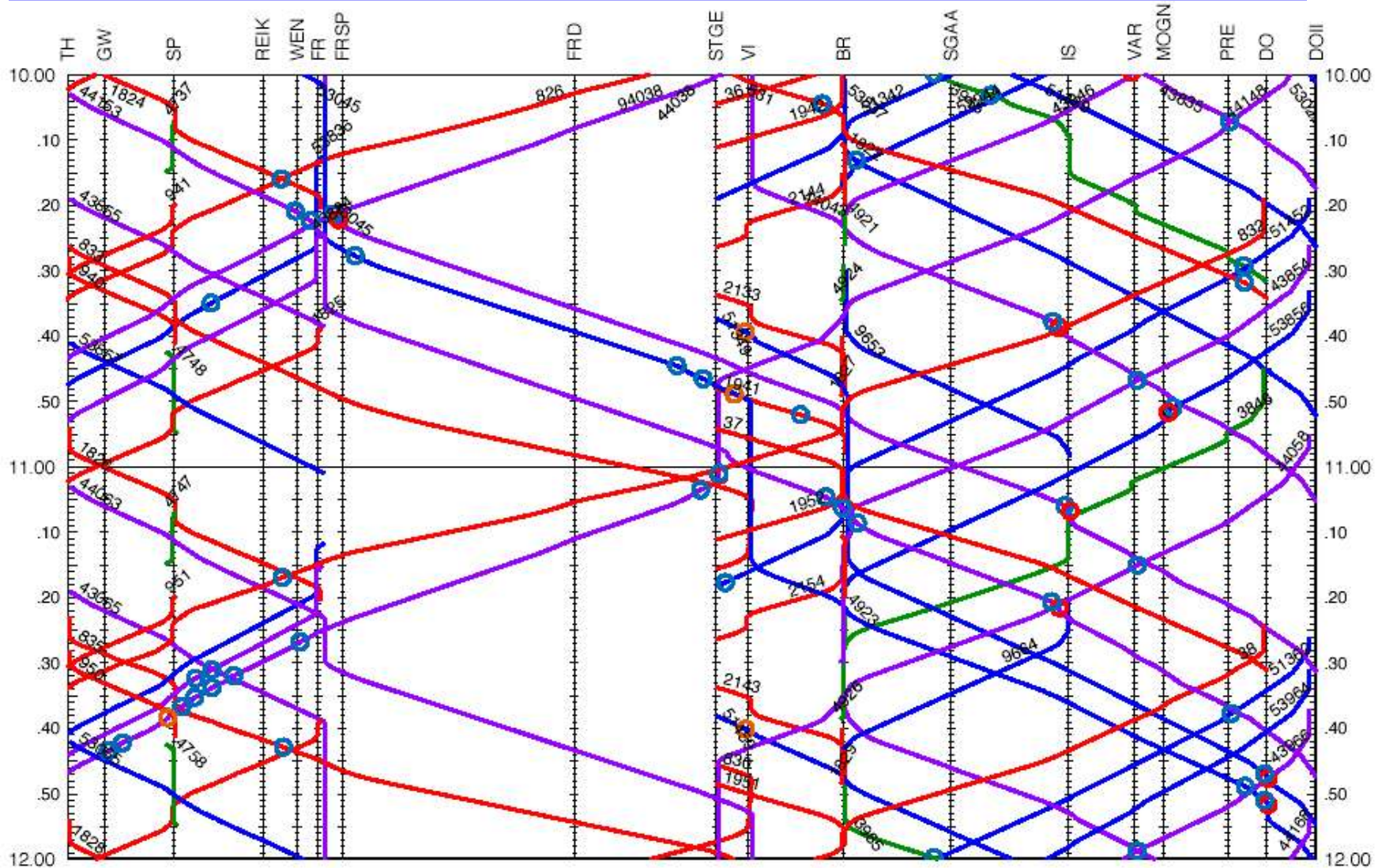
Message sequence charts (MSC)

Graphical means for representing schedules; **time** used vertically, geographical distribution horizontally.



No distinction between accidental overlap and synchronization

Time/distance diagrams as a special case



Evaluation of MSCs

Pro:

- Easy to use by different sets of people

Con:

- Precise specification of semantics?
- Extensions required to distinguish between may and must.

Typically combined with programming languages to represent actions

UML for real-time?

MSCs integrated into unified modeling language (UML).

UML: set of graphical notations for early design phases.

General applications in software design.

Initially not designed for real-time.

Lacking features (1998):

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

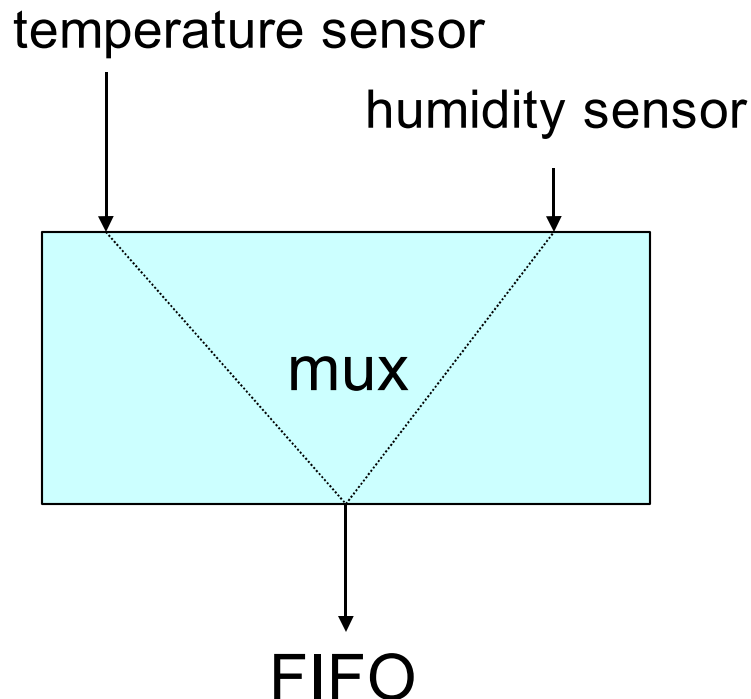
Projects on defining real-time UML based on previous work

- ROOM [Selic] is an object-oriented methodology
- „UML profile for schedulability, performance and time“
<http://www.rational.com/uml/resources/documentation>
- ...

Process networks

Many applications can be specified in the form of a set of communicating processes.

Example: system with two sensors:



Alternating read

loop

read_temp; read_humi

until false;

of the two sensors no the
right approach.

The case for multi-process modeling in imperative languages

```
MODULE main;
  TYPE some_channel =
    (temperature, humidity);
    some_sample : RECORD
      value : integer;
      line : some_channel
    END;
  PROCESS get_temperature;
  VAR sample : some_sample;
  BEGIN
    LOOP
      sample.value := new_temperature;
      IF sample.value > 30 THEN ....
      sample.line := temperature;
      to_fifo(sample);
    END
  END get_temperature;
```

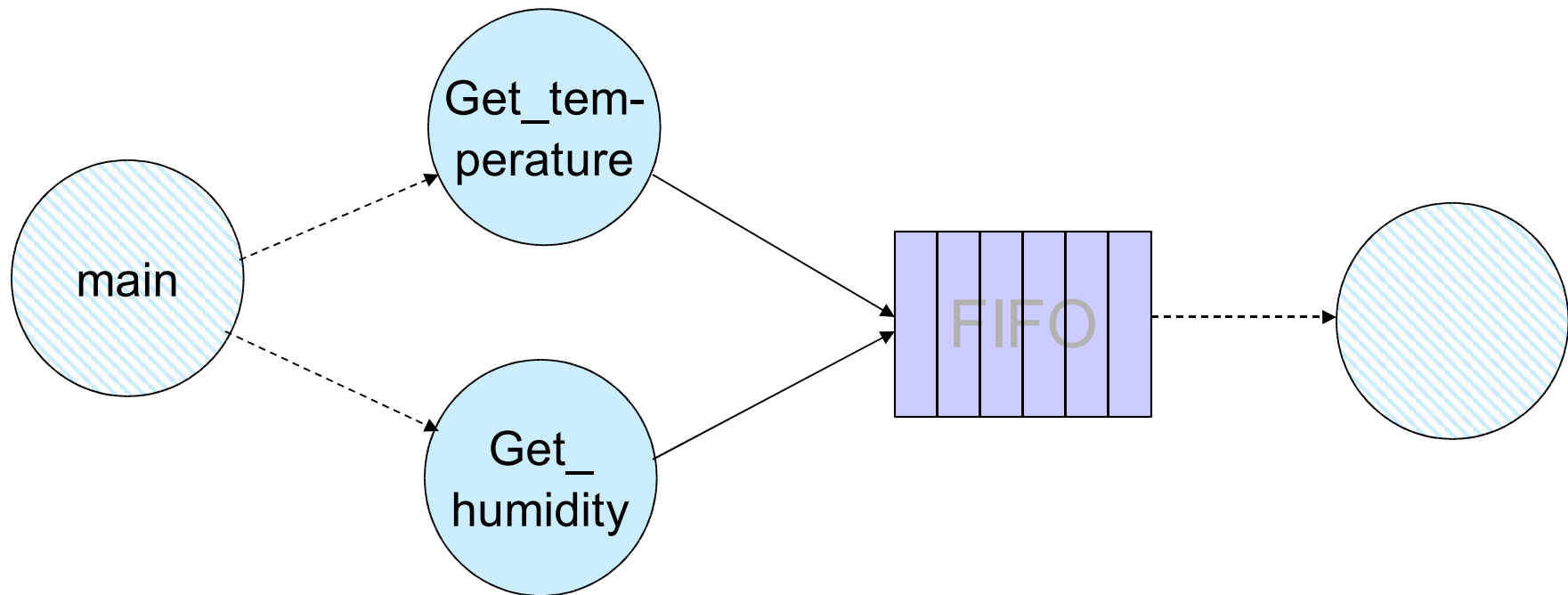
```
PROCESS get_humidity;
  VAR sample : some_sample;
  BEGIN
    LOOP
      sample.value := new_humidity;
      sample.line := humidity;
      to_fifo(sample);
    END
  END get_humidity;

BEGIN
  get_temperature; get_humidity;
END;
```

- Blocking calls new_temperature, new_humidity
- **Structure clearer than alternating checks for new values in a single process**

How to model dependencies between tasks/processes?

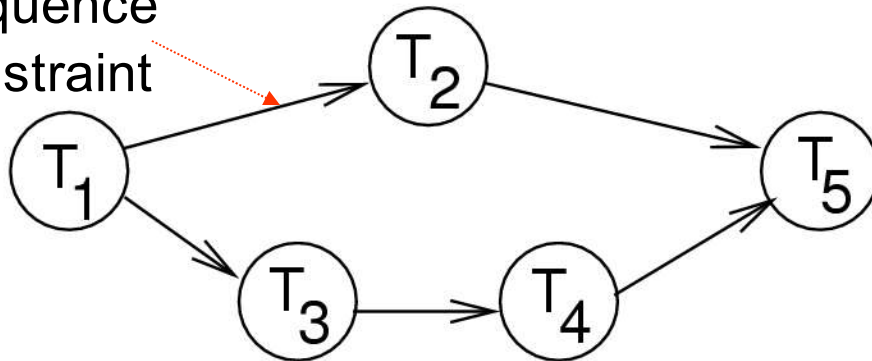
Dependencies between processes/tasks



☞ General discussion
of process networks

Task graphs

Sequence
constraint



Nodes are assumed to be a „program“ described in some programming language, e.g. C or Java.

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

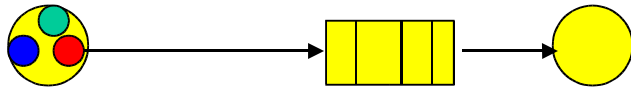
If $(v1, v2) \in E$, then $v1$ is called an **immediate predecessor** of $v2$ and $v2$ is called an **immediate successor** of $v1$.

Suppose E^* is the transitive closure of E .

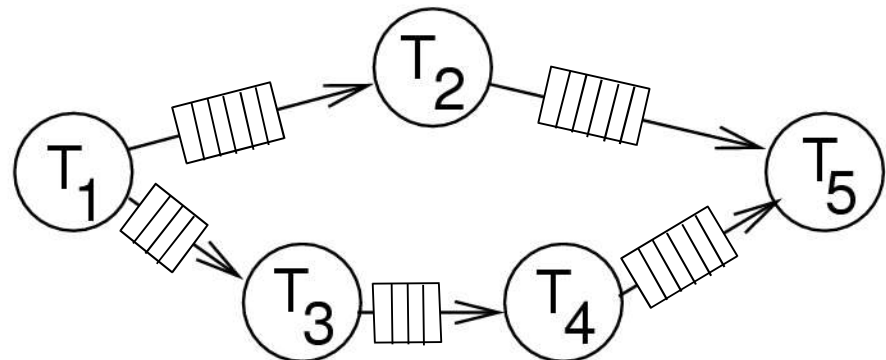
If $(v1, v2) \in E^*$, then $v1$ is called a **predecessor** of $v2$ and $v2$ is called a **successor** of $v1$.

Asynchronous message passing: Kahn process networks

For asynchronous message passing: communication between tasks is buffered



Special case: Kahn process networks: executable task graphs; Communication is assumed to be via infinitely large FIFOs



Properties of Kahn process networks

- Each node corresponds to one program/task;
- communication is only via channels;
- channels include FIFOs as large as needed;
- one producer and one consumer;
- channels transmit information within an unpredictable but finite amount of time;
- each node may be either *waiting* for input on **one** of its input channels or *executing*;
- mapping from ≥ 1 input seq. to ≥ 1 output sequence;
- **in general, execution times are unknown**;
- send operations are non-blocking, reads are blocking.

Example

☞ Model of parallel computations used in practice (e.g. at Philips).

```

Process f (in int u, in int v, out int w) {
  int i; bool b = true;
  for (;;) {
    i = b ? wait (u) : wait (v); // wait returns next token in FIFO, blocks if empty
    printf("%i\n", i);
    send (i, w); // writes a token into a FIFO w/o blocking
    b = !b;
  }
}

```

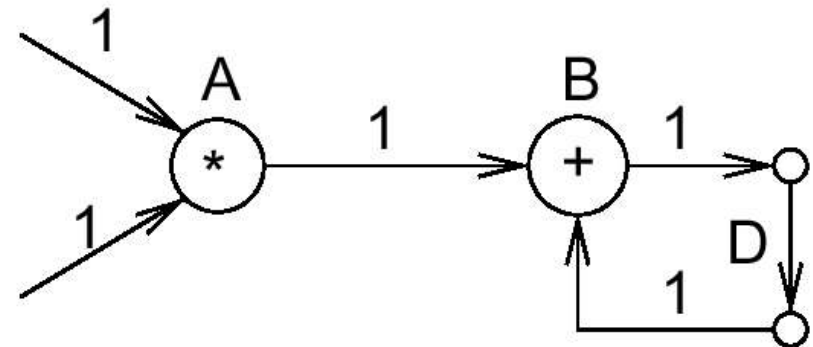
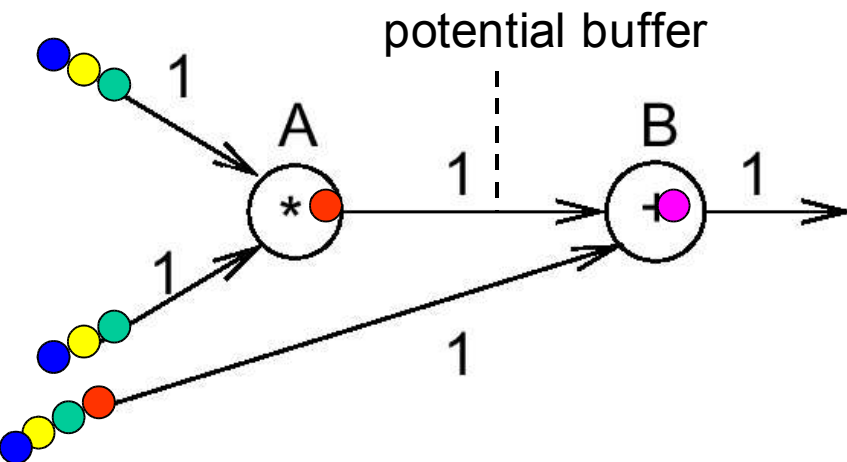
- ❖ A process can not check whether data is available before attempting a read
- ❖ A process can not wait for data on more than one port at a time
- ❖ Therefore, order of reads, writes depend only on data, not its arrival time
- ❖ It is a challenge to schedule KN without accumulating tokens.

☞ easier for SDF

Asynchronous message passing: Synchronous data flow (SDF)

Asynchronous message passing=
tasks do not have to wait until output is accepted.

Synchronous data flow =
all tokens are consumed at the same time.

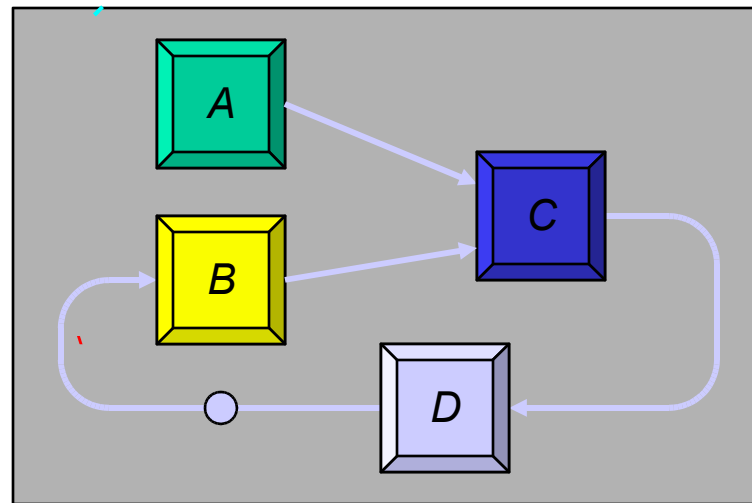


SDF model allows static scheduling of token production and consumption.

In the general case, **buffers may be needed** at edges.

Parallel Scheduling of SDF Models

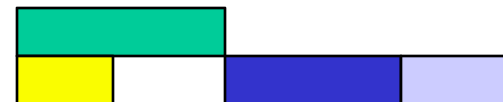
SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.



Many scheduling optimization problems can be formulated. Some can be solved, too!



Sequential



Parallel

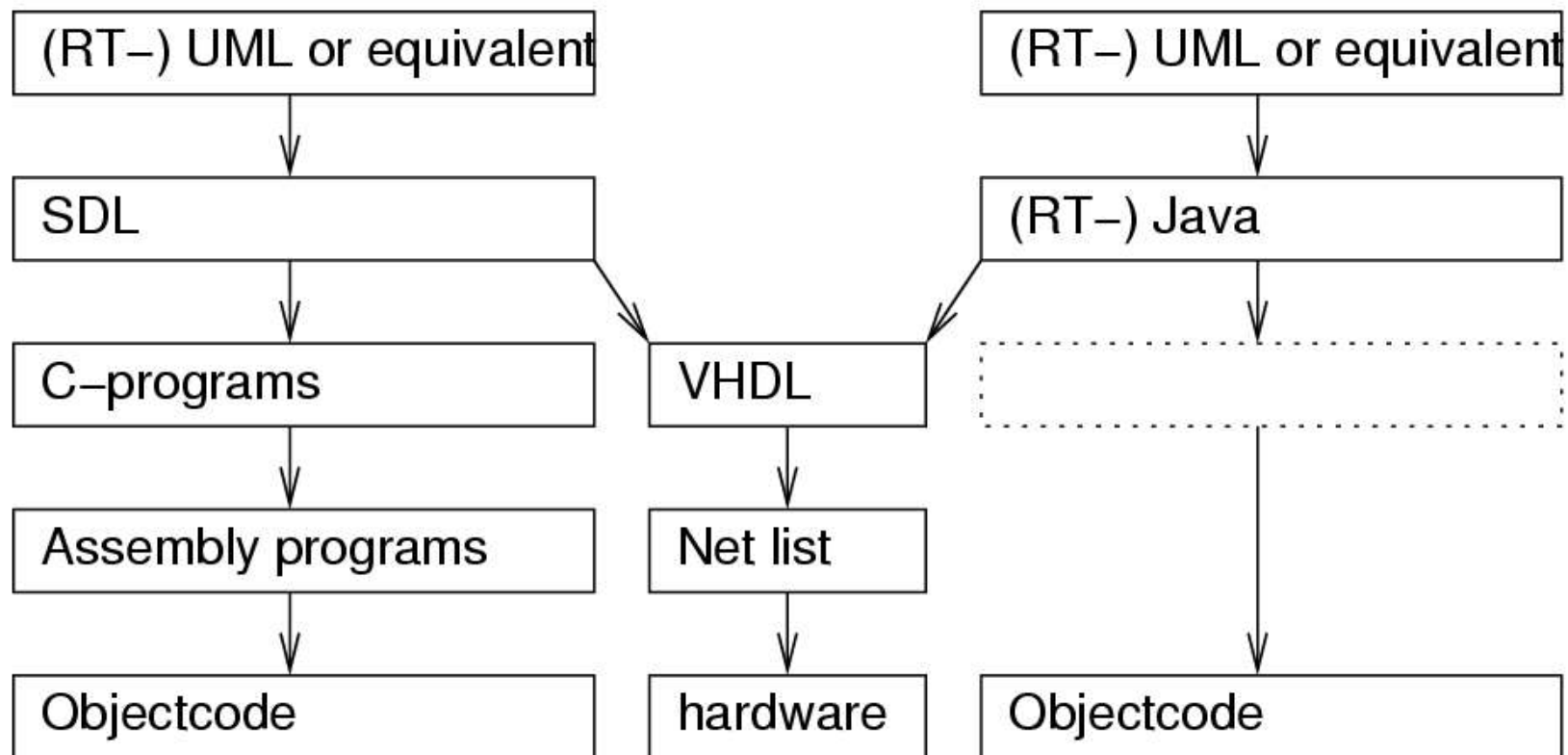
Source: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt

Additional Languages

- MATLAB/SIMULINK
- C, C++
- Java
- CSP (communicating sequential processes)
- SystemC (C++ extended by library for hardware modeling)
- Verilog (hardware description language)
- SystemVerilog (extended hardware description lang.)
- VHDL (hardware description language)
- SDL (communicating FSMs)
- Algebraic languages
- Esterelle

How to cope with language problems in practice?

Mixed approaches:



Summary

- Requirements for specification languages for embedded systems
- Models of computation
- Languages for non-distributed systems
 - StateCharts
- Languages for distributed systems
 - Petri nets
 - MSCs, link to UML for real-time applications
- Task graphs
 - Kahn process networks
 - Synchronous dataflow models
- Specific languages: MATLAB/Simulink, SystemC, Verilog,
- Language comparison, compromises