Embedded Systems 2

Embedded Systems

- Lectures:
  - Tuesday 10:15 -11:45
  - Thursday 14:15 -15:45

Midterm exam, Thursday December 16, 2010, 16-19
End-of-term exam, Monday February 14, 2011, 14-17
End-of-semester exam: tba

Embedded Systems

Registration through **HISPOS** (if HISPOS is not applicable – Non-CS, Erasmus, etc – send email to peter@cs.uni-saarland.de)

- Webpage

http://react.cs.uni-sb.de/teaching/embedded-systems-10-11

The course mailing list is available now. Subscribe to get notifications and the latest information: https://alan.cs.uni-saarland.de/cgi-bin/mailman/listinfo/es

- Tutorials

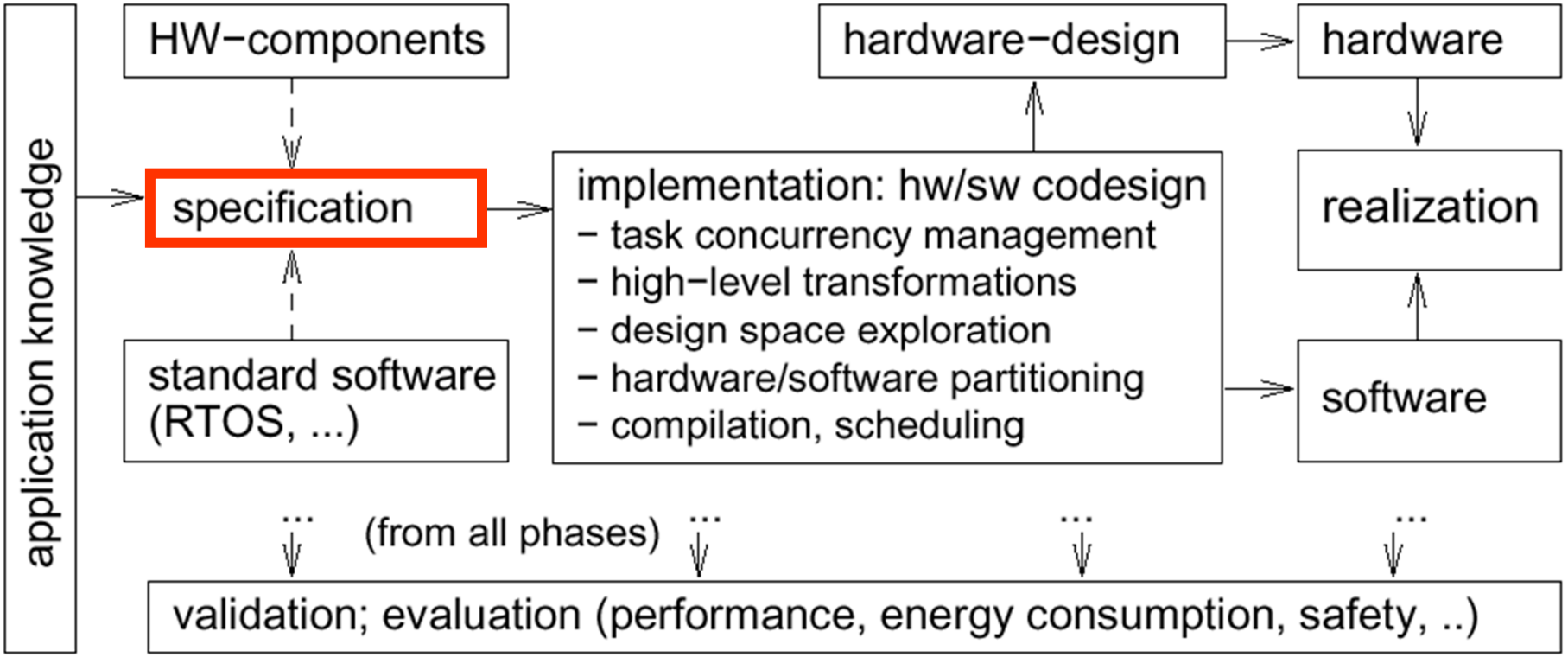
  Wednesday, 16:00-18:00, SR 107, E 1 3
  Friday, 12:00-14:00, SR 015, E 1 3 **(START: Wed. November 3rd)**
  Friday, 14:00-16:00, SR 016, E 1 3

Please indicate tutorial, matr nr, name, e-mail on **homework submissions.**

Specification – Models of Computation (MOC)

Specifications

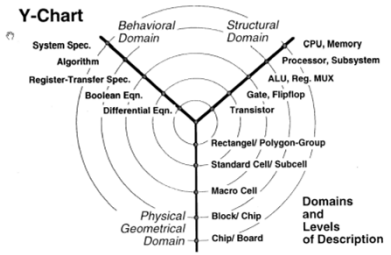
REVIEW



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

REVIEW

- **Hierarchy**
  Humans not capable to understand systems containing more than a few objects.
  Most actual systems require far more objects.
  two kinds of hierarchy are used:
  - Behavioral hierarchy
    Examples: states, processes, procedures.
  - Structural hierarchy
    Examples: multipliers, FPUs, processors, printed circuit boards



- **Timing behavior**
- **State-oriented behavior**
  suitable for reactive systems

Requirements for specification techniques (2)

- **Event-handling** (external or internal events)
- **No obstacles for efficient implementation**
- **Support for the design of dependable systems**
  Unambiguous semantics, ...
- **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state.

Requirements for specification techniques (3)

- **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**
- **Support for the design of large systems**
- **Domain-specific support**

Requirements for specification techniques (4)

- **Readability**
- **Portability and flexibility**
- **Non-functional properties**
  fault-tolerance, availability, EMC-properties, weight, size, user friendliness, extendibility, expected life time, power consumption...
- **Adequate model of computation**

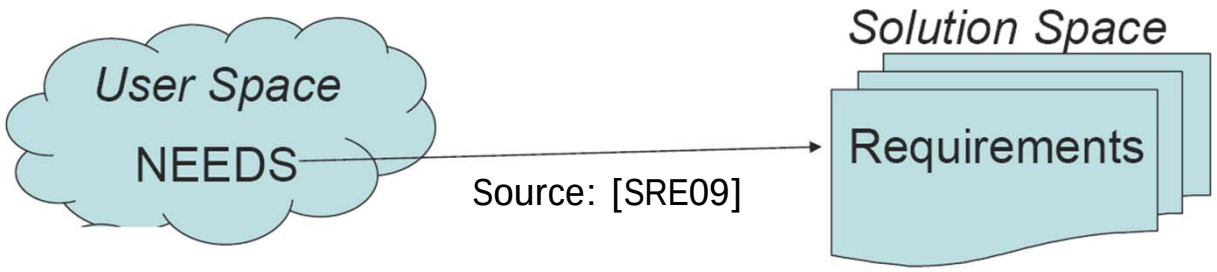
- → **we have to live with compromises**

What is a requirement ?

- Describes **what** the system should do but not how to implement it

- IEEE 1012 standard
  (IEEE= Institute of Electrical and Electronics Engineers)

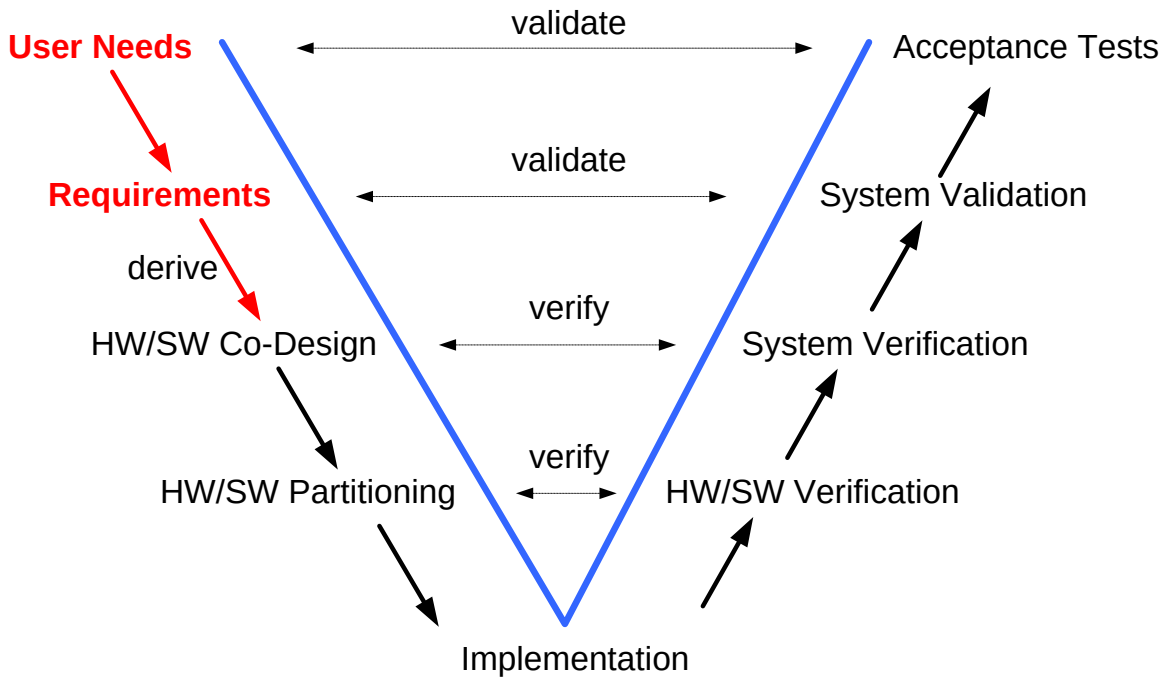
  - A condition or capability of the system needed by a user to solve a problem or achieve an objective
  - A condition or capability that must be met or possessed by a system… to satisfy a contract standard, specification, or other formally imposed document



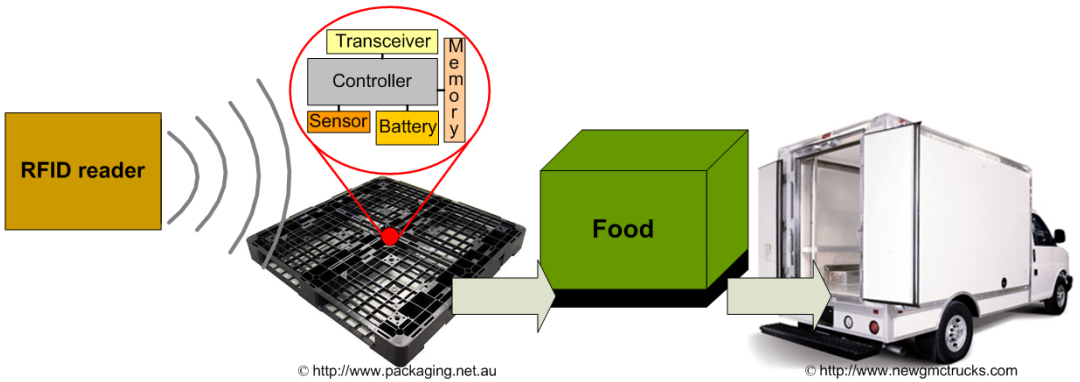
- Ranges from a high-level abstract statement of a service (function, feature) or of a system constraint to a detailed mathematical functional specification

Introduction (1)

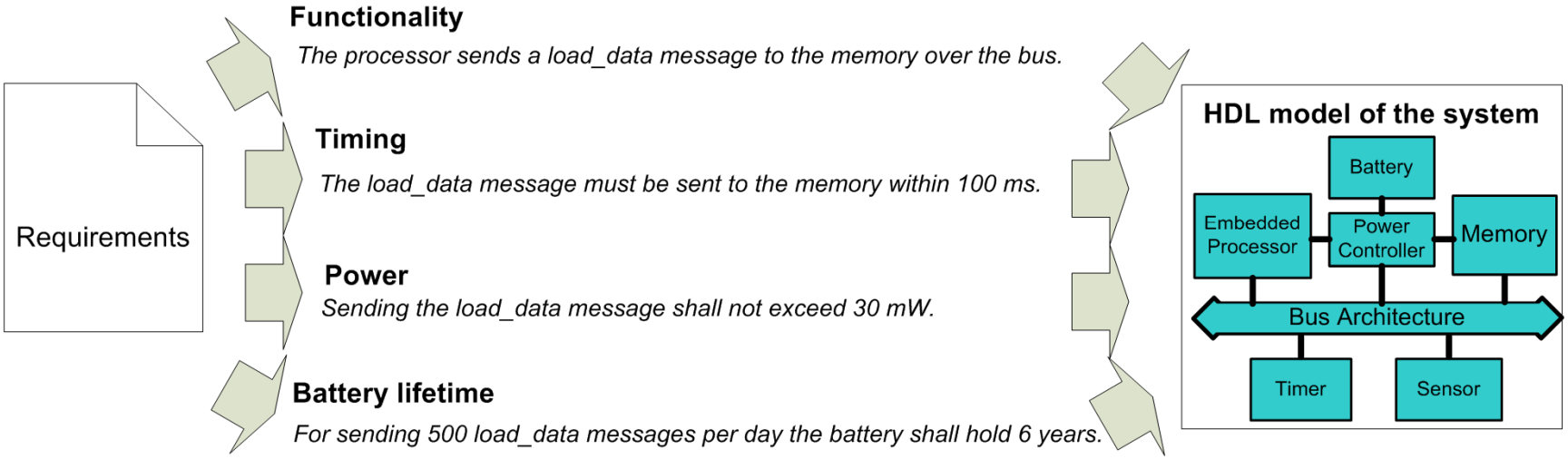
Design Flow (V-Model)



Introduction (2)



- Requirements-Driven Design



Introduction (3)

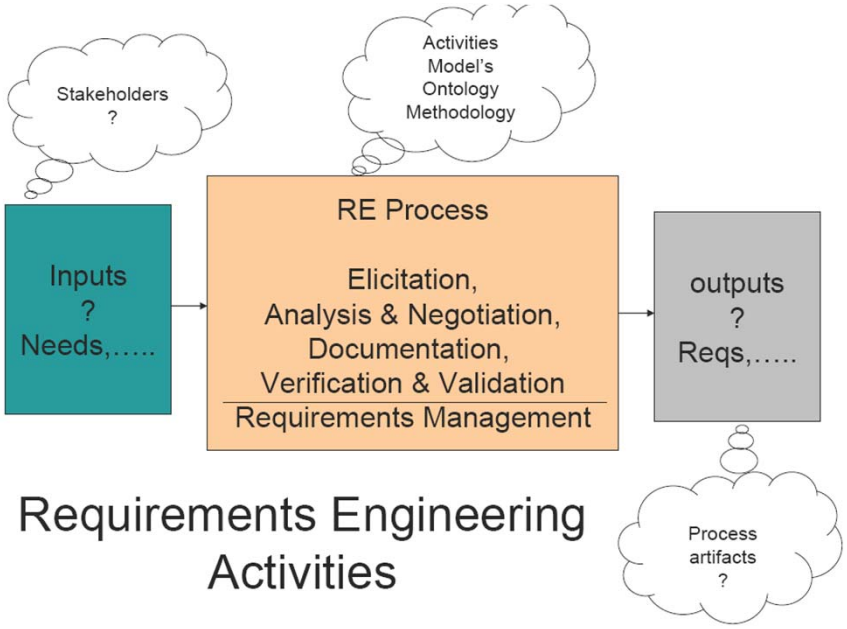
- Design has to fulfil the given requirements
- Rising chip complexity results in x1000 requirements
- Requirements must be specified and managed carefully
- Requirements must be tightly integrated (linked) with the design
- Late re-designs due to incorrect requirements are costly

F. Reuning, Design Engineer at Conti Temic (Siemens VDO) "*We have large problems to verify our requirements comprehensively in a standardized way*"

C. Nippert, Design Automation, Infineon Munich: *"Our long-term goal is to integrate requirements and constraints smoothly into the design flow to verify them, but we are far away from that!"*

Requirements engineering/management

- Requirements Engineering (RE)
  - Elicitation/fetch & Modelling: The process of specifying the services (functions) that the customer requires from a system and the constraints under which it operates and is developed (=requirements)
  - Validation & Analysis: Ensures that requirements are complete, consistent, and relevant



Requirements Engineering Activities

Source: [SRE09]

Examples of functional requirements

- When the Memory receives a READ request it shall transmit the data at the given address to the controller.
- When the Memory receives a WRITE request it shall store the data to the given address.
- When the Memory receives a READ request it shall transmit the data at the given address to the controller within 55,70ns.

Non-functional requirements

- Product requirements
  - Requirements which specify that the delivered product must have certain qualities e.g. execution speed, reliability, etc.
    - *8.1 The memory shall have an equal cycle time of 55,70ns.*
    - *8.2 The operational voltage of the memory shall be between 4,5V and 5,5V.*

- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc
    - *9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.*

- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. safety, interoperability requirements, legislative requirements, etc

Verification vs. validation

- Verification:

  "Are we building the product right"

  The systems should conform to its specification

- Validation:

  "Are we building the right product"

  The system should do what the user really requires

Modeling

- Abstract view of a design
  - Representation of reality in each design step
    - Apply analysis, synthesis and verification techniques
- Core of automated design flow
  - Varying levels of abstraction
    - Level & organization of detail
  - Well-defined and unambiguous semantics
    - Objects, composition rules and transformations

- Models of behavior (Models of Computation)
  - Concurrent computation
  - Communication
- Models of structure
  - Processing, storage and communication elements (PEs and CEs)
  - Networks of busses

Models of Computation (MoCs) - Basic Concepts

- MoC is composed of a description mechanism (syntax) and rules for computation of behavior given the syntax (semantics)

- It is chosen for its suitability: compactness, ability to synthesize, optimize the behavior of implementation

- Most MoCs permit distributed system of description (a collection of communicating modules), and gives rules of computation of each module (function), and how they communicate.

Models of computation

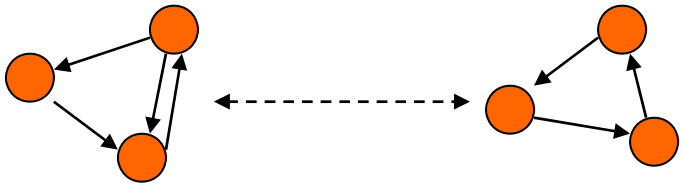
- **Models of computation define** [Lee, UCB, 1999]:
  - How computations of several components proceed.
  - What does it mean to be a **component**: Subroutine? Process? Thread?
  - The mechanisms by which components **interact**: Message passing? *Rendez-vous*?
  - **What components know** about each other (global variables? Implicit behavior of other components)

Programming Models

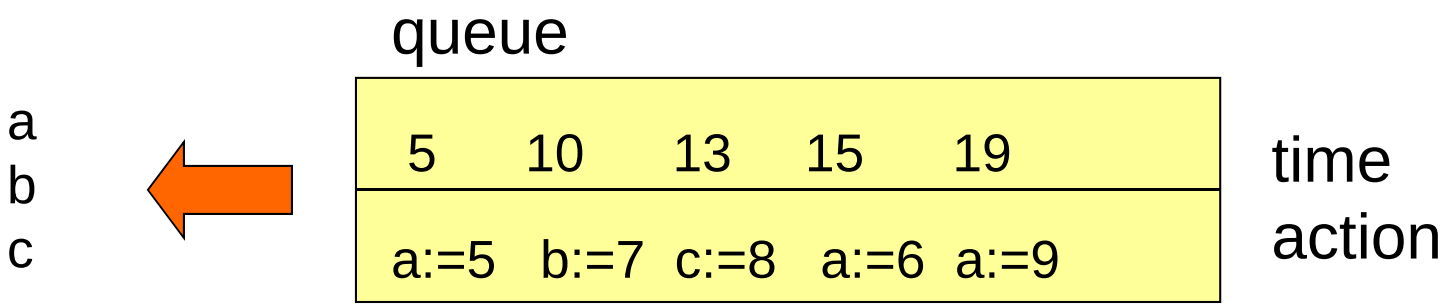
- Imperative programming models
  - Ordered sequence of statements that manipulate program state
  - Sequential programming languages [C, C++, …]
- Declarative programming models
  - Dataflow based on explicit dependencies (causality)
  - Functional or logical programming languages [Prolog]
- Synchronous programming models
  - Reactive vs. transformative: explicit concurrency
  - Lock-step operation of concurrent statement blocks
  - Synchronous languages [Esterel (imperative), Lustre (declarative)]

Models of computation
- Examples (1) -

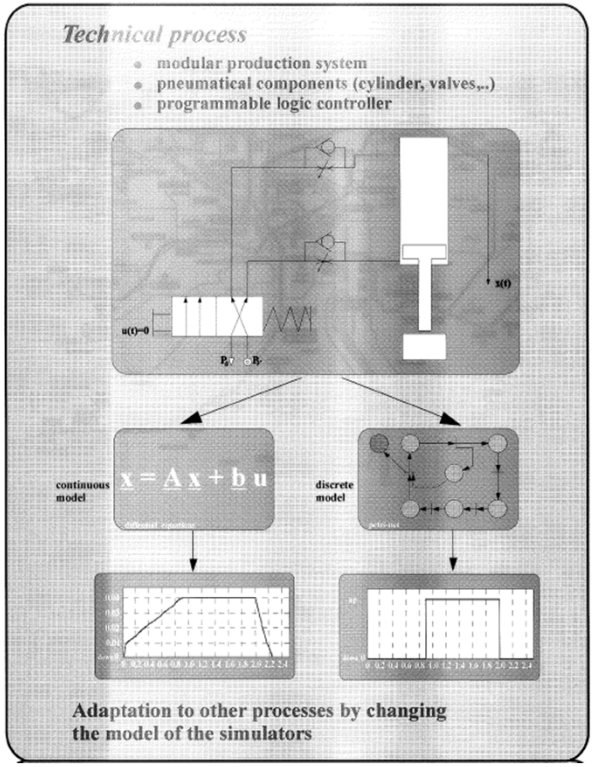
- Communicating finite state machines (CFSMs):



- Discrete event model

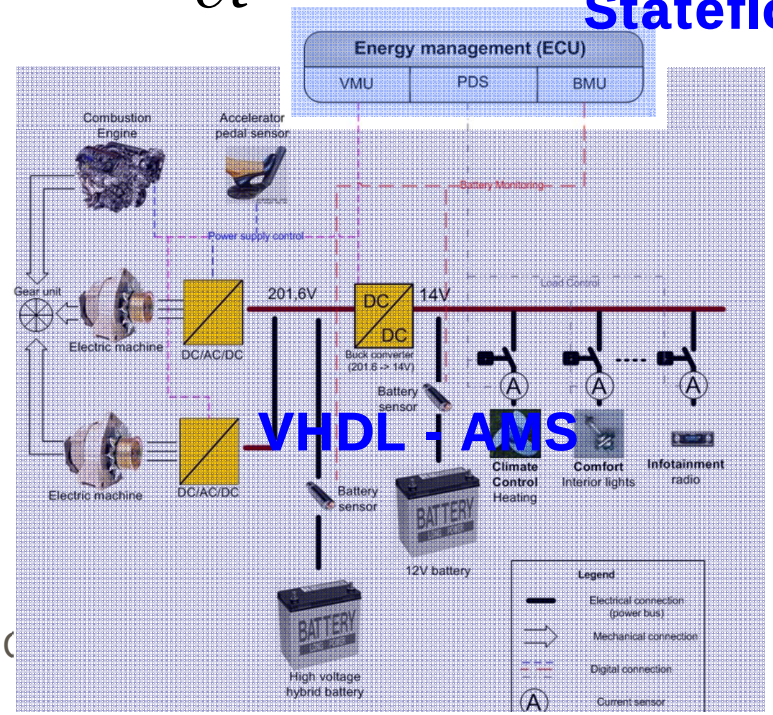


Models of computation
- Examples (2) -



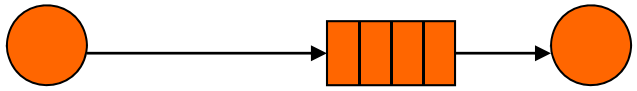
- Differential equations

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



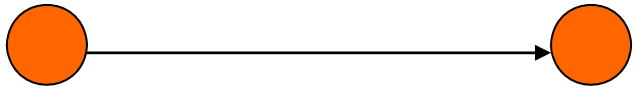
- Needed for hybrid systems with analog part

Models of communication



- Asynchronous message passing

deliver a message from sender to receiver, without waiting for the receiver to be ready, sender and receiver can overlap their computation, buffer full ? (block the sender, discard future messages)



- Synchronous message passing

sender will not continue until the receiver has received the message

StateCharts

- StateCharts = the only unused combination of
  „flow“ or „state“ with „diagram“ or „charts“
- Based on classical automata (FSM):
  StateCharts = FSMs + Hierarchy + Orthogonality + Broadcast communication
- Industry standard for modelling automotive applications
- Appear in UML (Unified Modeling Language), Stateflow, Statemate, …
- *Warning: Syntax and Semantics may vary.*

- **Start with brief review on Finite State Machines.**

Mealy automaton

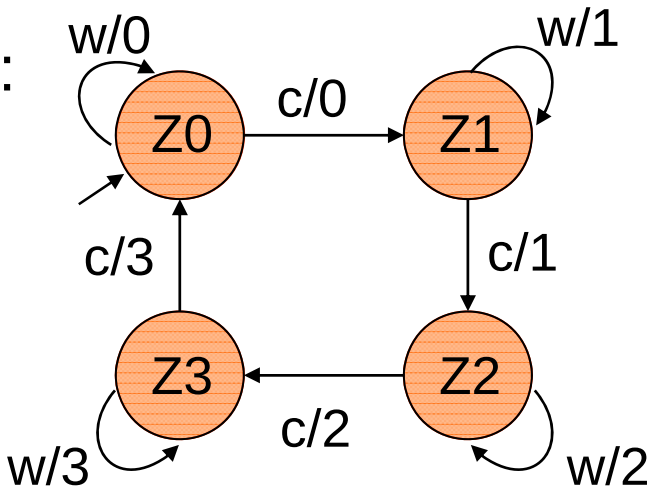
- **Definition:**

  $M=(I, O, S, s_0, \delta, \lambda)$ is a Mealy automaton iff

  - I is a finite, non-empty set (input symbols),
  - O is a finite, non-empty set (output symbols),
  - S is a finite, non-empty set (states),
  - s_0 … initial state,
  - $\delta : S \times I \rightarrow S$ (transition function),
  - $\lambda : S \times I \rightarrow O$ (output function).

- Example for representation:

  - the output depends on the current state and the current input symbol,
  - the next state depends on the current state and the current input symbol

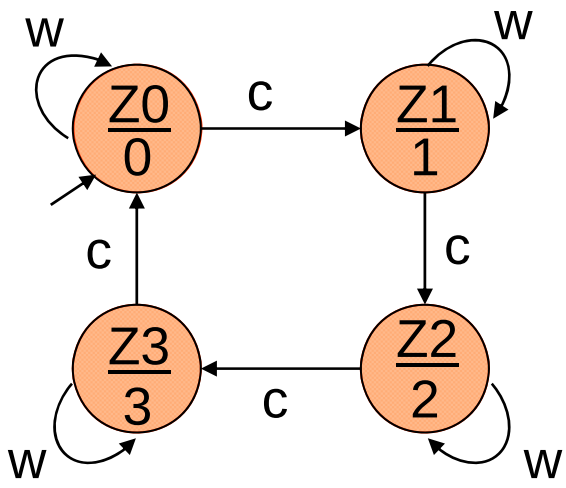


Moore automaton

- **Definition:**

  $M=(I, O, S, s_0, \delta, \lambda)$ is a Moore automaton iff
  - I is a finite, non-empty set (input symbols),
  - O is a finite, non-empty set (output symbols),
  - S is a finite, non-empty set (states),
  - s_0 …initial state,
  - $\delta : S \times I \rightarrow S$ (transition function),
  - $\lambda : S \rightarrow O$ (output function).
- Example for representation:

  - the output function does not depend on the current input symbol



Mealy-Moore FSMs

- The set of states define the state space
- State space are flat
  - All states are at the same level of abstraction
  - All state names are unique
- State models are single threaded
  - Only a single state can be valid at any time

Mealy-Moore FSMs

- Moore state models: all actions are upon state entry
  - Non-reactive (response delayed by a cycle)
  - Good for implementation

- Mealy state models: all actions are in transitions
  - Reactive (0 response time)

Models of an elevator controller

"If the elevator is stationary and the floor requested is equal to the current floor, then the elevator remains idle.

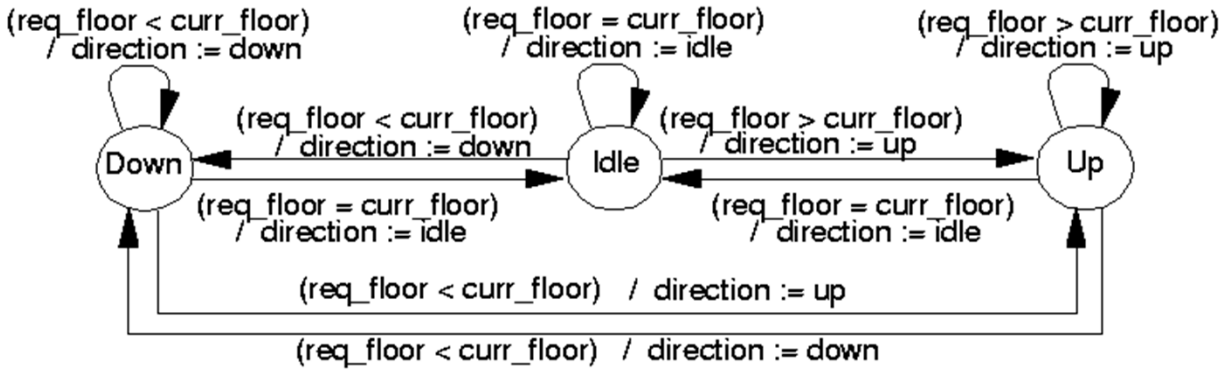
If the elevator is stationary and the floor requested is less than the current floor, then lower the elevator to the requested floor.

If the elevator is stationary and the floor requested is greater than the current floor, then raise the elevator to the requested floor."

(a) English description

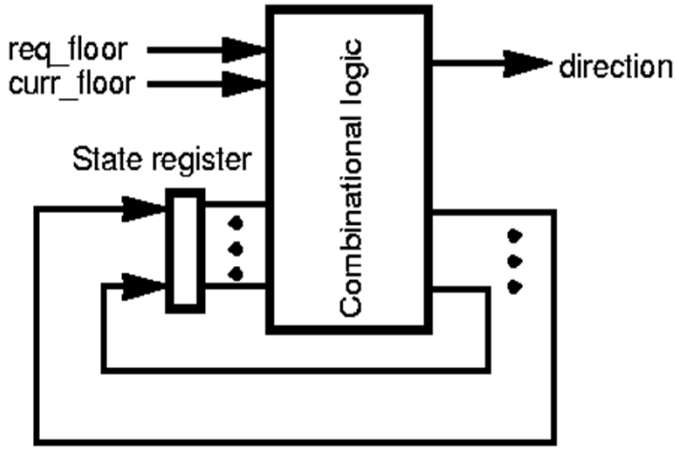
```
loop
    if (req_floor = curr_floor) then
      direction := idle;
    elsif (req_floor < curr_floor) then
      direction := down;
    elsif  (req_floor > curr_floor) then
      direction := up;
    end if;
end loop;
```

(b) Algorithmic model

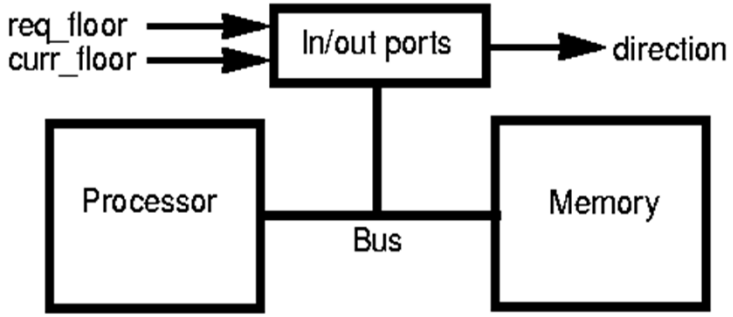


(c) State-machine model

Architectures for implementing the elevator controller



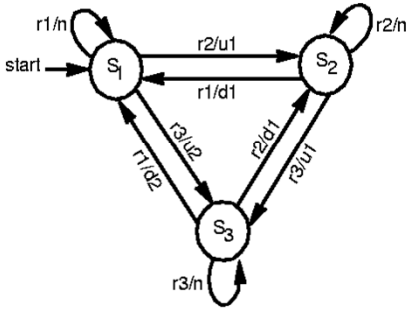
(a) Register level



(b) System level

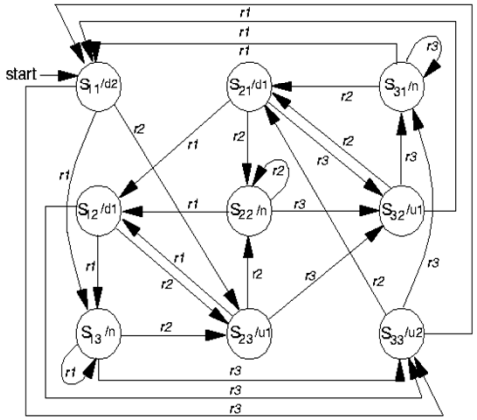
Moore and Mealy automata can be transformed into each other

State oriented: Finite-state machine (Mealy model)



S = { s1, s2, s3}
I = {r1, r2, r3}
O = {d2, d1, n, u1, u2}
f: S x I -> S
h: S x I -> O

State oriented: Finite-state machine (Moore model)



StateCharts

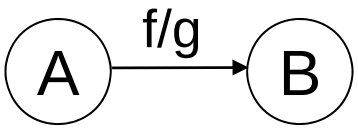
- Statecharts introduced in
  Harel: "StateCharts: A visual formalism for complex systems". Science of Computer Programming, 1987.
- More detailed in
  Drusinsky and Harel: "Using statecharts for hardware desription and synthesis", IEEE Trans. On Computer Design, 1989.
- Formal semantics in
  Harel, Naamad: "The statemate semantics of statecharts", ACM Trans. Soft. Eng. Methods, 1996.

StateCharts – Additional features compared to classical deterministic automata

- Non-determinism
- Hierarchy
- Variables with complex data types
- Concurrency
- Transitions with conditions
- Different I/O: transitions can
  - Be active depending on the presence of **events**,
  - “produce events”,
  - change variables
- Timers used to produce “timeout events”.

Non-deterministic transitions

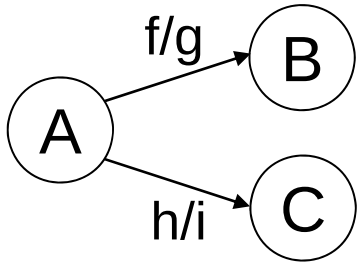
Edge label (simple version):



- Transition from A to B iff event f is present.
- Effect of transition from A to B: Event g is produced.
- Events may be
  - Present or
  - Not present
- Events may be
  - External events (provided by the *environment*)
  - Internal events (produced by internal transitions)
- Produced events exist only for one step.

Non-deterministic transitions

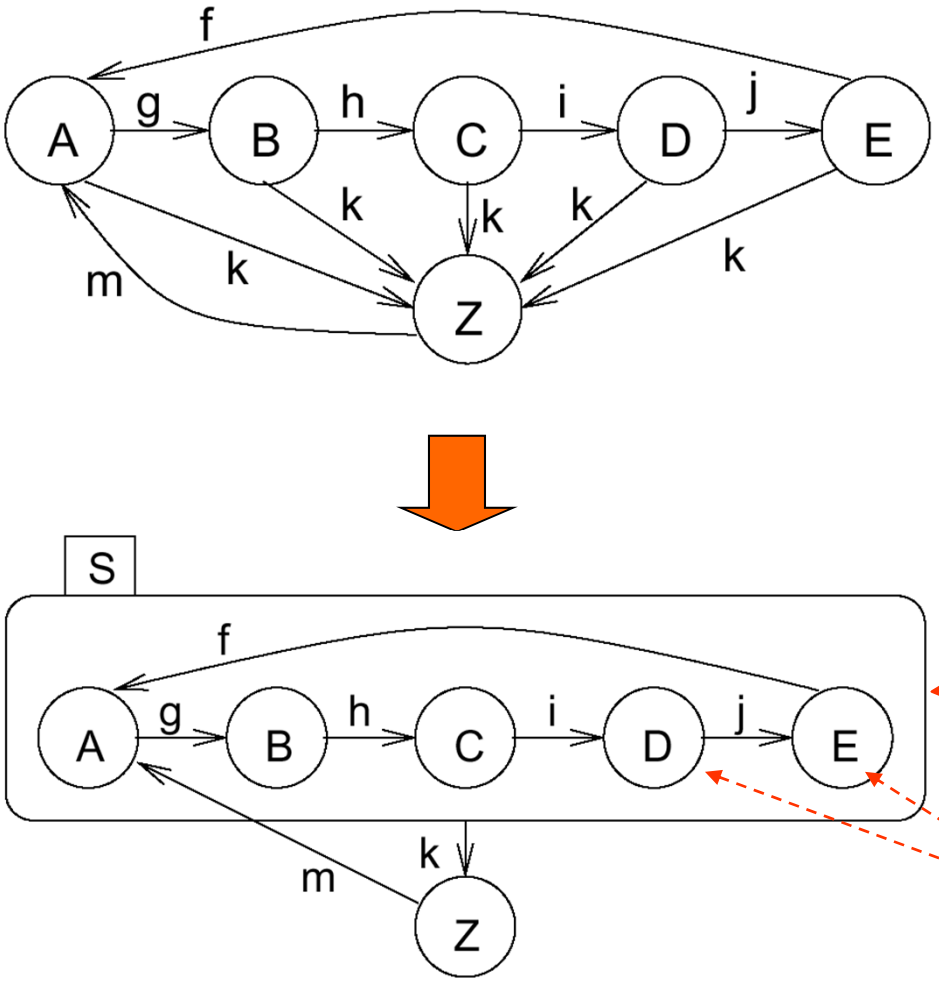
- Non-determinism:



- Events f and h may be present **at the same time**.

  Non-deterministic transitions,
  different behaviours are possible

Introducing hierarchy



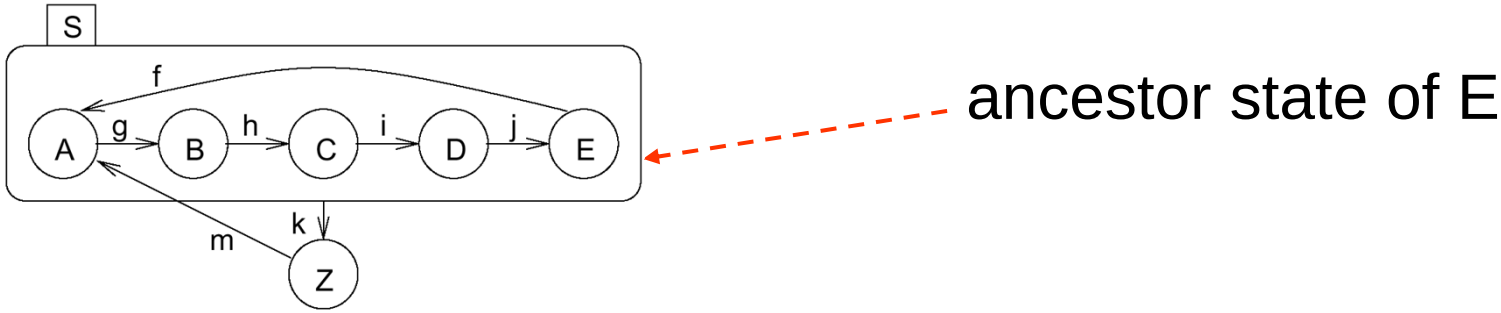
FSM will be **in** exactly one of the substates of S if S is **active** (either in A or in B or ..)

superstate

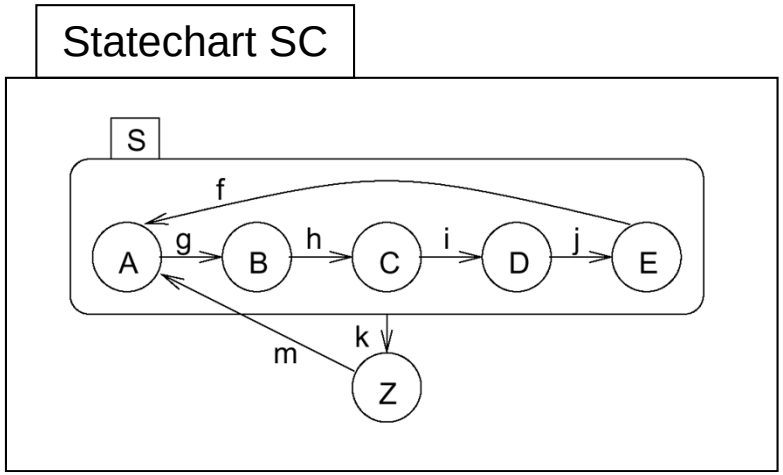
substates

Definitions

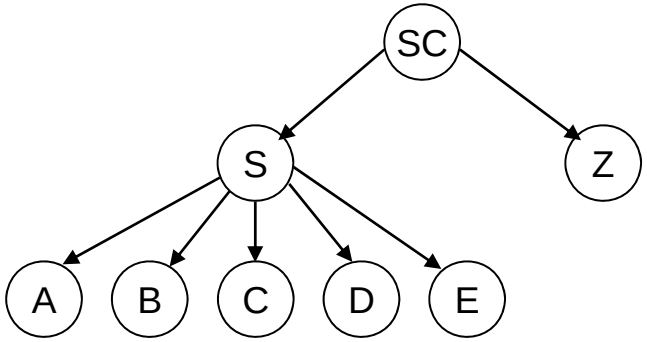
- Current states of FSMs are also called **active** states.
- States which are not composed of other states are called **basic states.**
- States containing other states are called **super-states**.
- For each basic state s, the super-states containing s are called **ancestor states**.
- Super-states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



Hierarchy



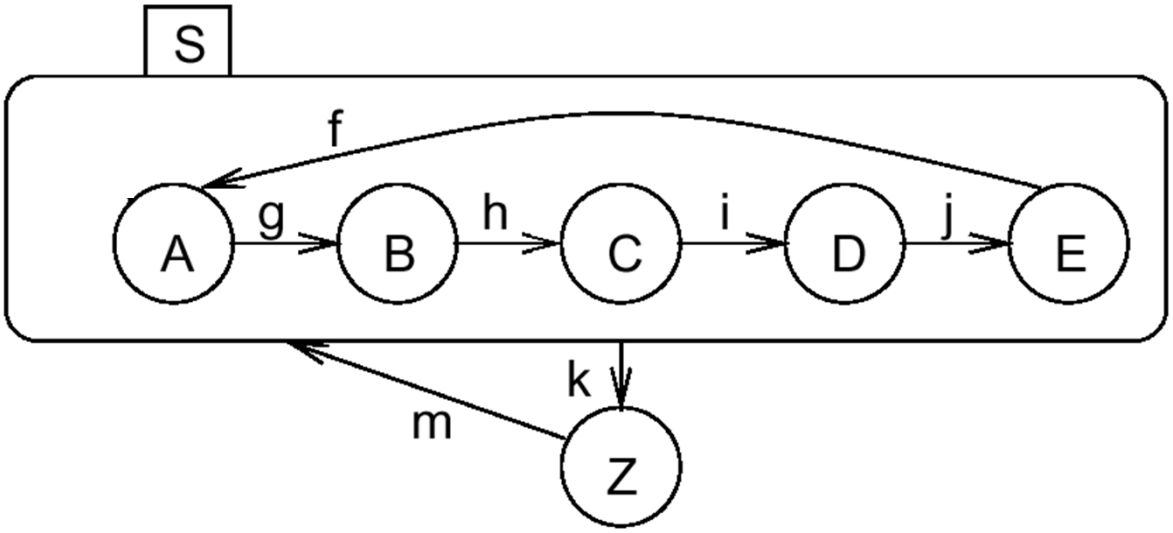
- Hierachy information may be represented by a hierarchy tree with basic states as leaves.



• Transitions between all levels of hierarchy possible!

• When a basic state is active, then all its ancestor states are active, too.

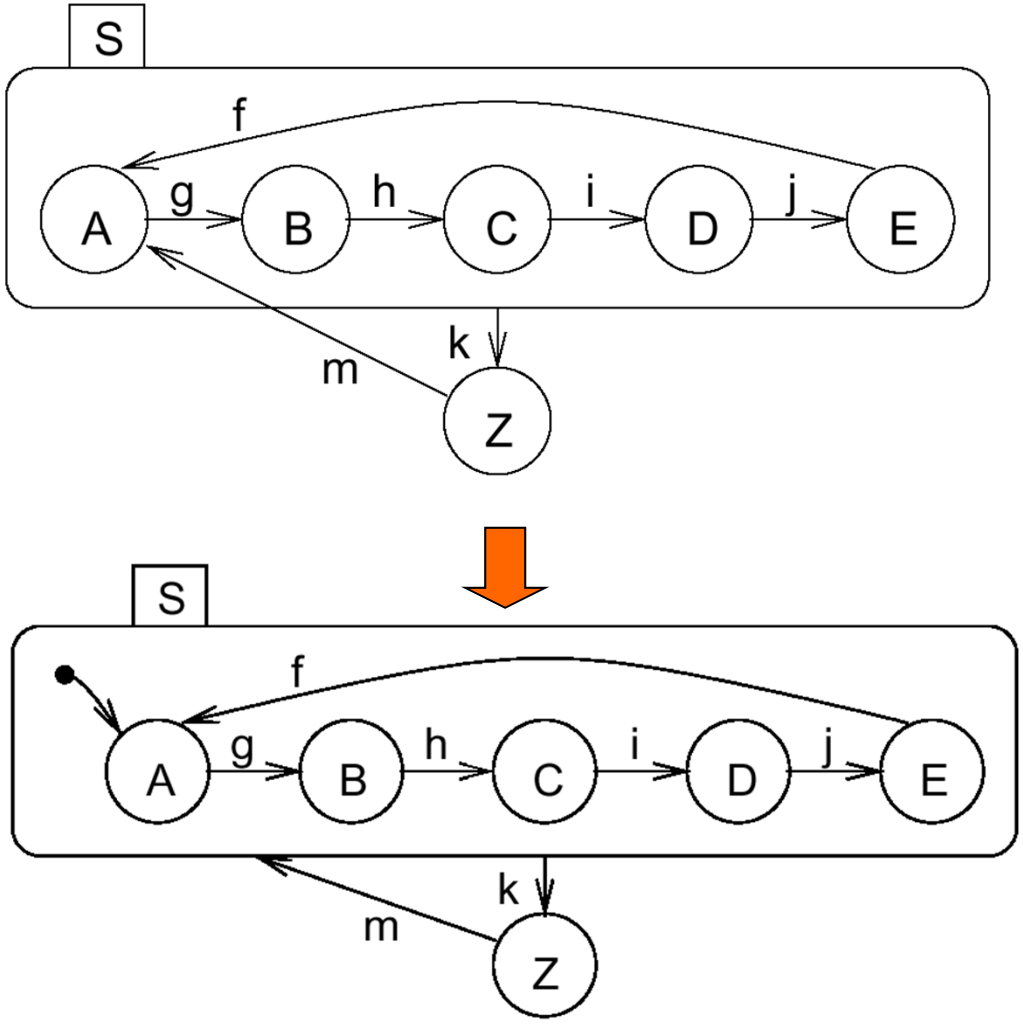
Hierarchy - transitions to super-states



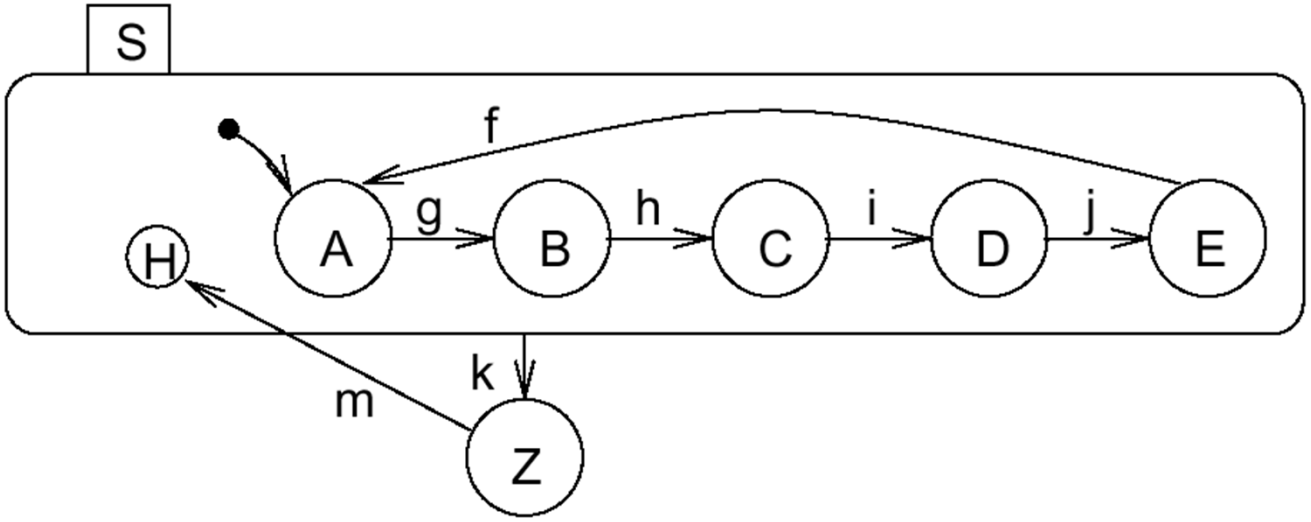
- What is the meaning of transitions to superstates,
  i.e., what basic state is entered when a superstate is entered?
  - default state mechanism
  - history mechanism

Default state mechanism

- Filled circle indicates sub-state entered whenever super-state is entered.
- Not a state by itself!
- Allows internal structure to be hidden for outside world

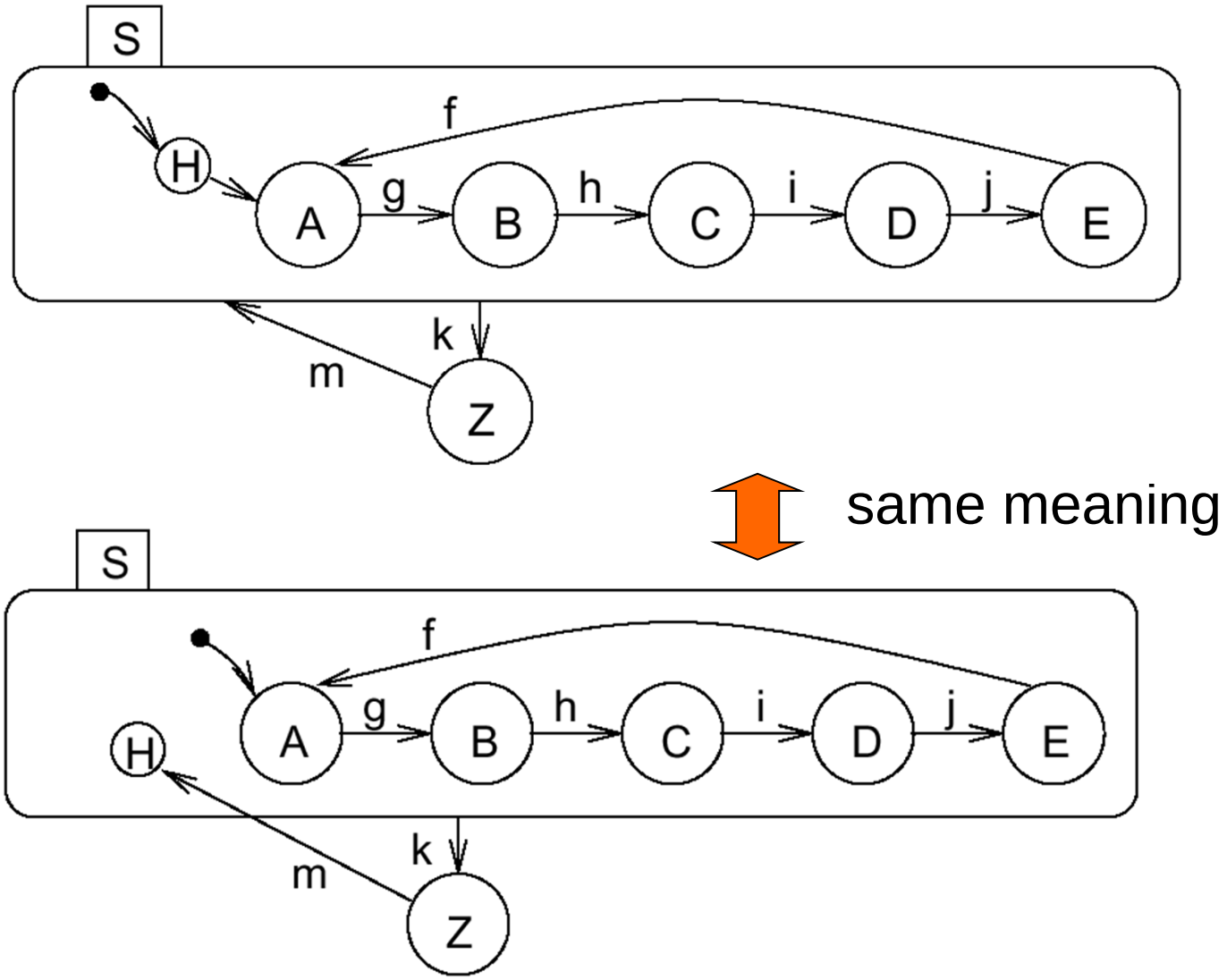


History mechanism



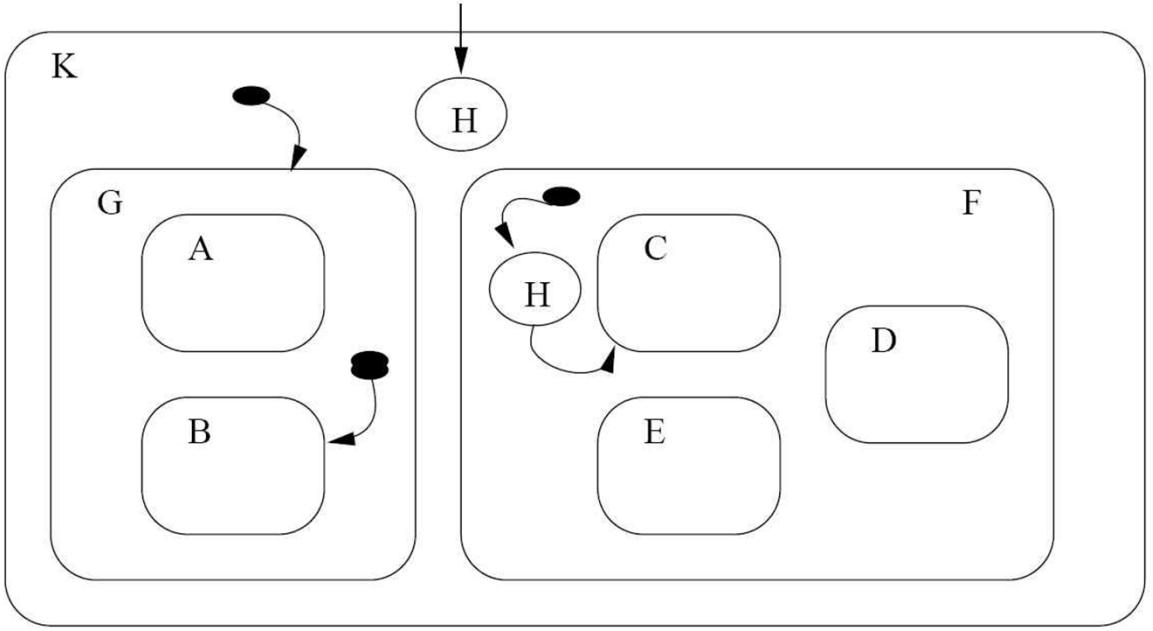
- For event m, S enters the state it was in before S was left (can be A, B, C, D, or E). If S is entered for the very first time, the default mechanism applies.

Combining history and default state mechanism



History and default state mechanism

- History and default mechanisms may be used at different levels of hierarchy.



Variables with complex data types

Problem of classical automata:

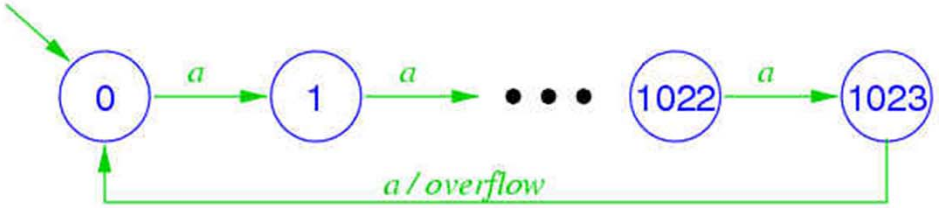
- Both control and data have to be represented as graphical states

Here:

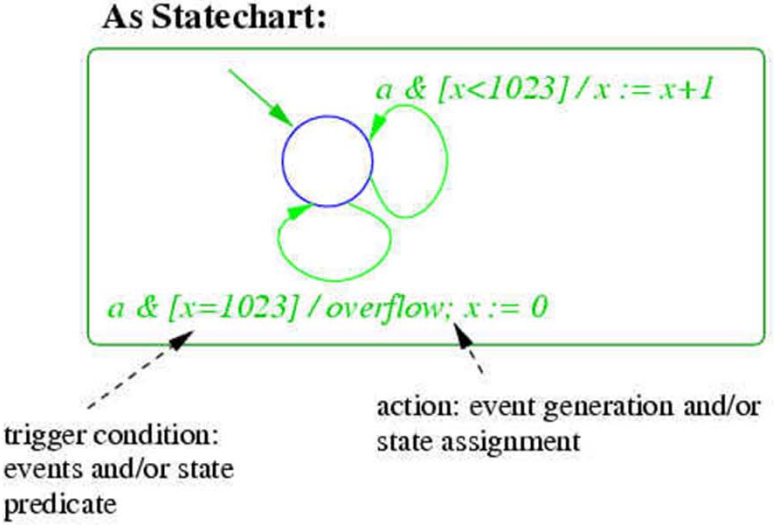
- Include typed variables (e.g. integers, reals, strings, records) to represent data
- Both „graphical states“ and variables contribute to the state of the statechart.
- Notation:
  - „graphical states“ = states
  - „graphical states“ + variables = **status**

A 10-Bit counter, counting on event a and issuing *overflow* after 1024 occurrences:

As FSM:



As Statechart:



Events and variables

Events:

- Exist only until the next evaluation of the model
- Can be either internally or externally generated

Variables:

- Values of variables keep their value until **they are reassigned.**

General form of edge labels

Meaning:

- Transition may be taken, if event occurred in last step and condition is true
- If transition is taken, then reaction is carried out.

Conditions:

- Refer to values of variables

Actions:

- Can either be assignments for variables or creation of events

Example:

- a & [x = 1023] / overflow; x:=0

Events, conditions, actions

- Possible events (incomplete list):
  - Atomic events
    - Basic events: A, B, BUTTON_PRESSED
    - Entering, exiting a state: en(S), ex(S)
    - Values of conditions: tr(cond), fs(cond)
    - Change of conditions: [cond], e.g. [X>5]
    - Change of values: ch(X)
    - Access to variables: rd(X), wr(X)
    - Timeout event tm(e,d):
      - Timeout event tm(e, d) is emitted d time units after event e has occured
  - Compound events: logical connectives and, or, not

Events, conditions, actions

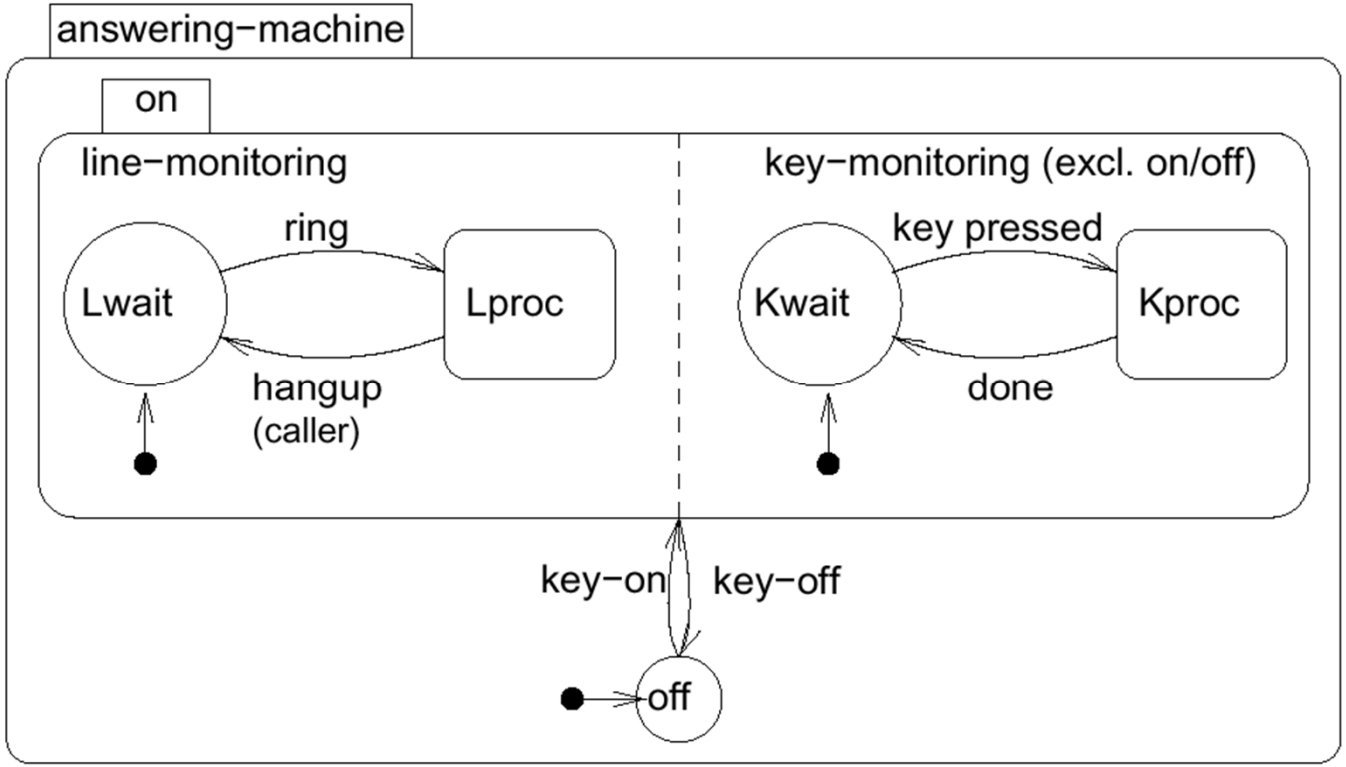
- Possible conditions (incomplete list):
  - Atomic conditions
    - Constants: true, false
    - Condition variables (i.e. variables of type boolean)
    - Relations between values: X > 1023, X · Y
    - Residing in a state: in(S)
  - Compound events: logical connectives and, or, not

Events, conditions, actions

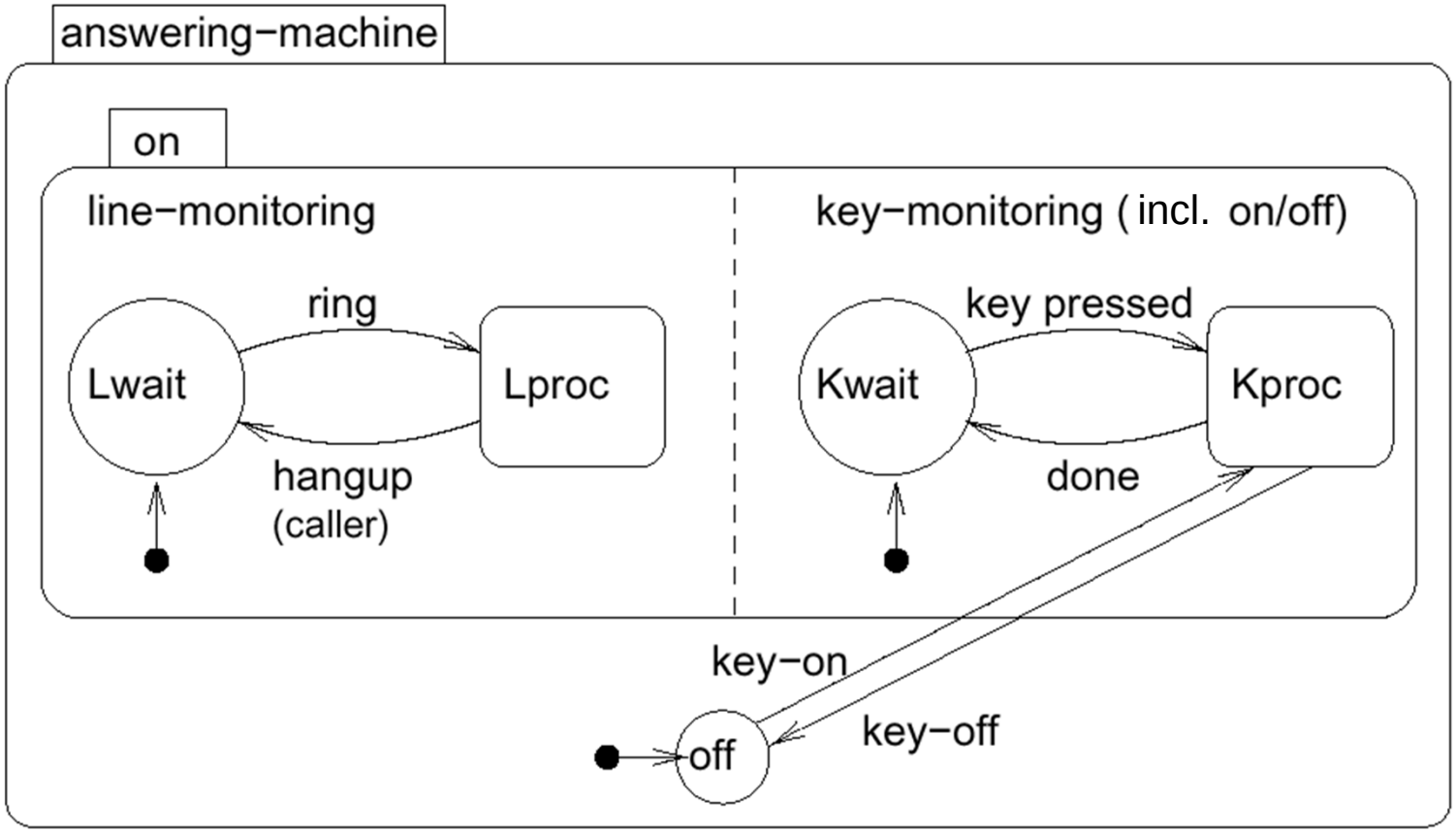
- Possible actions (incomplete list):
  - Atomic actions
    - Emitting events: E (E is event variable)
    - Assignments: X := expression
    - Scheduled actions: sc!(A, N) (means perform action after N time units)
  - Compound actions
    - List of actions: A1; A2; A3
    - Conditional action: if cond then A1 else A2

Concurrency

- Convenient ways of describing concurrency are required.
- **AND-super-states**: FSM is in **all** (immediate) sub-states of a AND-super-state; Example:

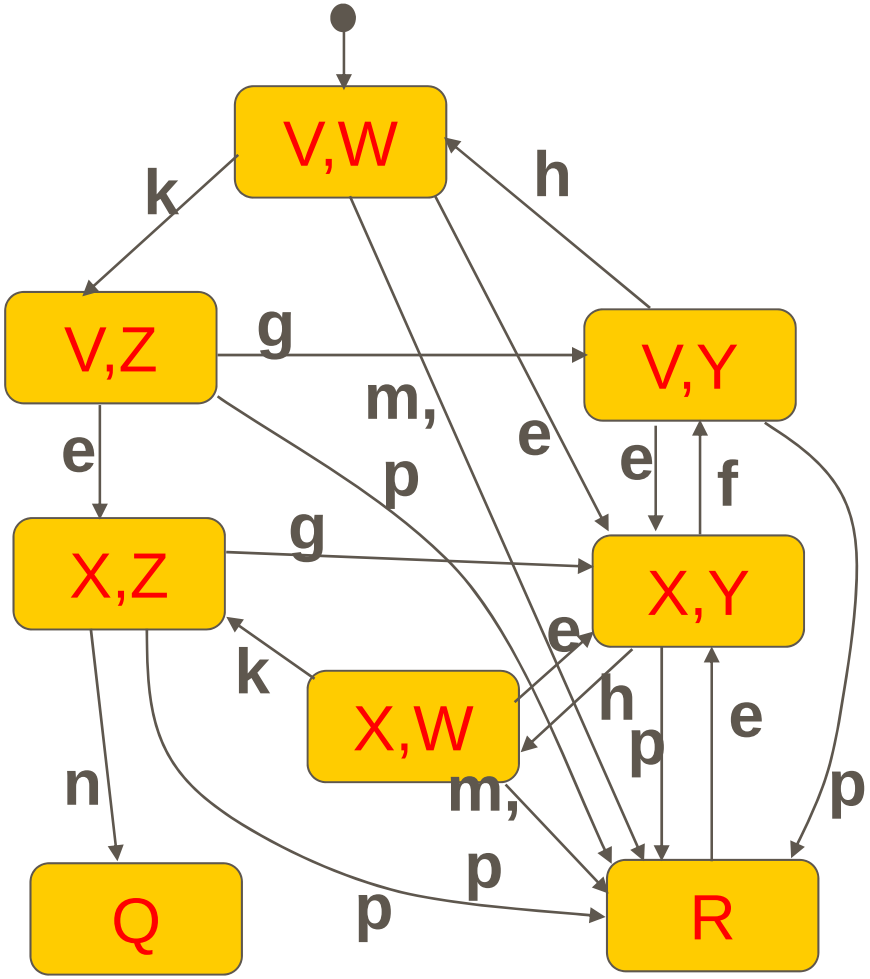


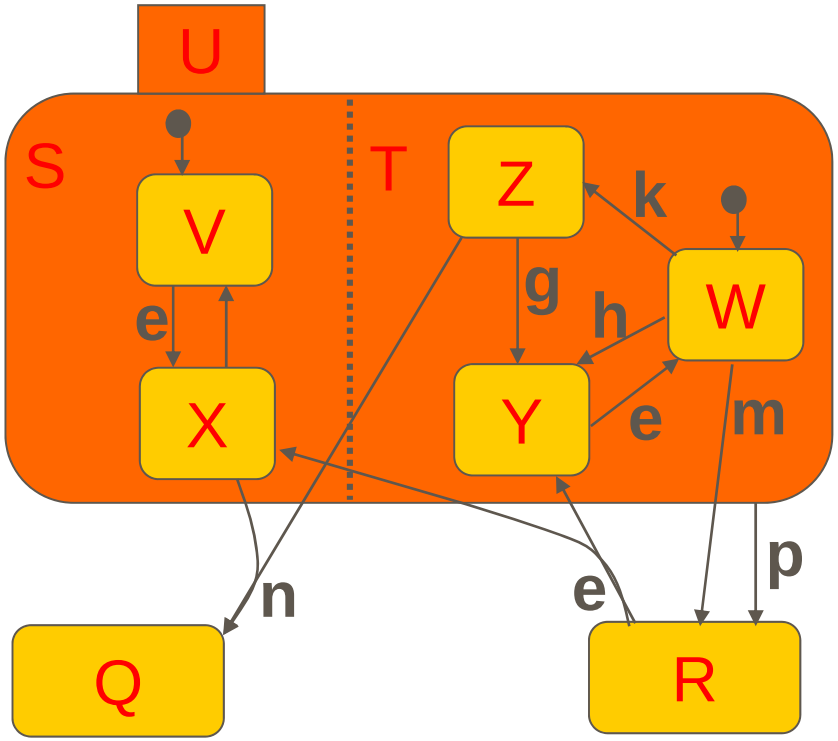
Entering and leaving AND-super-states



- Line-monitoring and key-monitoring are entered and left, when key-on and key-off events occur.

Benefits of AND-decomposition





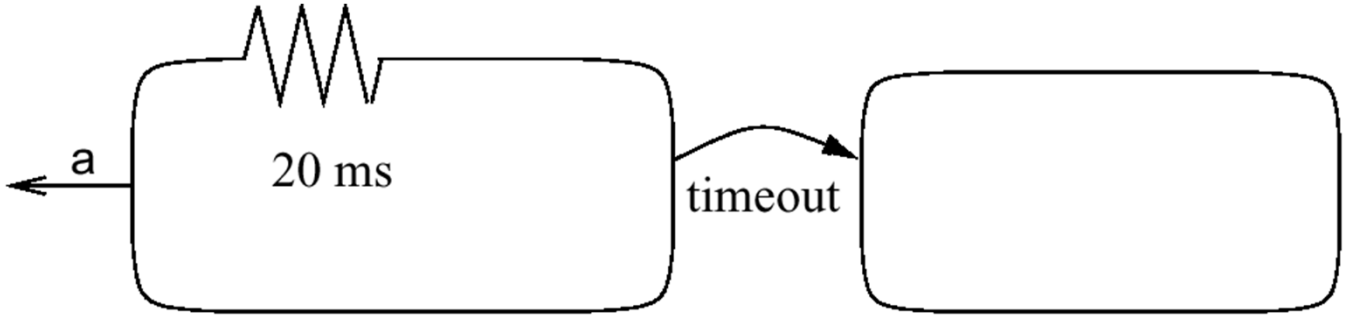
Types of states

In StateCharts, states are either

- **basic states,** or
- **AND-super-states,** or
- **OR-super-states.**

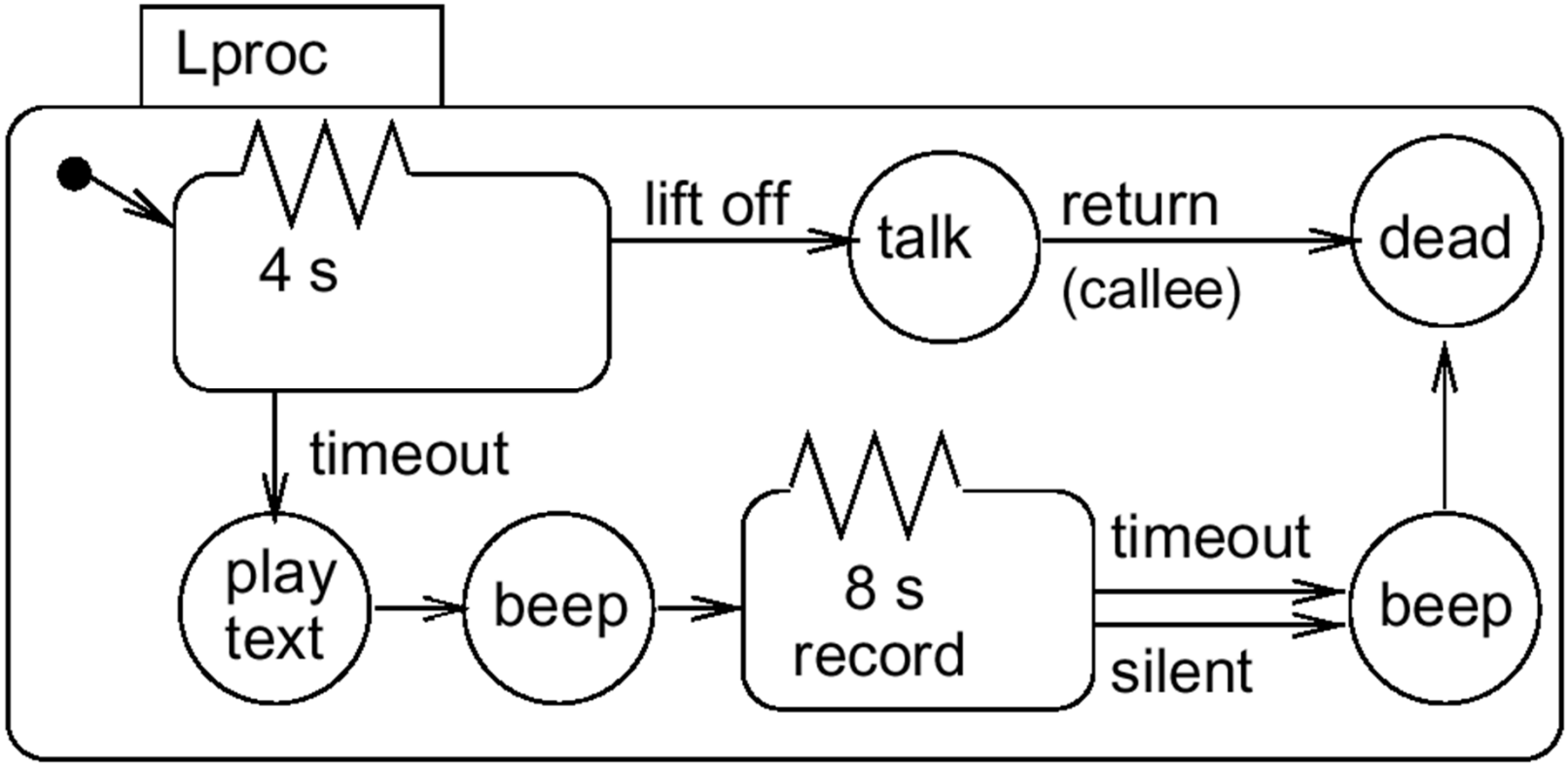
Timers

- Since time needs to be modeled in embedded systems, timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.

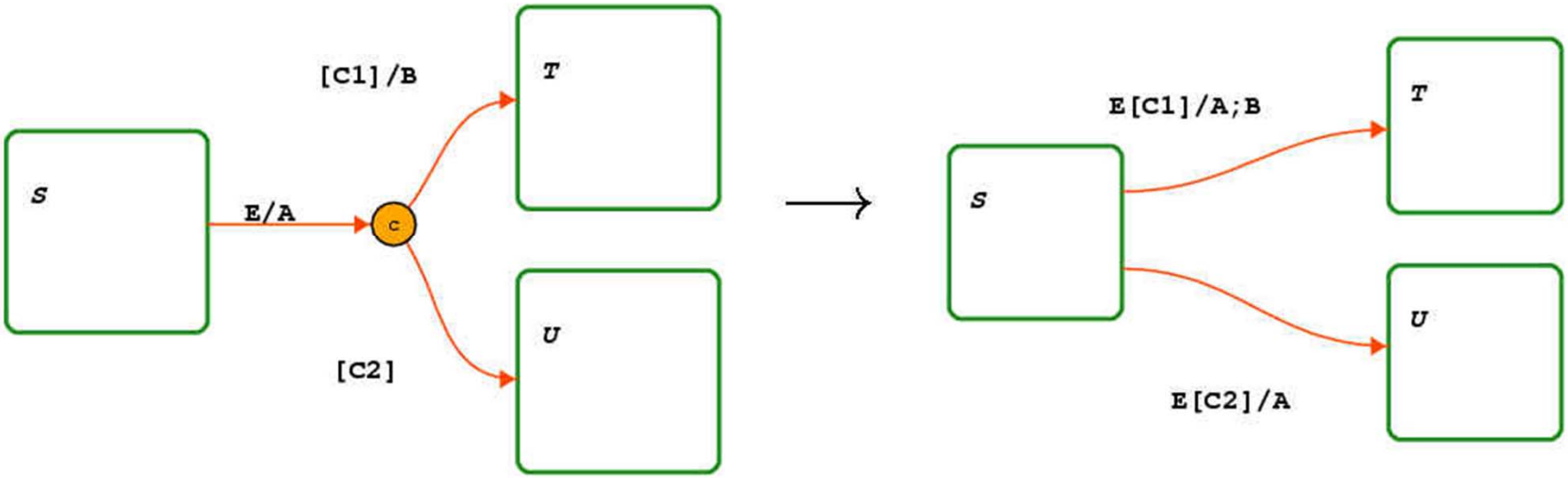


If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

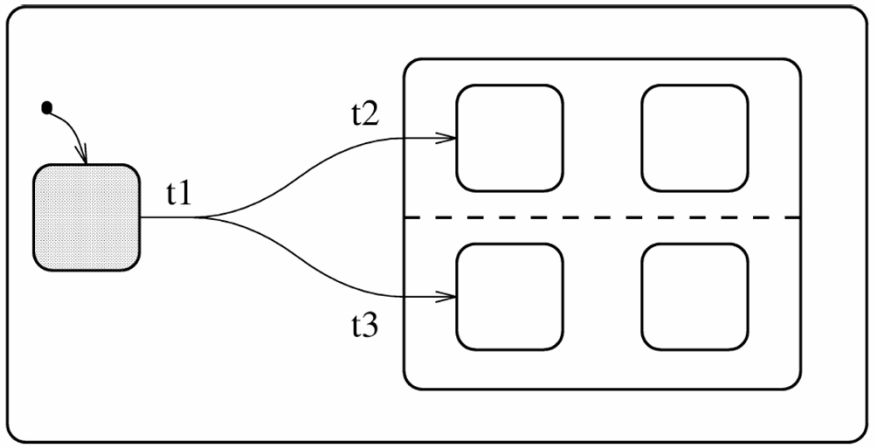
Using timers in answering machine

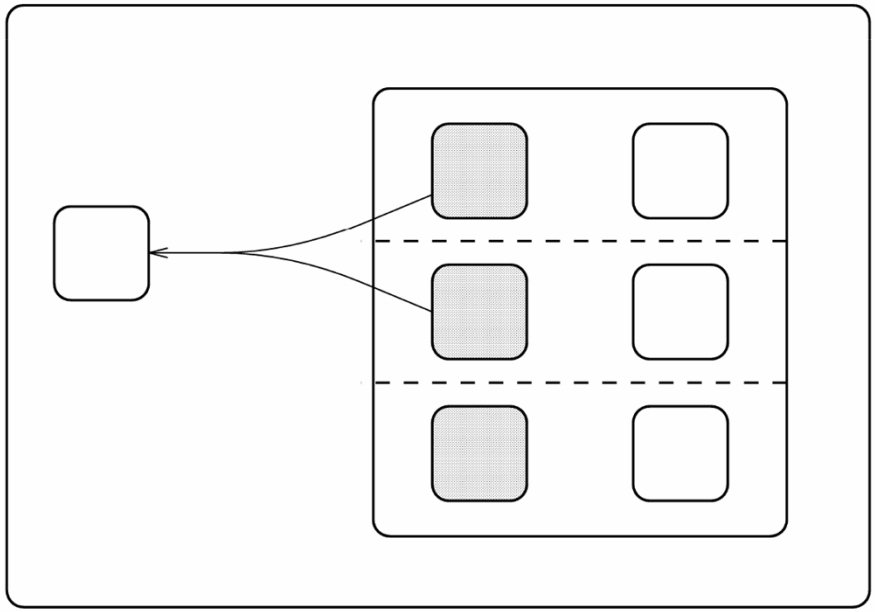


Condition connector

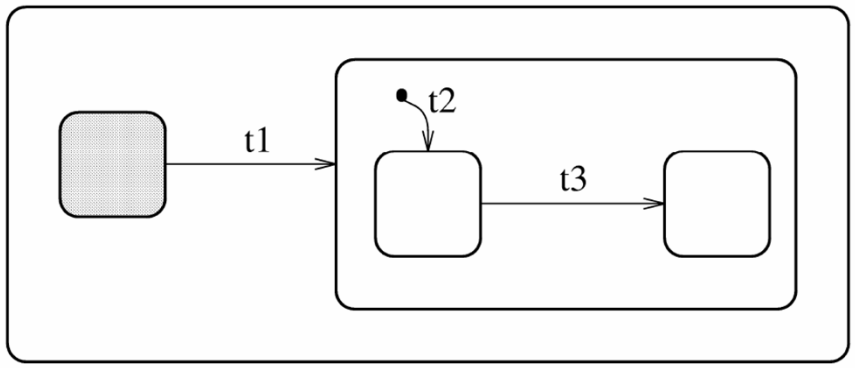


Join and Fork Connectors

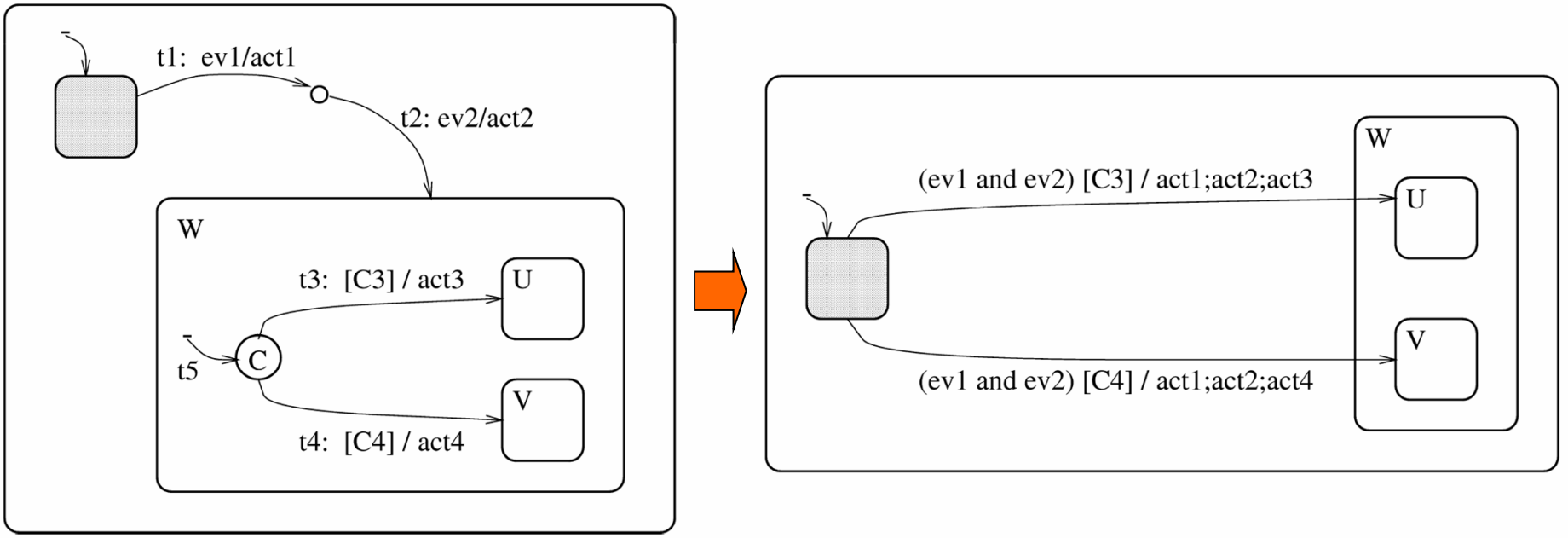




Compound transitions

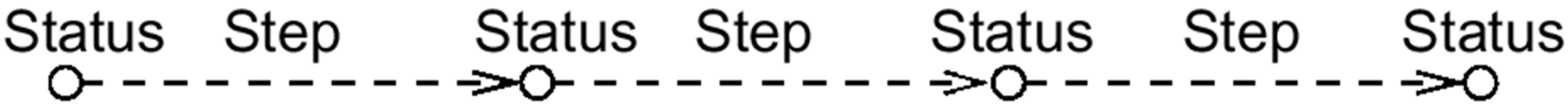


t1 and t2 must be executed together



Semantics of StateCharts

- Execution of a StateChart model consists of a sequence of **steps**
- A step leads from one **status** to another



- One step:
  - Given:
    - Current system status s_i
    - Current time t
    - External changes Δ
  - Find:
    - New status s_{i+1}

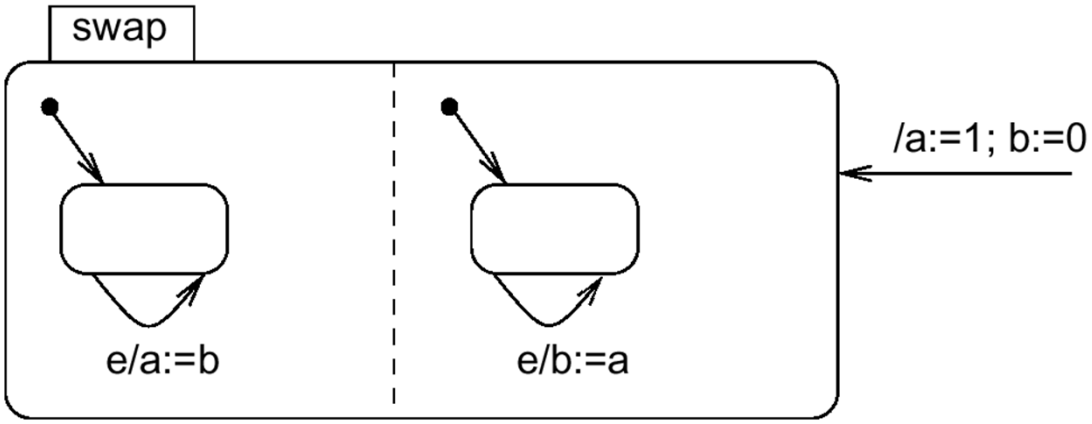
The StateCharts simulation phases (StateMate Semantics)

- How are edge labels evaluated?
- Three phases:

1. Effect of external changes on events and conditions is evaluated,
2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
3. Transitions become effective, variables obtain new values.

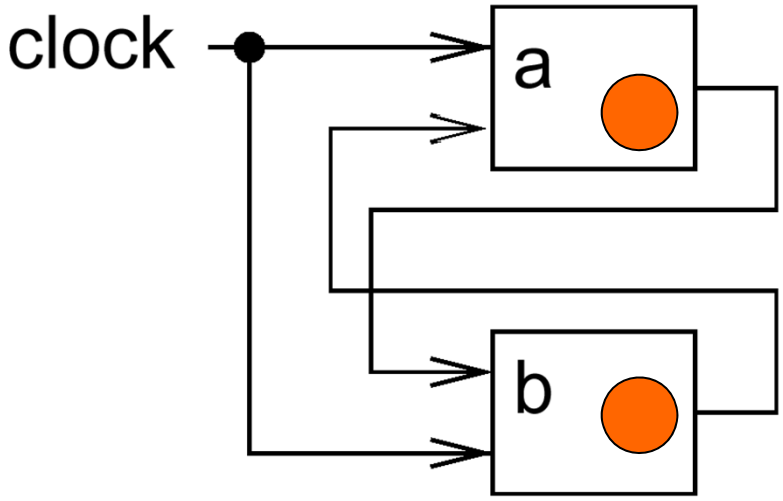
- Separation into phases 2 and 3 guarantees and reproducible behavior.

Example



- In phase 2, variables a and b are assigned to temporary variables. In phase 3, these are assigned to a and b. As a result, variables a and b are swapped.
- In a single phase environment, executing the left state first would assign the old value of b (=0) to a and b. Executing the right state first would assign the old value of a (=1) to a and b. The result would depend on the execution order.

Reflects model of clocked hardware

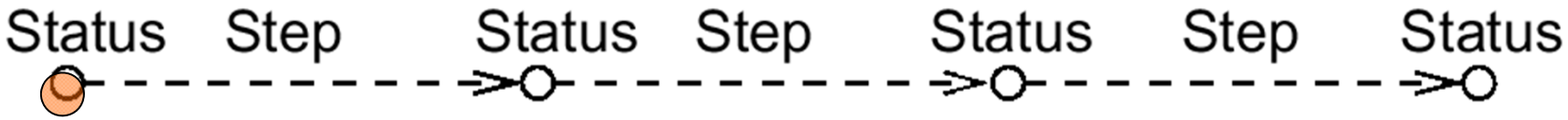


- In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.

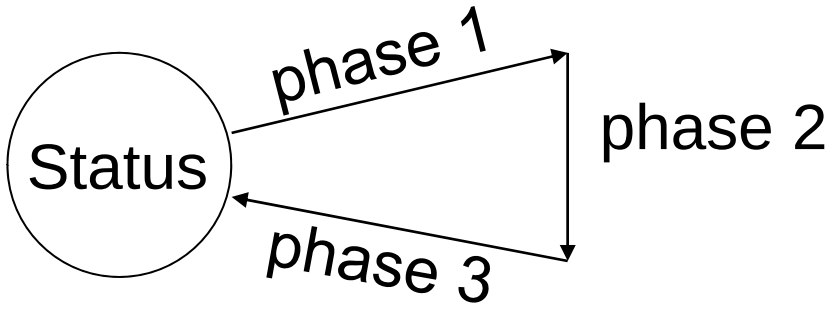
Steps

- Execution of a StateMate model consists of a sequence of (status, step) pairs



Status= values of all variables + set of events + current time

Step = execution of the three phases (StateMate semantics)



Other implementations of StateCharts do not have these 3 phases (and hence are non-determinate)!

Other semantics

- Several other specification languages for hierarchical state machines (UML, dave, …) do not include the three simulation phases.
- These correspond more to a SW point of view with no synchronous clocks.
- LabView seems to allow turning the multi-phased simulation on and off.

Broadcast mechanism

- Values of variables are visible to all parts of the StateChart model
- New values become effective in phase 3 of the current step and are obtained by all parts of the model in the following step.

!

☞ StateCharts implicitly assumes a **broadcast** mechanism for variables
(→ implicit ***shared memory communication***
–other implementations would be very inefficient -).

☞ StateCharts is appropriate for local control systems (☺), but not for distributed applications for which updating variables might take some time (☹).

Lifetime of events

- **Events live until the step following the one in which they are generated** (“one shot-events“).

Time models

- External events and external changes of variables are associated with physical times.
- But how does time proceed internally?
- How many steps are performed before external changes are evaluated?

The synchronous time model

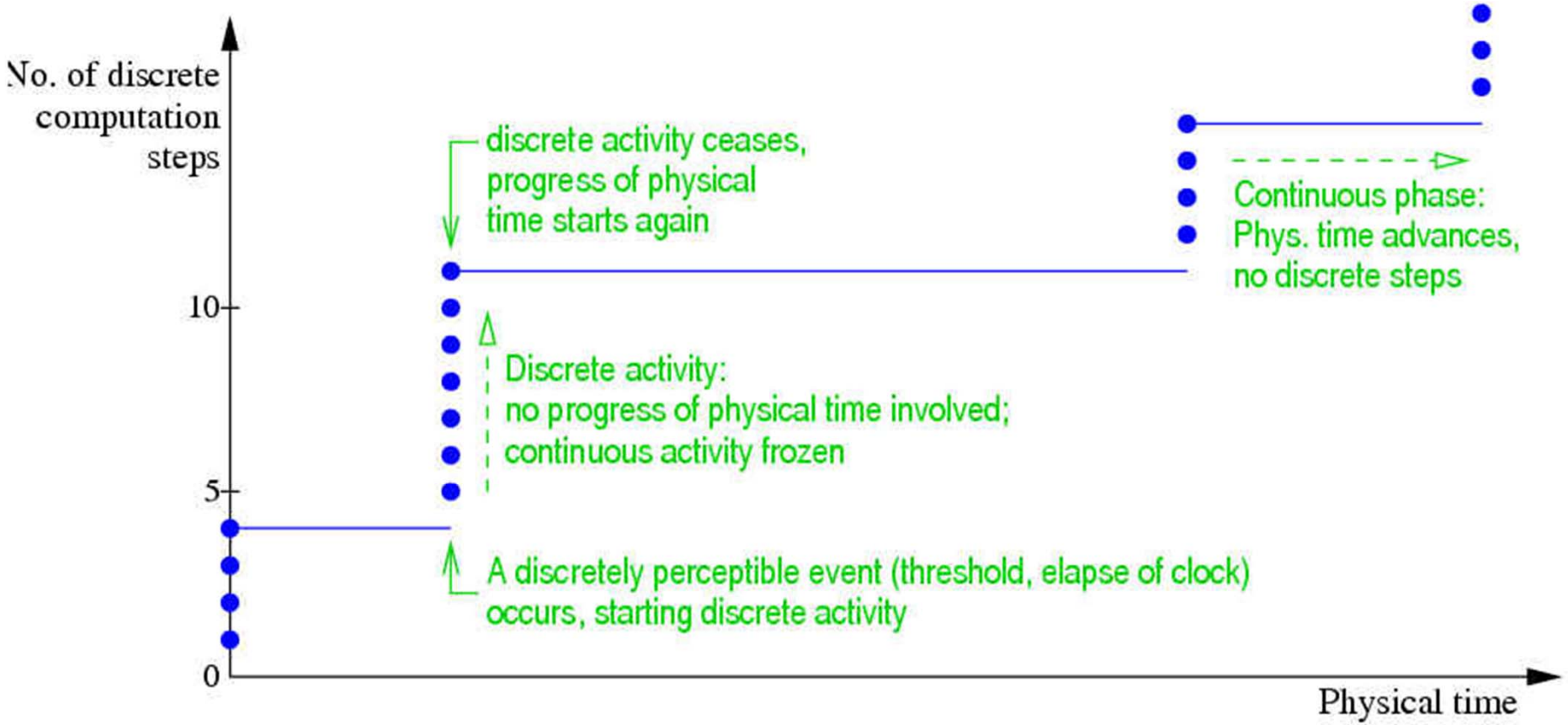
- A single step every time unit.
- If the current step is executed at time *t*, then the next step is executed at time *t+1*.
  - Events and variable changes are communicated between different states during one time unit.
  - External changes are only accumulated during one time unit.

The super-step time model (1)

- A step of the statechart does not need time.
- Super-steps are performed:
  - A super-step is a sequence of steps.
  - A super-step terminates when the status of the system is stable.
  - During a super-step the time does not proceed and thus external changes are not considered.
- After a super-step, physical time restarts running, i.e. activity of the environment will be possible again.
- The computation of the statechart is resumed when
  - external changes enable transitions in the statechart
  - Timeout events enable transitions of the statechart

The super-step time model (2)

- Two-dimensional time:



- Assumption: Computation time is neglegible compared to dynamics of the environment.

The super-step time model (3)

- During one super-step the number of communications between different states is not restricted. All communications are assumed to be performed in zero time.
- Simplified model for reality.
- Can only be realistic, if
  - Discrete computations are fast compared to dynamics of the environment.
  - Discrete computations will be stable after a restricted number of steps.
- Timeout events can reactivate a statechart

  $\Rightarrow$ Possible to specify statecharts which permit progress of physical time after a limited number of steps and reactivate themselves via timeout events

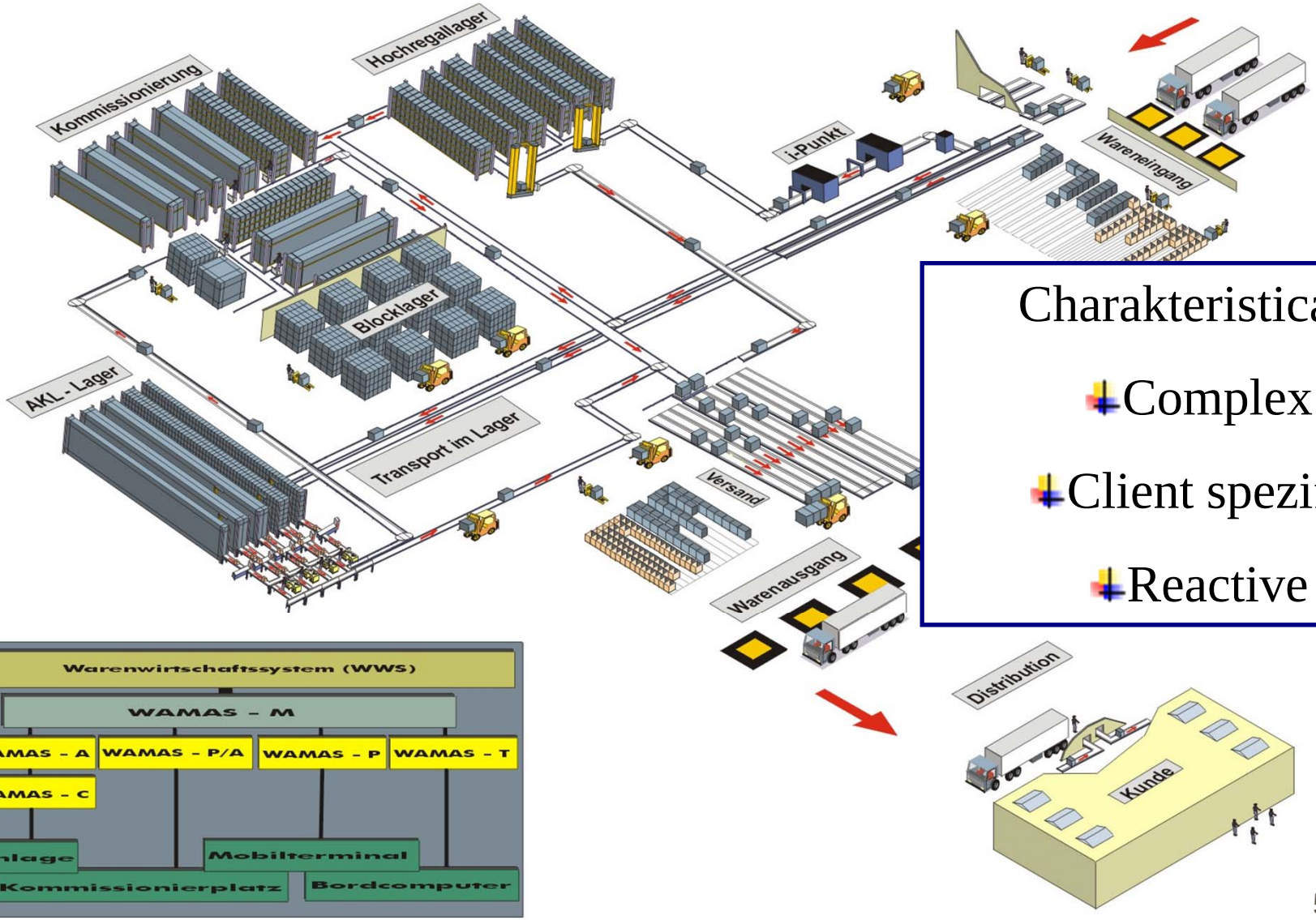
Evaluation of StateCharts (1)

- **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:**
  - Not useful for distributed applications,
  - No program constructs,
  - No description of non-functional behavior,
  - No object-orientation,
  - No description of structural hierarchy.
  - Generated C programs may be inefficient
  - Generated VHDL programs mostly behavioral

Use case – logistic system



Charakteristica

- Complex
- Client spezific
- Reactive