AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture & Design of Embedded Real-Time Systems (TI-AREM)

Behavioral Object Analysis: State Machines 1.

UNIFIED
MODELING
LANGUAGE
TM

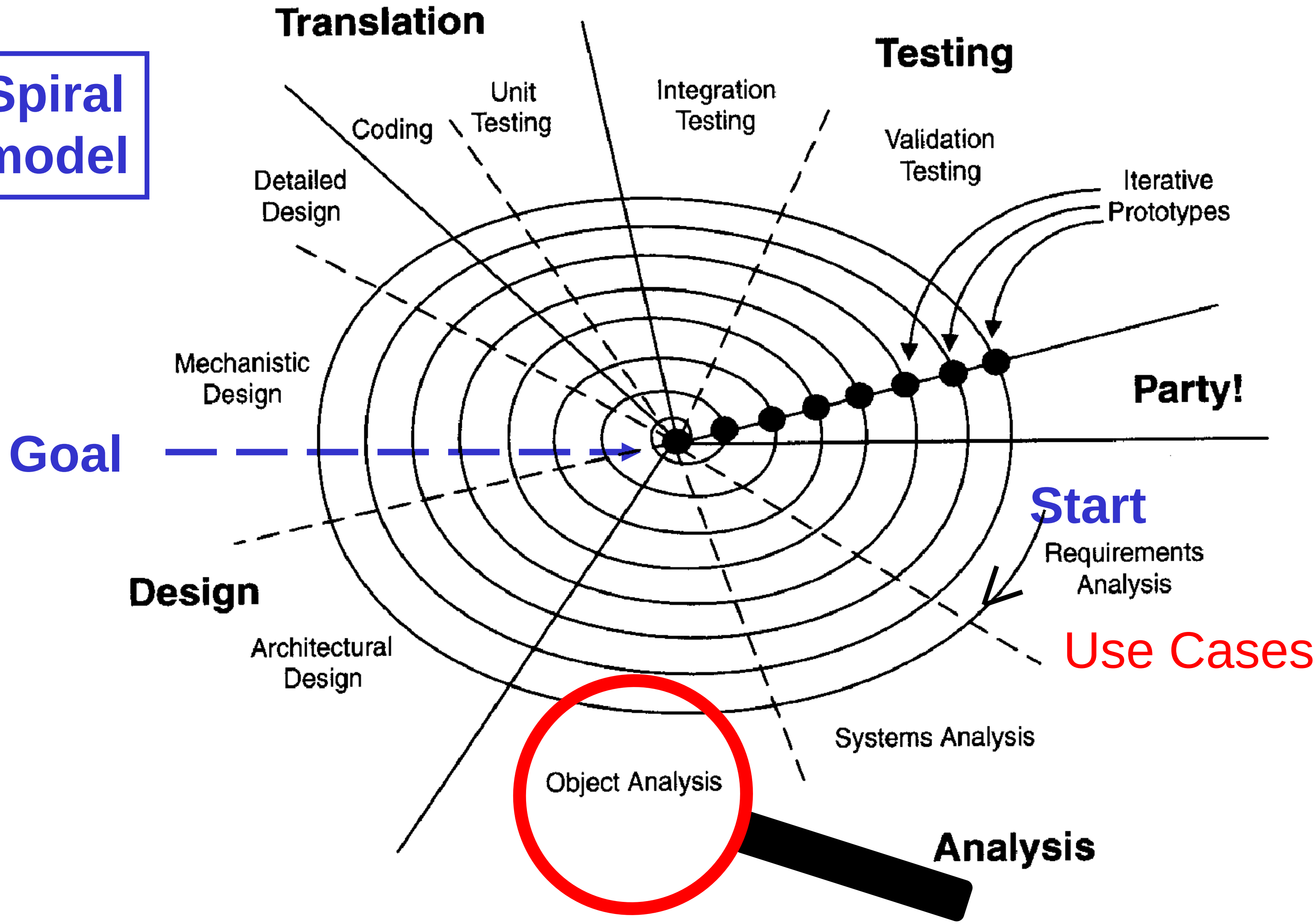
Version: 2-2-2015

Agenda

- Introduction to ROPES:
  - Behavioral Object Analysis
- State Machines
  - Simple
  - Advanced
  - Protocol State Machines
  - Example of an automated rail car system

ROPES: an iterative Rapid Development Process

Spiral model



Object Behavior

- An object's behavior is defined by:
  - the set of operations defined on the class
  - the constraints on their applications
- Functional constraints are often modeled by **Finite State Machines (FSM)**
- Non-functional constraints – also called Quality of Service (QoS)
  - Performance, accuracy etc.
  - Can be specified by **OCL** (Object Constraint Language – defined by UML)

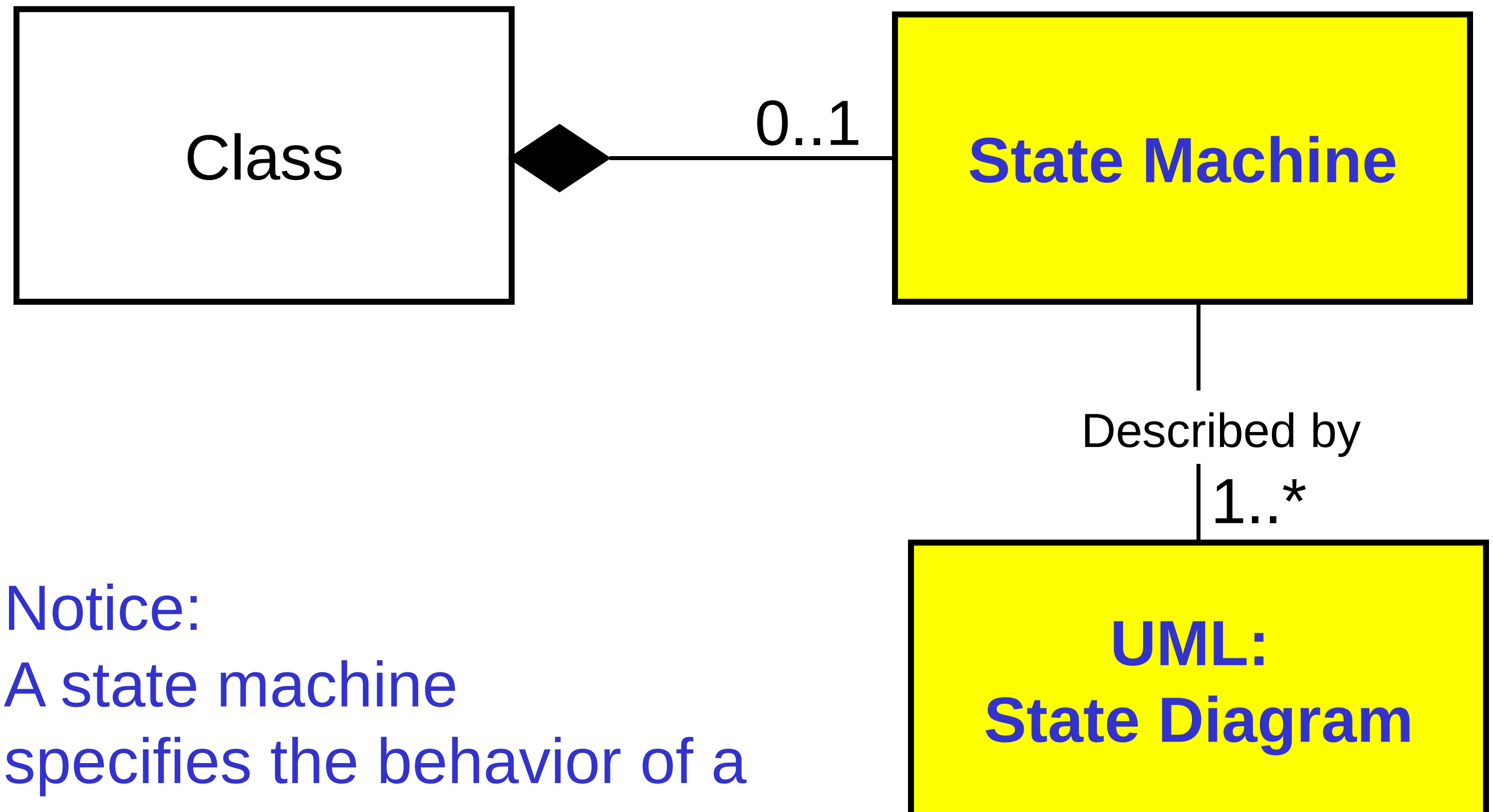
Three Types of Behavior

- An object can have one of the following three types of behavior:
  - **Simple behavior**
    - Responds always to an input in exactly the same way – does not depend on the objects history
  - **Continuous behavior**
    - Depends on the objects history but in a smooth, continuous way
    - Behavior that cannot be divided into disjoint states
    - Examples: digital filters, PID control loops, fuzzy logic
  - **State behavior**
    - Reactions to same input is dependent of the actual state
    - The objects behavior can be divided into disjoint sets
    - Specified as a state machine with a UML state diagram

Two Types of State Machines

- **Behavioral State Machines**
  – used to specify behavior of various model elements e.g. classes
- **Protocol State Machines**
  – new in UML 2.0
  – used to express usage protocols e.g. to express an order of invocations of an objects operations
  – can be associated with interfaces and ports
  – interfaces may own a protocol state machine that specifies event sequences and pre/post conditions for the operations and receptions described by the interface

Behavioral State Machines and Classes



Notice:
A state machine specifies the behavior of a single class or a Use Case

UML State Diagram History

- Based on Finite State Machine theory
  - used in many years in design of digital HW
  - **Mealy Machines**: event / action semantic
  - **Moore Machines**: event / activity semantic
- UML **State diagrams**
  - based on David Harel's state chart theory
  - first introduced in OO in OMT 1991 => UML standard in 1997
  - UML's State Diagrams have important enhancements compared to traditional state machines
    - for example nested and orthogonal states

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

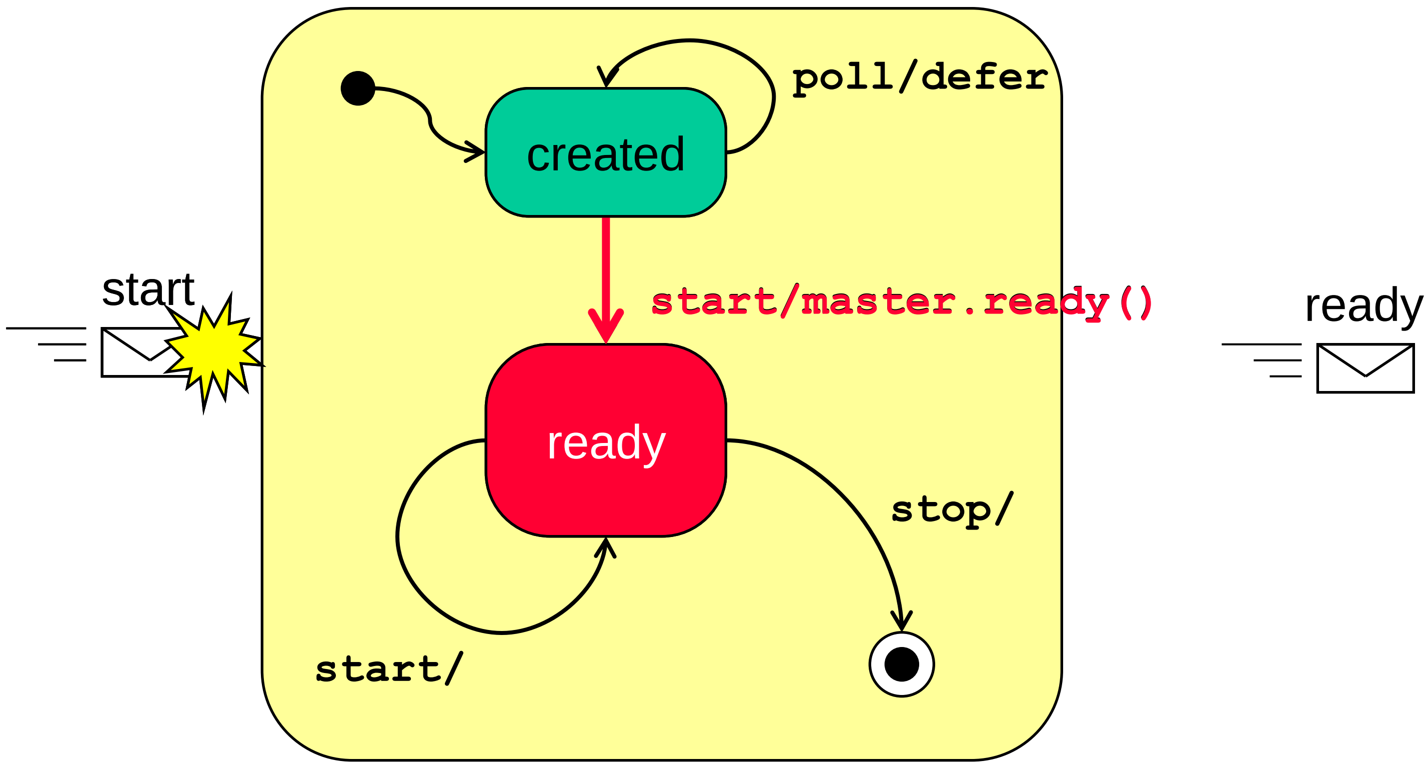
Definition of State

- **State definition:**
  - “A **state** is a condition that persist for **a significant period of time**, which is both **distinguishable** from other such conditions and **disjoint** with them”.
- A more pragmatic definition:
  - “A state is a condition of an object during which **a set of events is accepted** and some **actions** and **activities** are executed, and the object can reach some set of states based on the events it accepts”

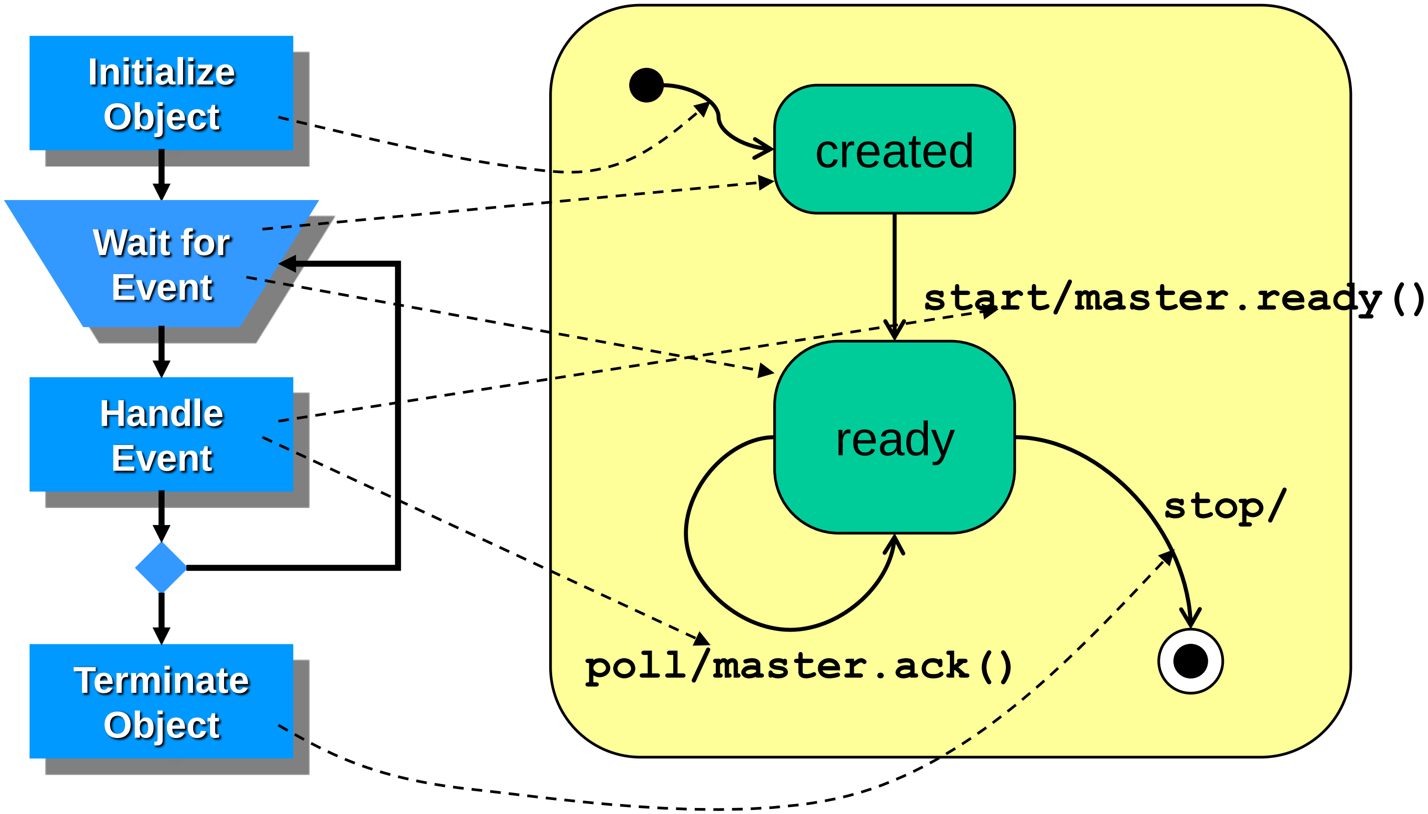
AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

State Machine Behavior (1)

- Event handling details depend on state

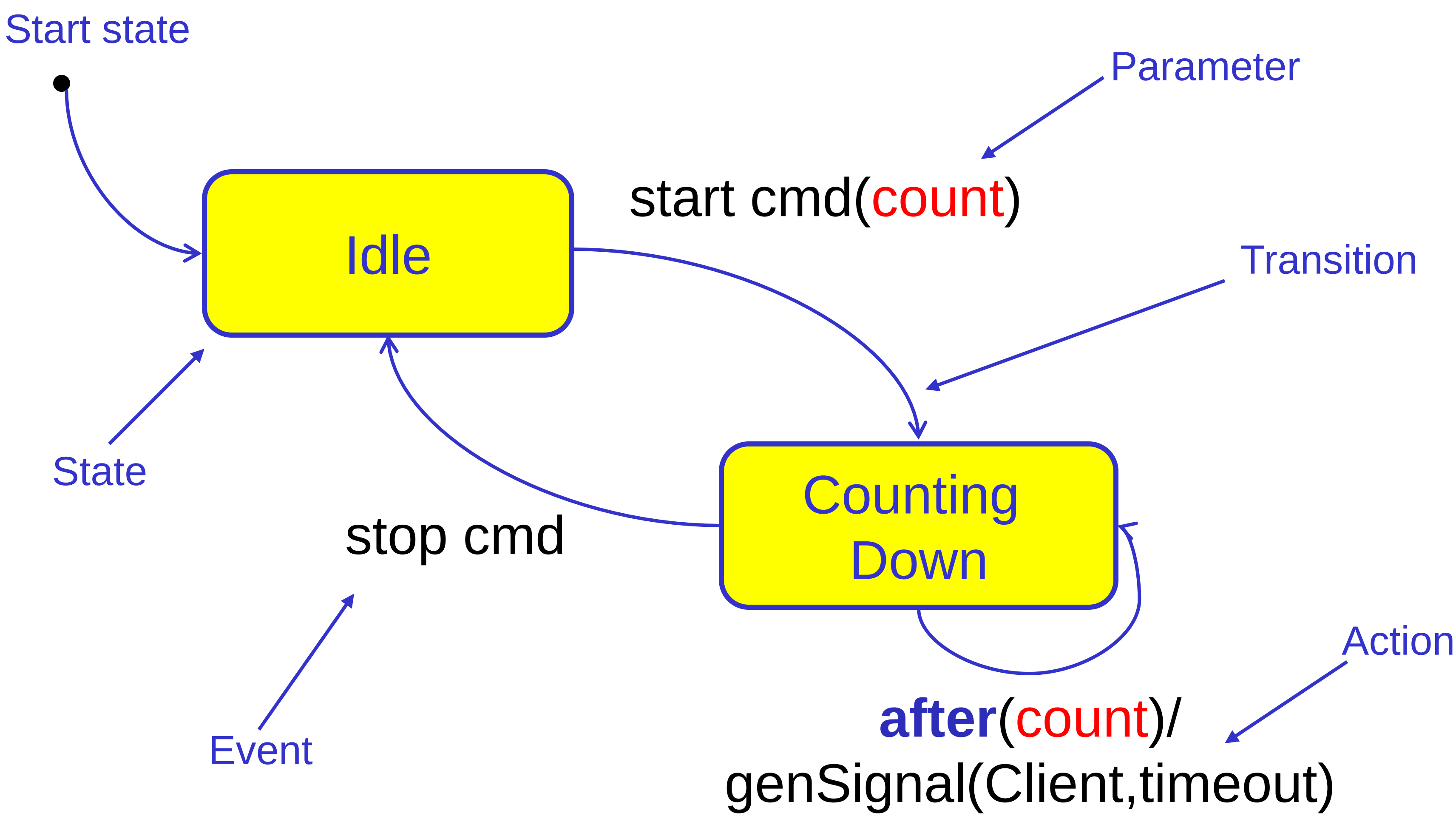


State Machine Behavior (2)



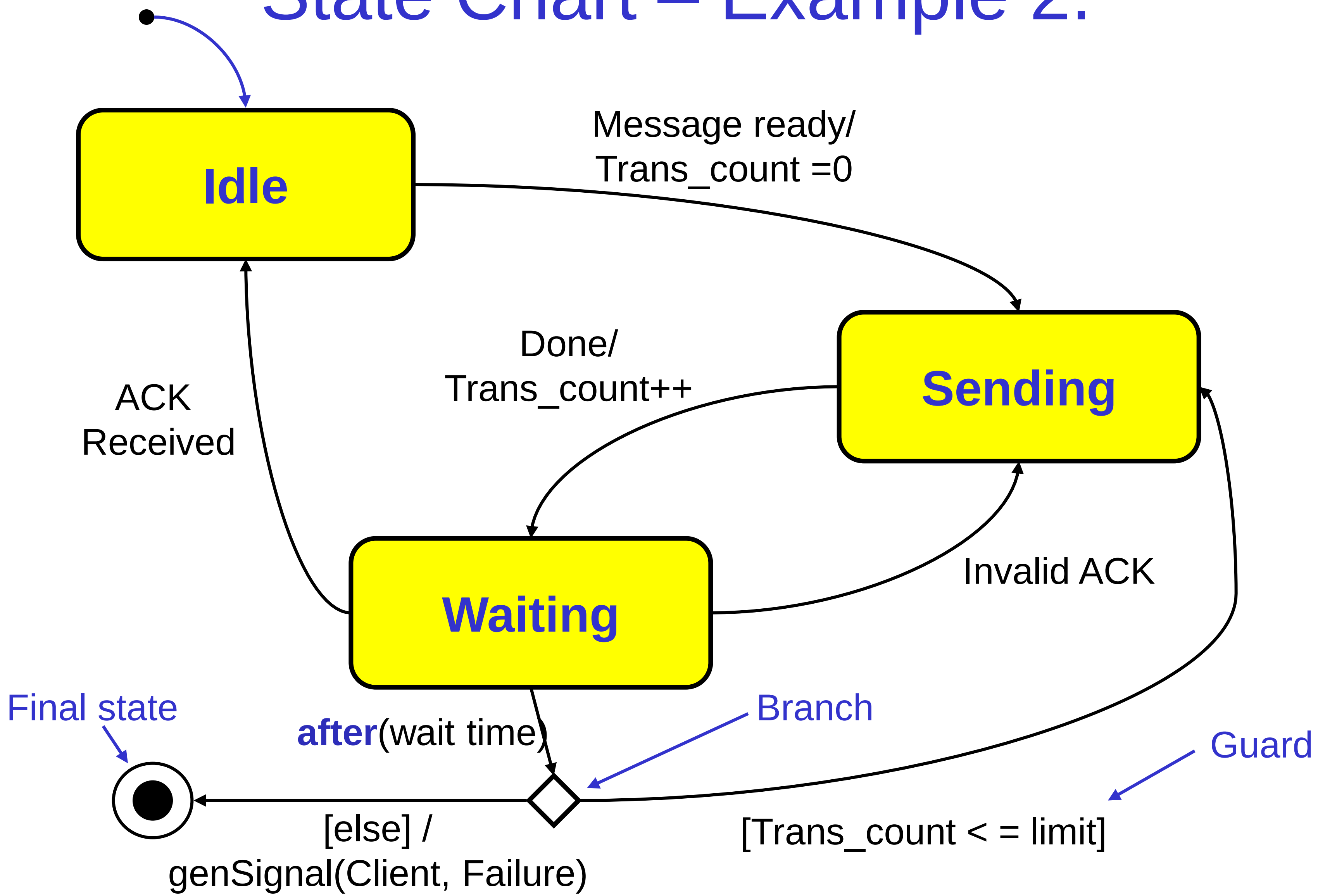
UML State Diagrams are based on a Run-to-completion semantic

State Chart – Example 1.



Retriggerable One-Shot Timer

State Chart – Example 2.



Transitions and Events

UML defines four kinds of events:

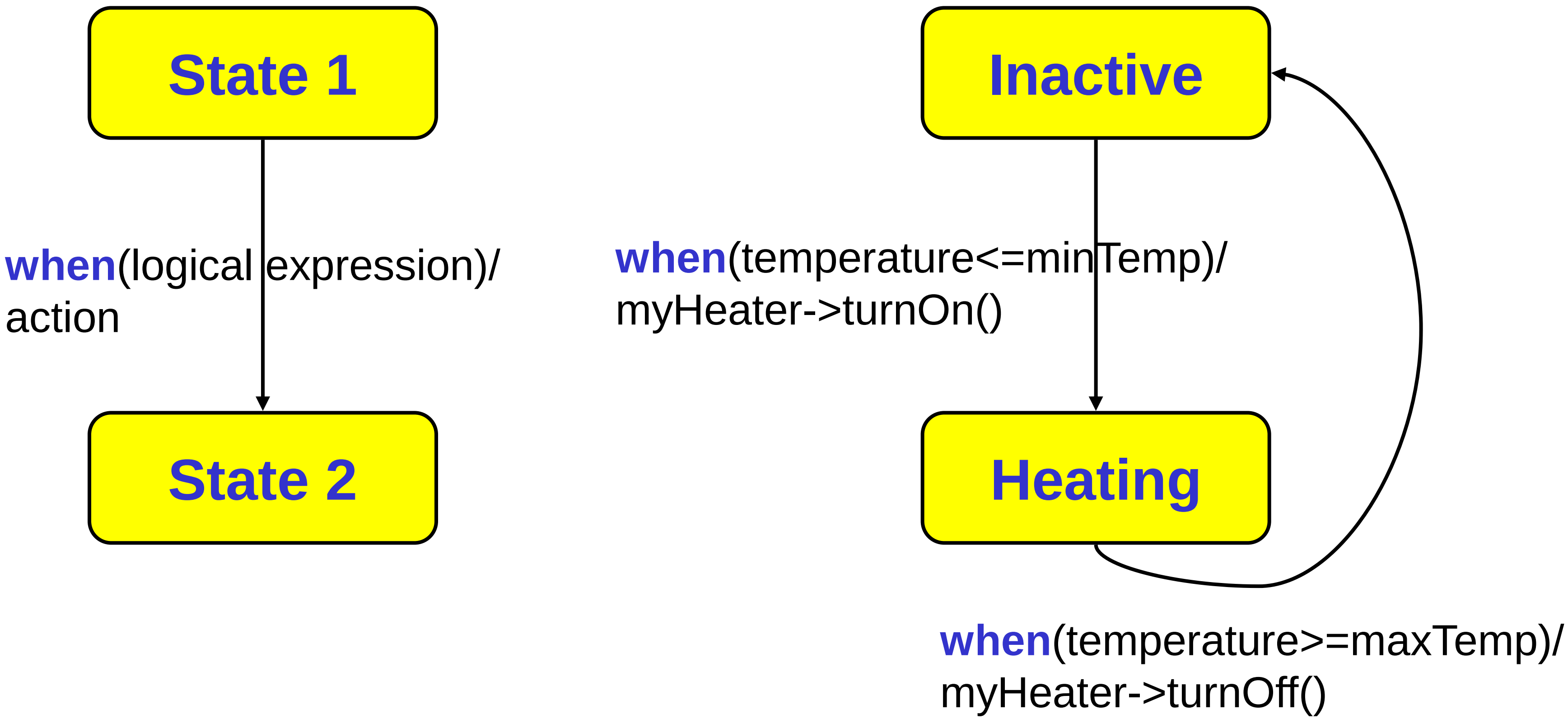
1. **Signal events**
   – An **asynchronously** occurrence of interest arising form outside the scope of the state machine
2. **Call events**
   – An explicit **synchronous** notification of an object by another
3. **Change events**
   – An event based on the changing of an attribute value
4. **Time events**
   – The elapse of a specific duration of time or the arrival of an absolute time

Syntax for a State Transition

event-trigger (parameters) [guard] / action list

Field	**Description**
Event-trigger	The name of the event triggering the transition
Parameters	A comma-separated list of data parameters passed with the event signal
Guard	A boolean expression that must evaluate to TRUE for the transition to be taken, often used with the conditional connector
Action list	A comma-separated list of actions executed as a result of the transition being taken. These operations may belong to this or another object

Change Events

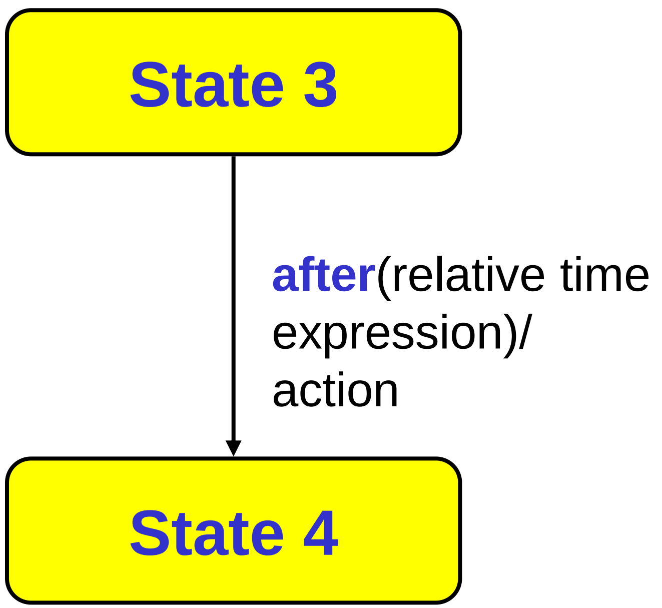


Notice: A when clause will be continuously evaluated

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

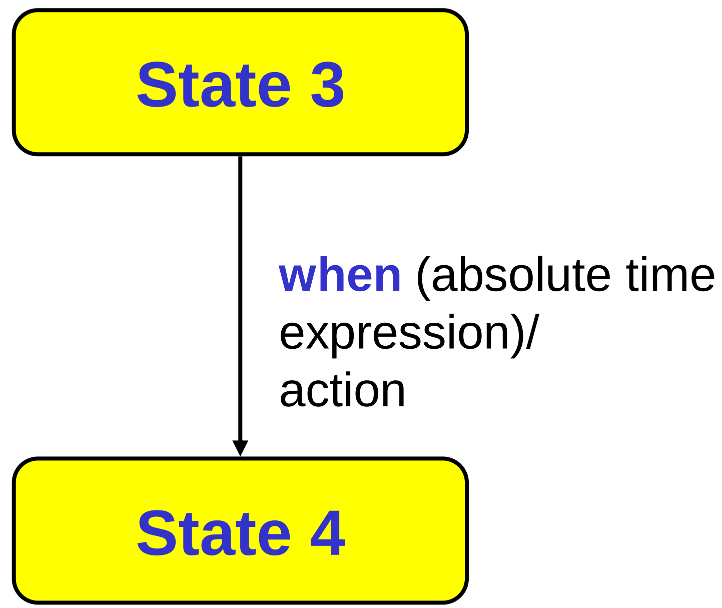
Time Events

after



after(10ms)

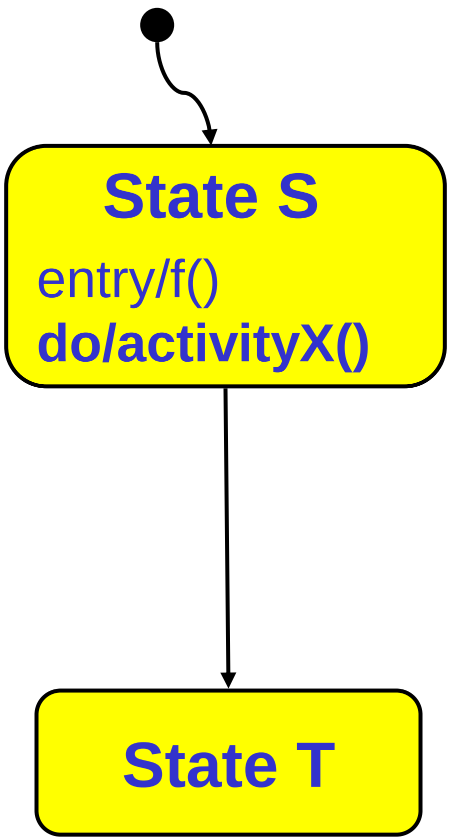
when



when(actualTime = 23.59.00)/ startBackUp

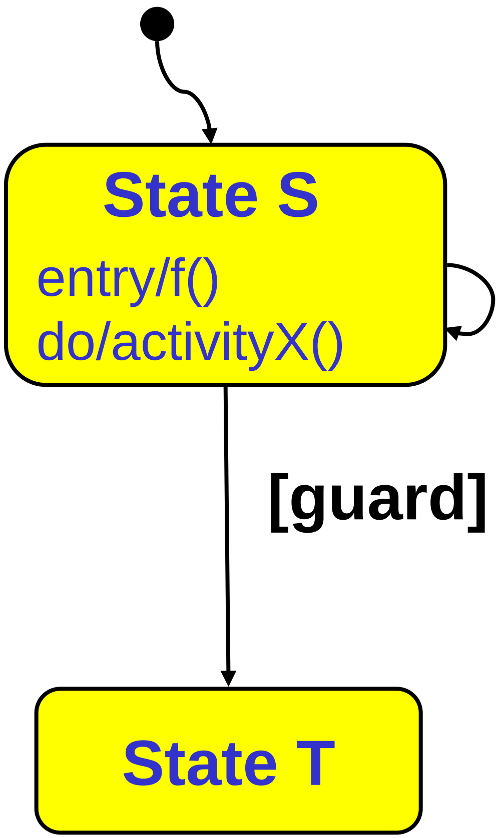
Null-triggered Transition

- Triggered upon completion of entry actions and any state activities
- May contain a guard condition
- Will only be evaluated once, even if the guard condition later becomes true
- Also called a “completion transition”



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Null-triggered Transition Problem



If the guard evaluates to FALSE then we have a problem.

If State S has a self transition e.g. EventX – the guard will be evaluated once more

Guards

- A guard occurs in connection with an event
- The boolean guard expression is only evaluated, when the event occurs
- A guard can be a logical expression based on the objects attribute
- A guard can also test the state of a concurrent substate [IS_IN(idle)] (AND state)

Actions

- An action can be:
  - to call an operation on the same object
  - to call an operation on an associated object
  - to create or to destroy other objects
  - manipulation of attributes for example an assignment
  - to send a signal to other and-states or to other objects

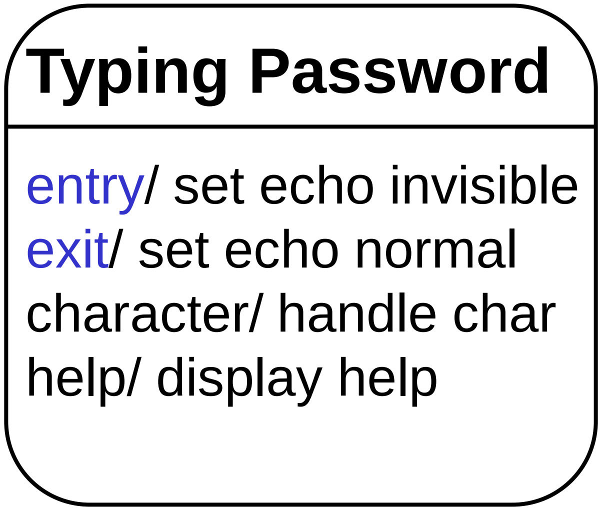
Actions and Activities

- Actions are short and non-interruptible
- Actions are performed during state transitions or during internal transitions
- Activities are only performed in states
- Activities have longer durations and are performed as long as the state machine is in a given state

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

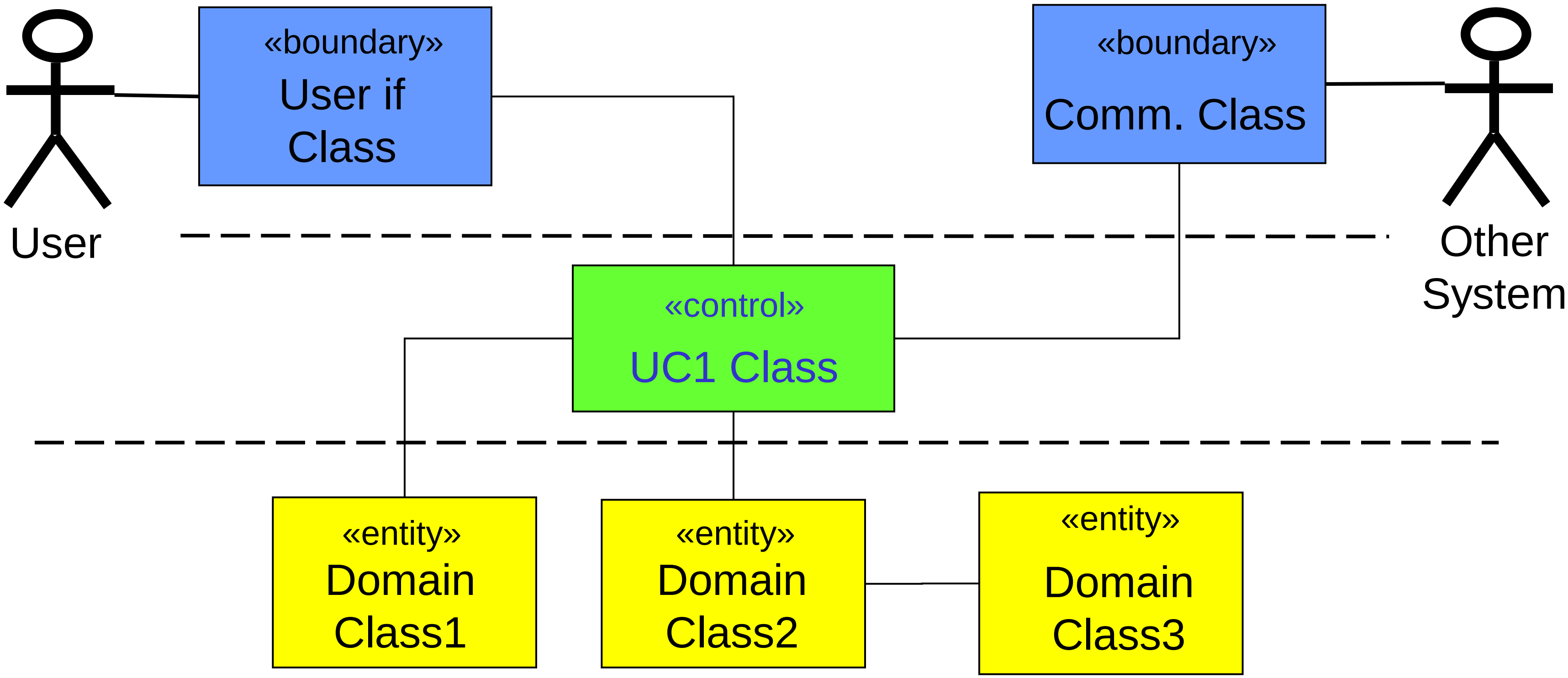
State Actions

- **entry** / action list
- **exit** / action list
- events / action list
- **do** / **activity** list
- **defer** / event list



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Analysis Classes and State Machines



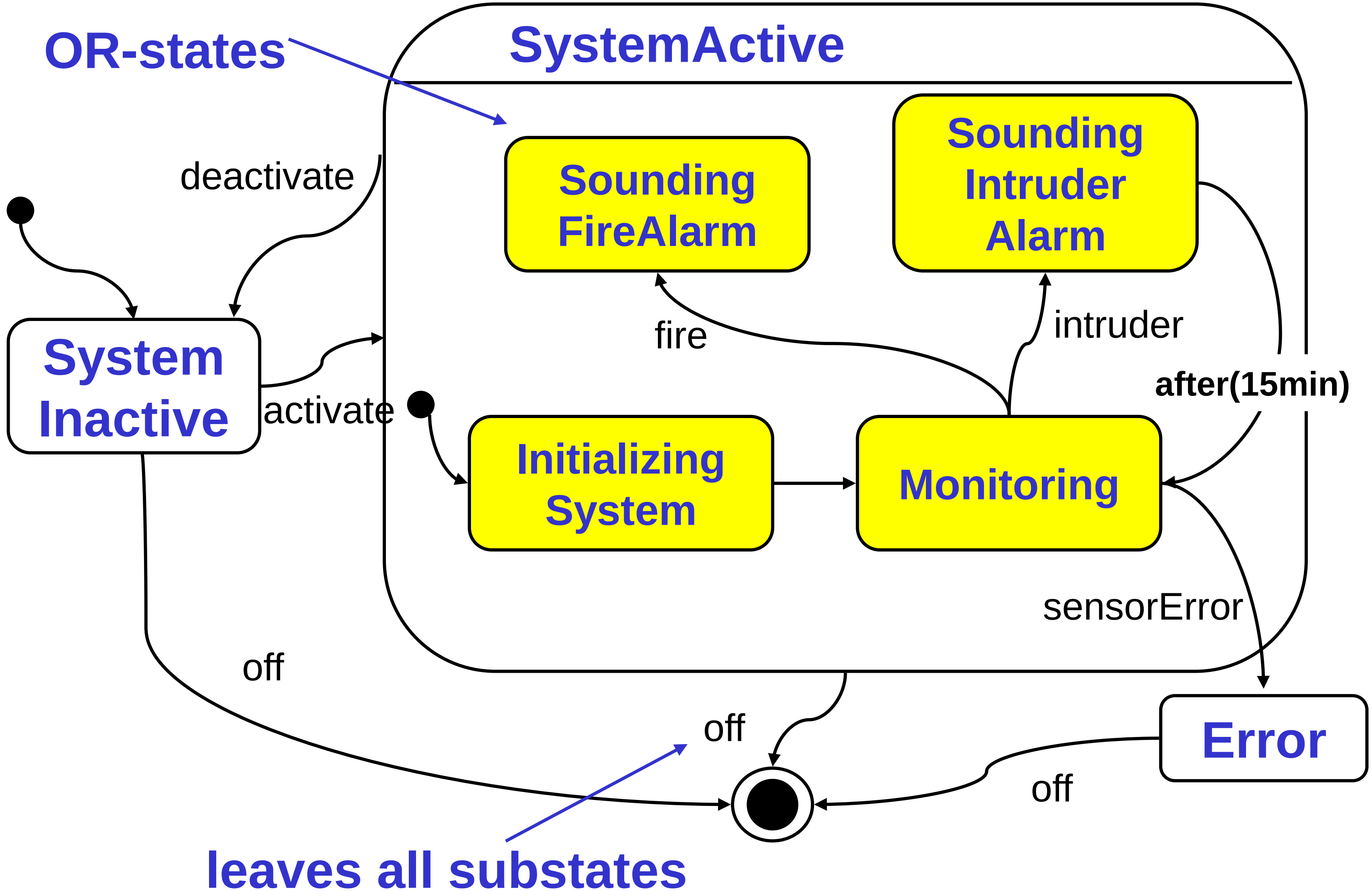
Boundary and control classes are typical candidates for state machine behavior
Entity or Domain classes are candidates for state machines, describing object life cycles

State Decomposition (OR-AND states)

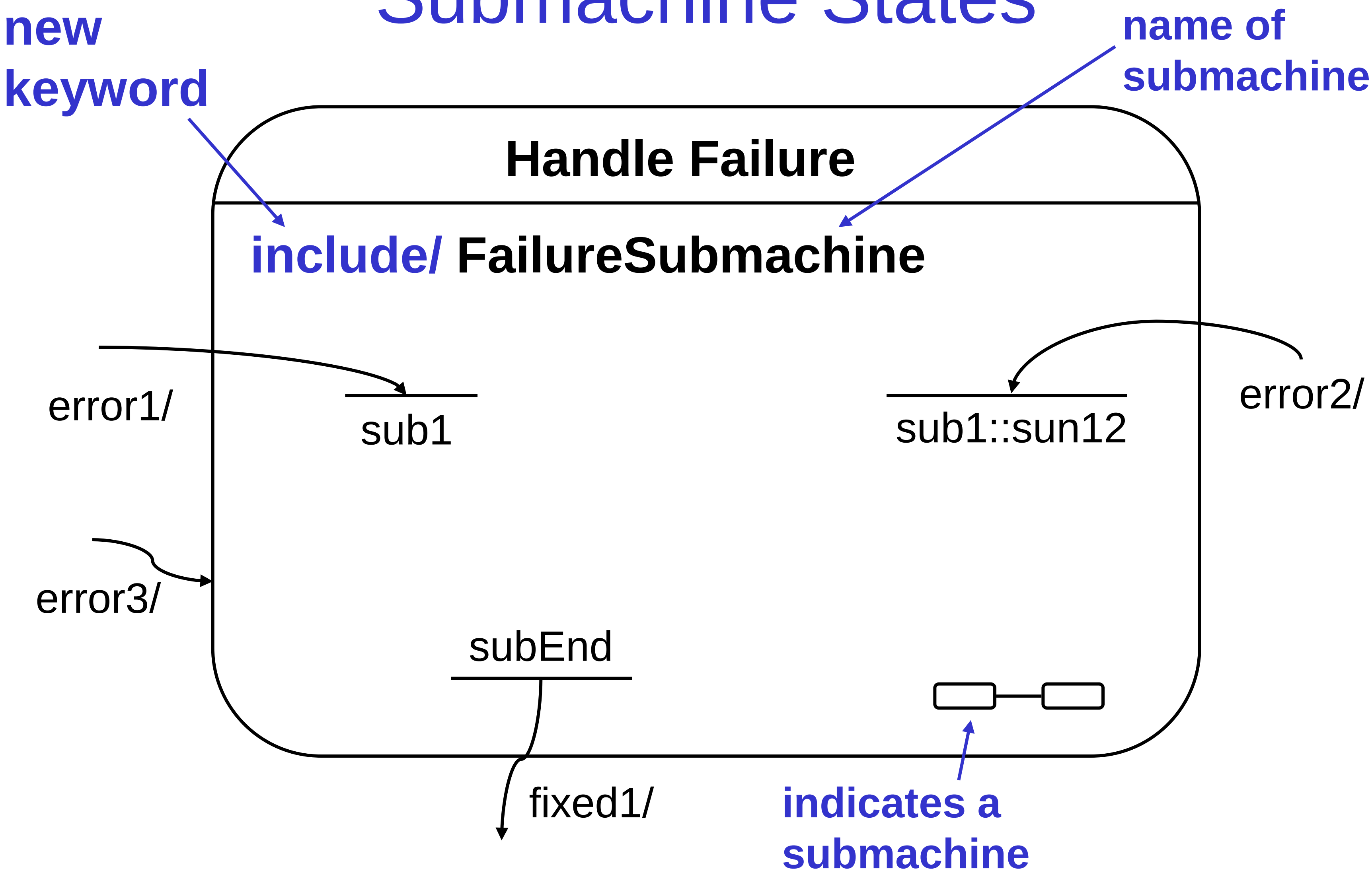
States may be decomposed into either

- **OR-states**
  - A superstate may be decomposed into any number of OR-states (substates)
  - When the object is in the superstate, it must be in exactly one of its OR-substates
  - A state machine with OR-states is called **a hierarchical state machine**
- **AND-states**
  - A superstate may be decomposed into any number of AND-states (independent behavior)
  - When in the containing superstate, the object must be in every active AND-substate
  - Orthogonal regions - shown with dashed lines

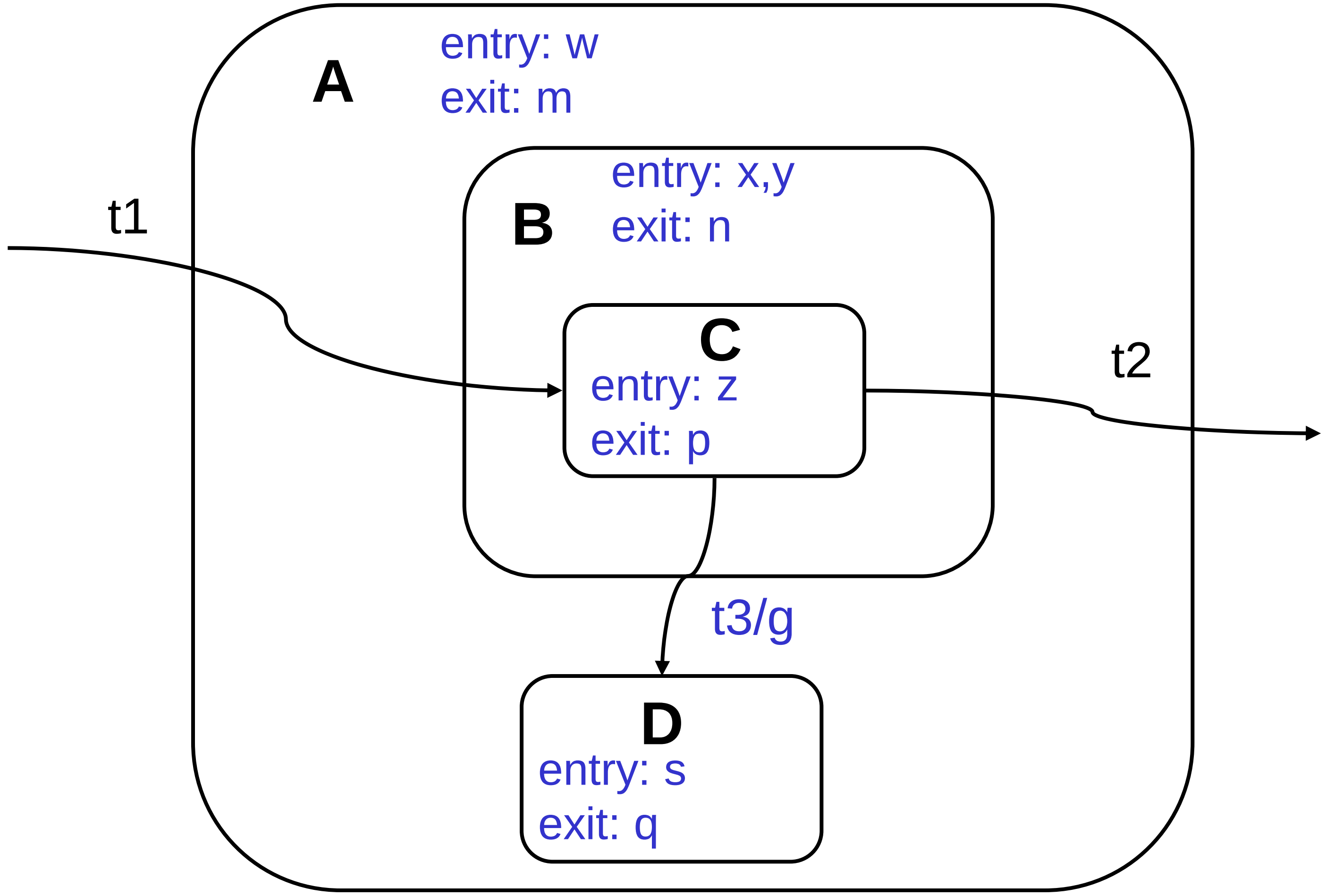
Hierarchic State Machine - Example



Submachine States

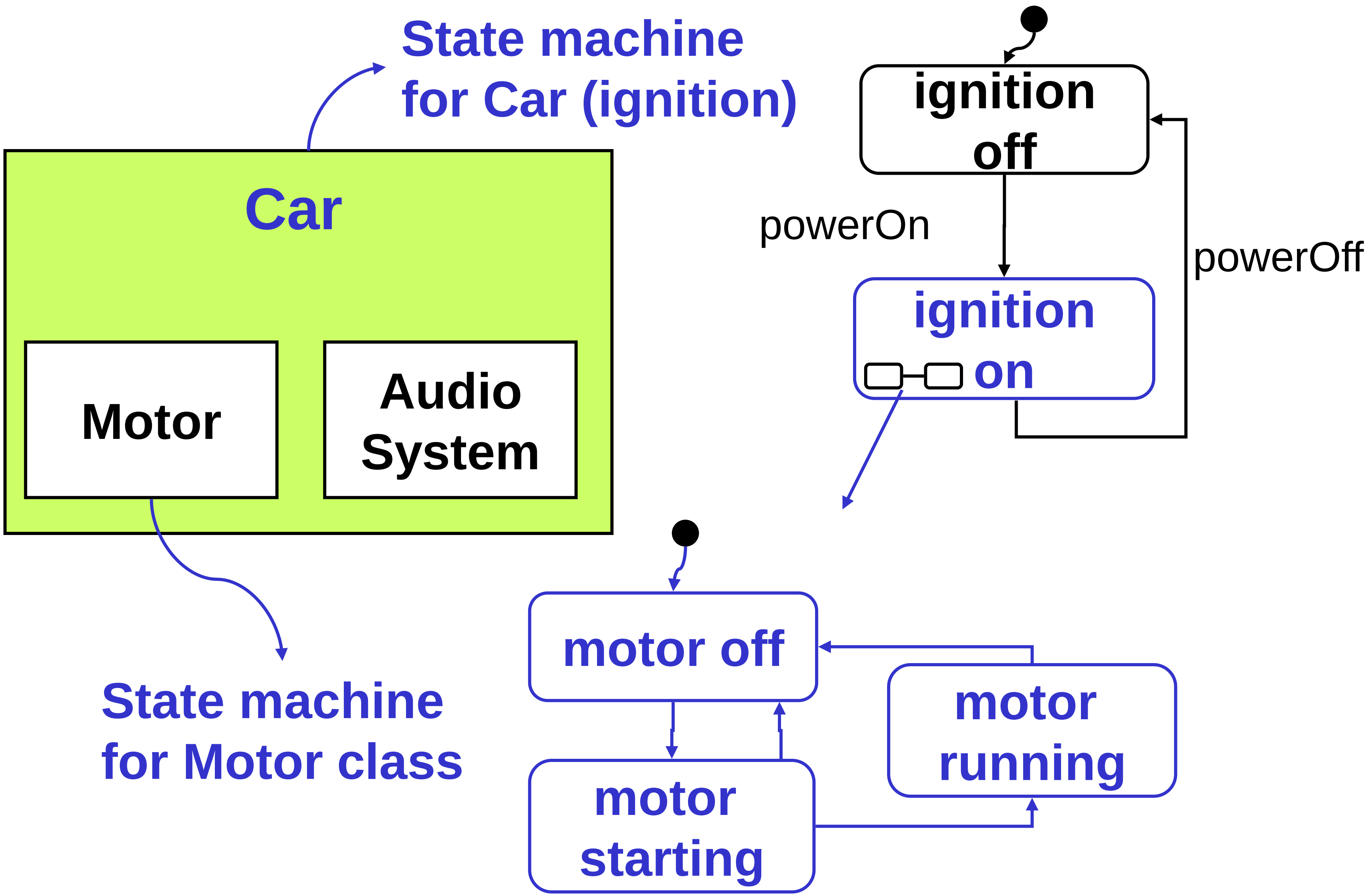


Actions and Nested States

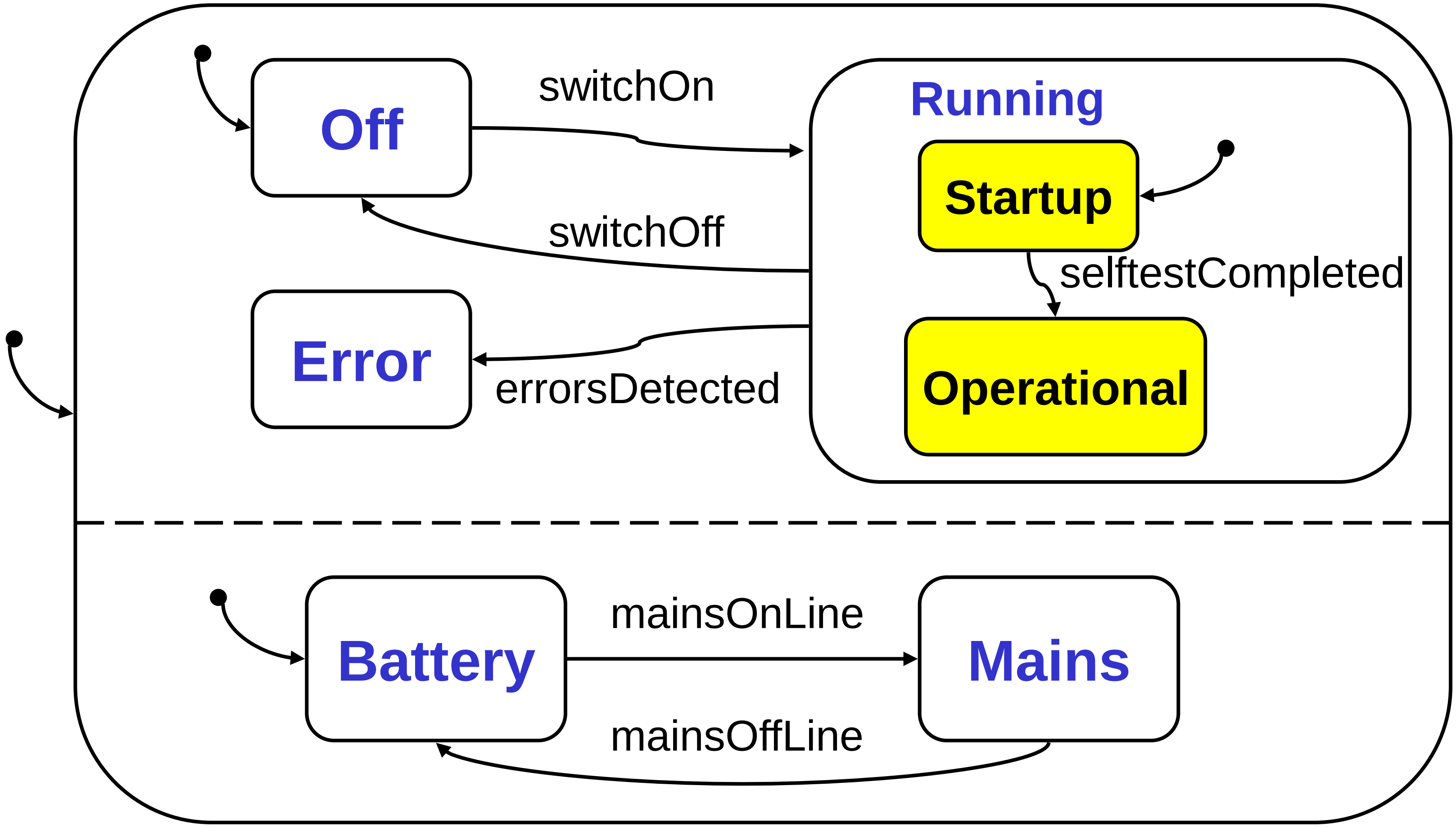


t1 => w,x,y,z **entry actions in the nesting order**
t2 => p,n,m **exit actions in the reverse nesting order**
t3=> p,n,g,s

Hierarchical Control

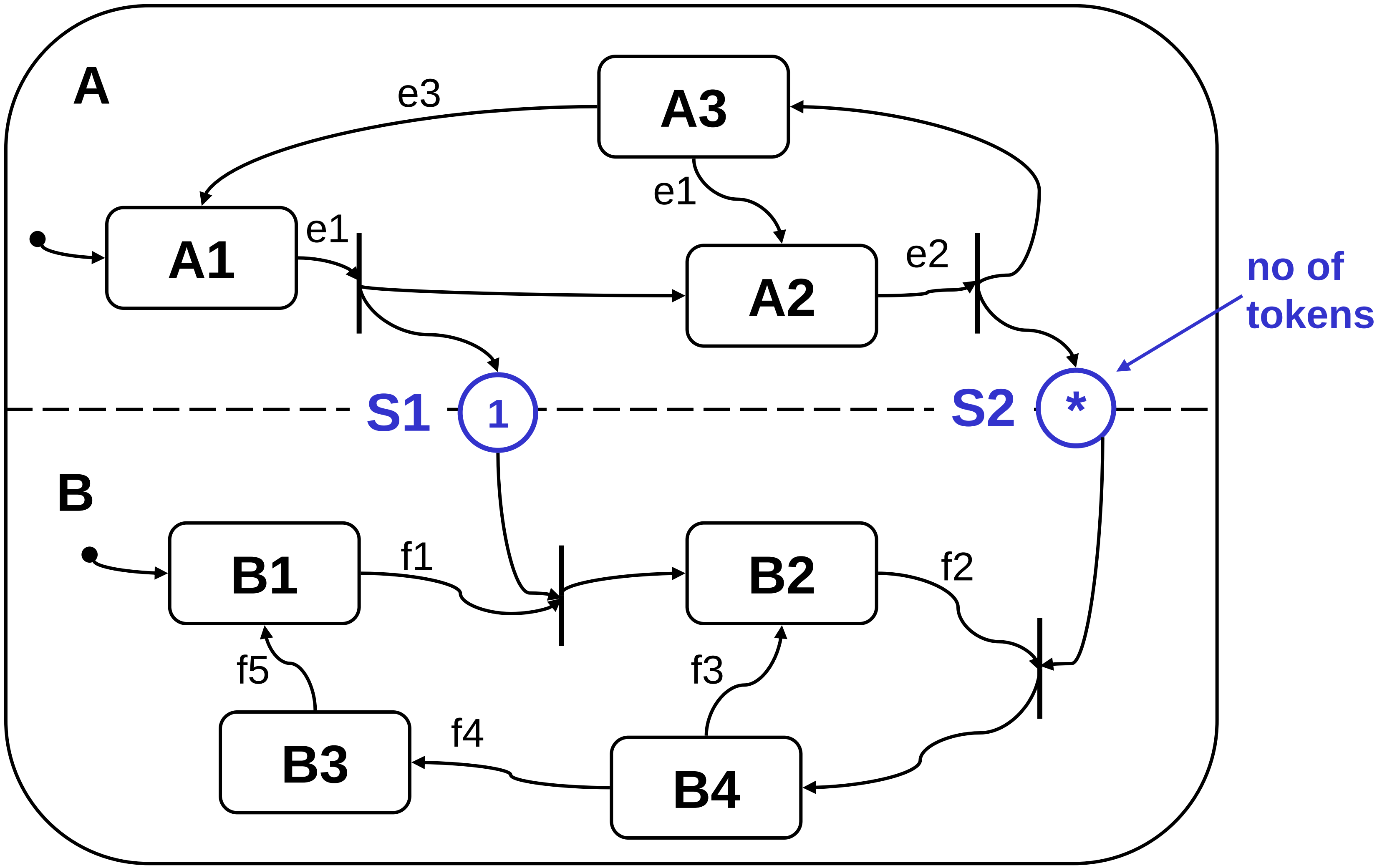


Concurrent States - Example

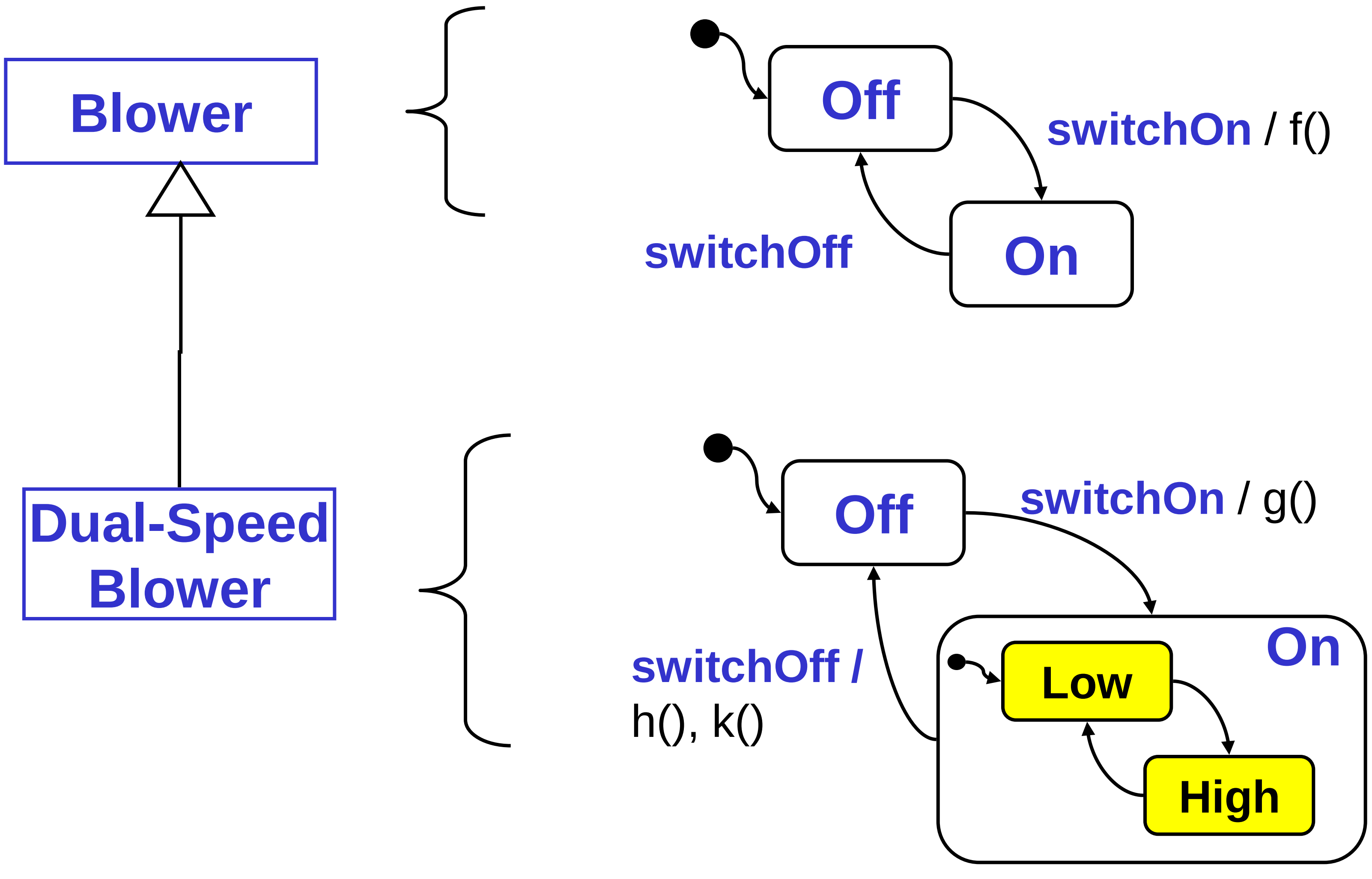


Could be generic pattern for many embedded systems

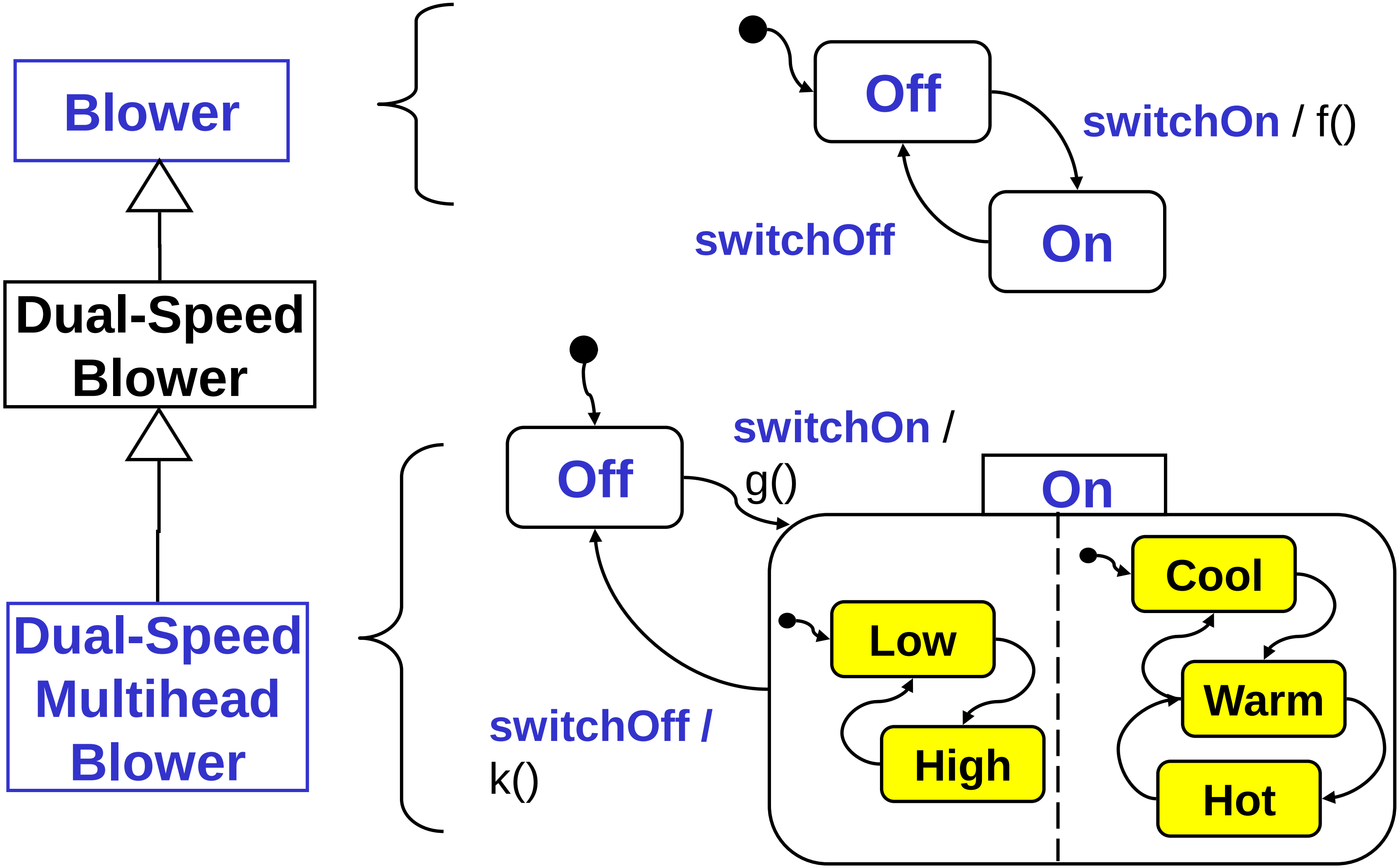
Synch States (S1,S2)



Inherited State Models (1)



Inherited State Models (2)

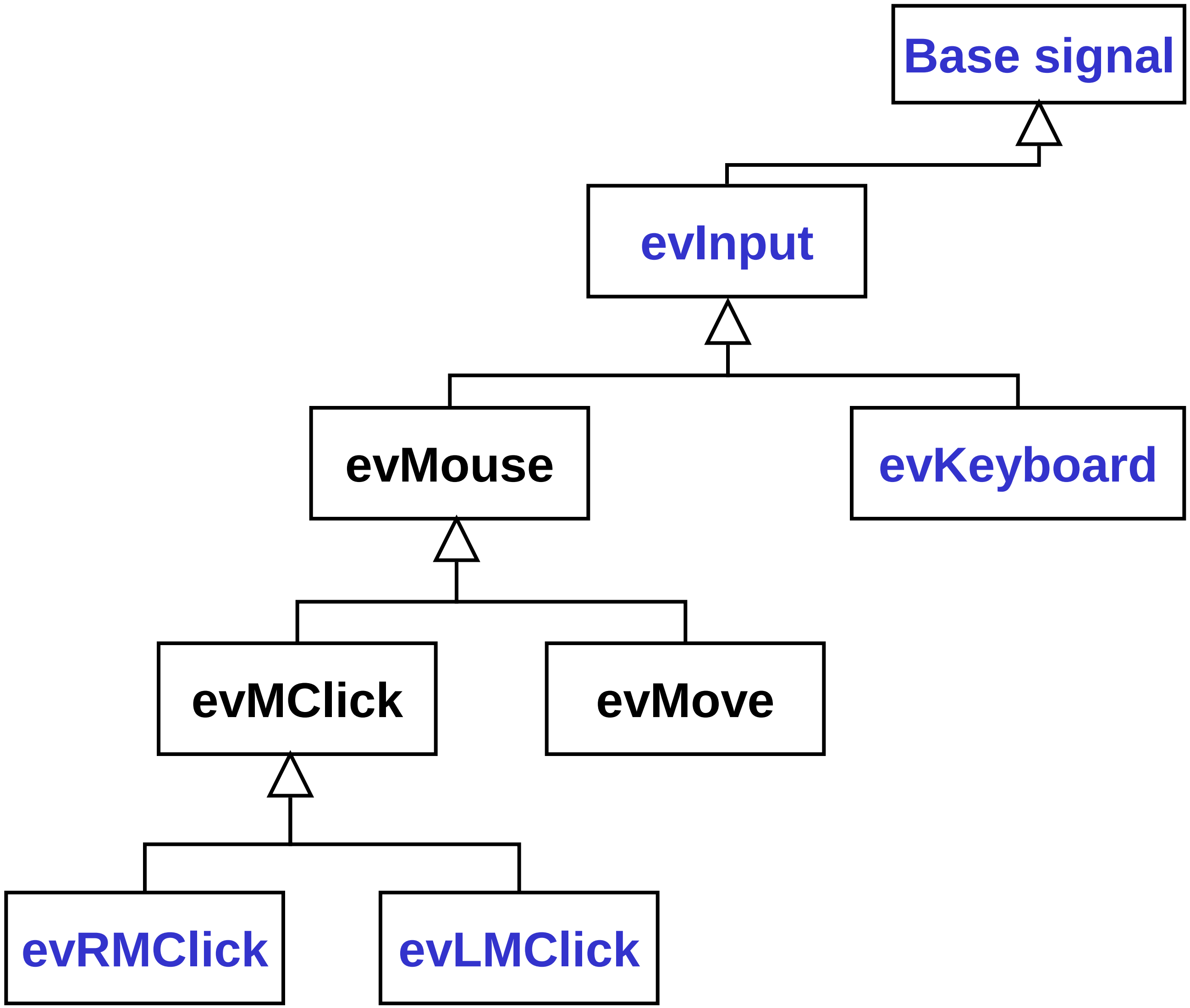


Rules for Inherited State Models

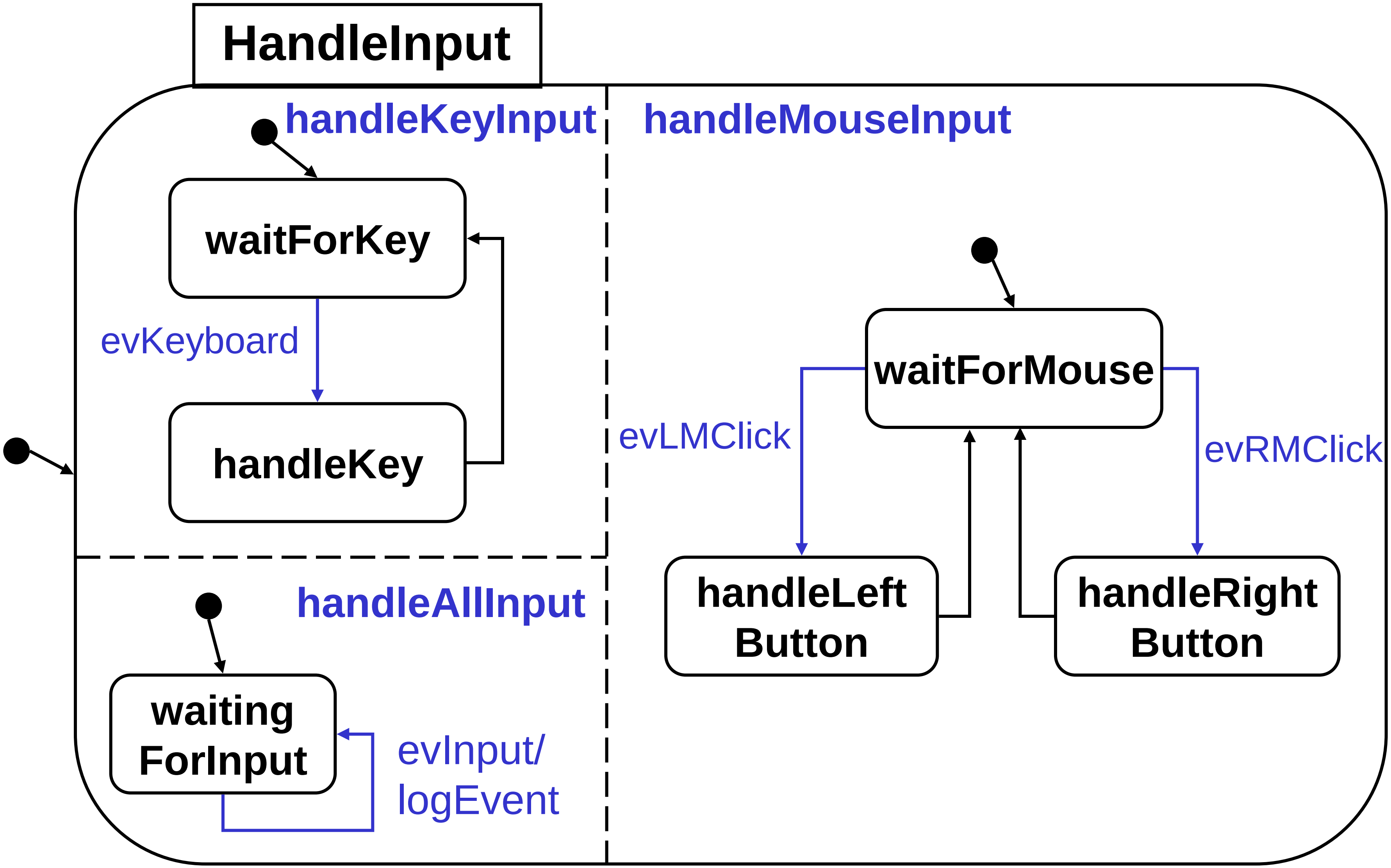
- **New states and transitions** must be freely added in the child class, including substates
- States and transitions in the parent cannot be deleted
- **Action and activity lists may be changed**
- Action and activities may be **specialized**
- Substates must not alter their superstate
- Transitions may be retargeted to diff. states
- **Orthogonal components** may be added

Helps to obtain compliance with Liskov's Substitution Principle

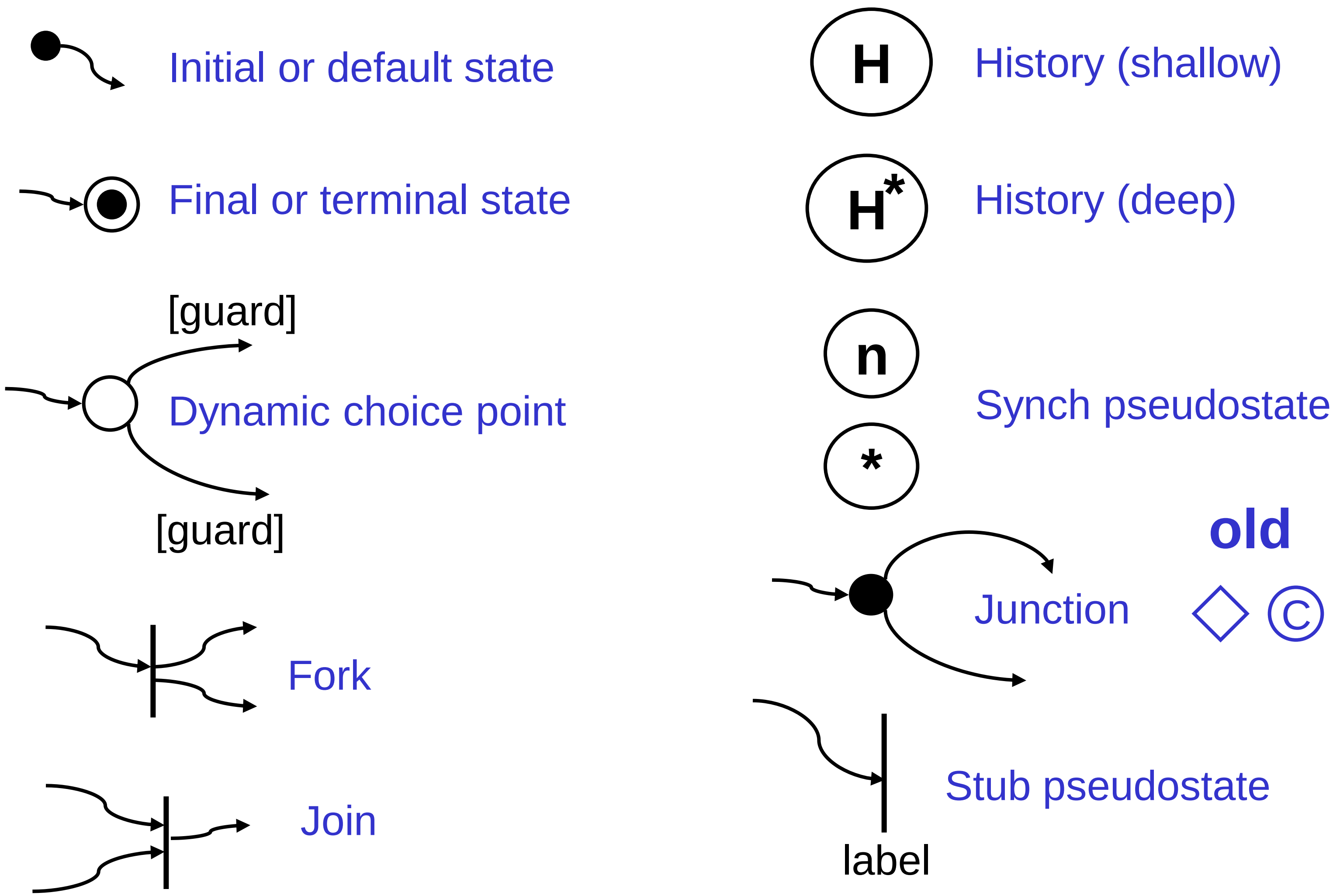
Event Hierarchy



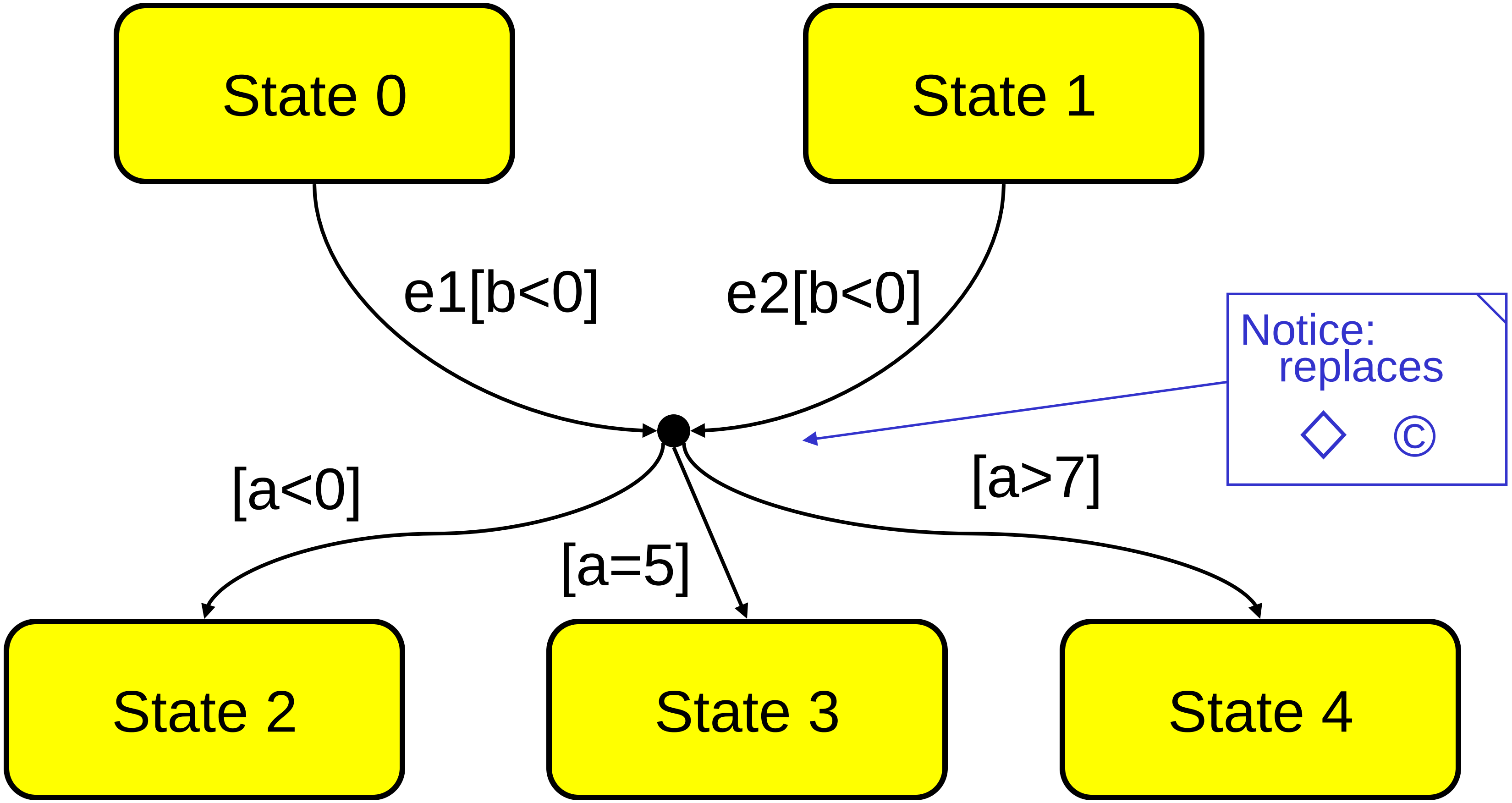
Using an Event Hierarchy



Pseudo State Notations

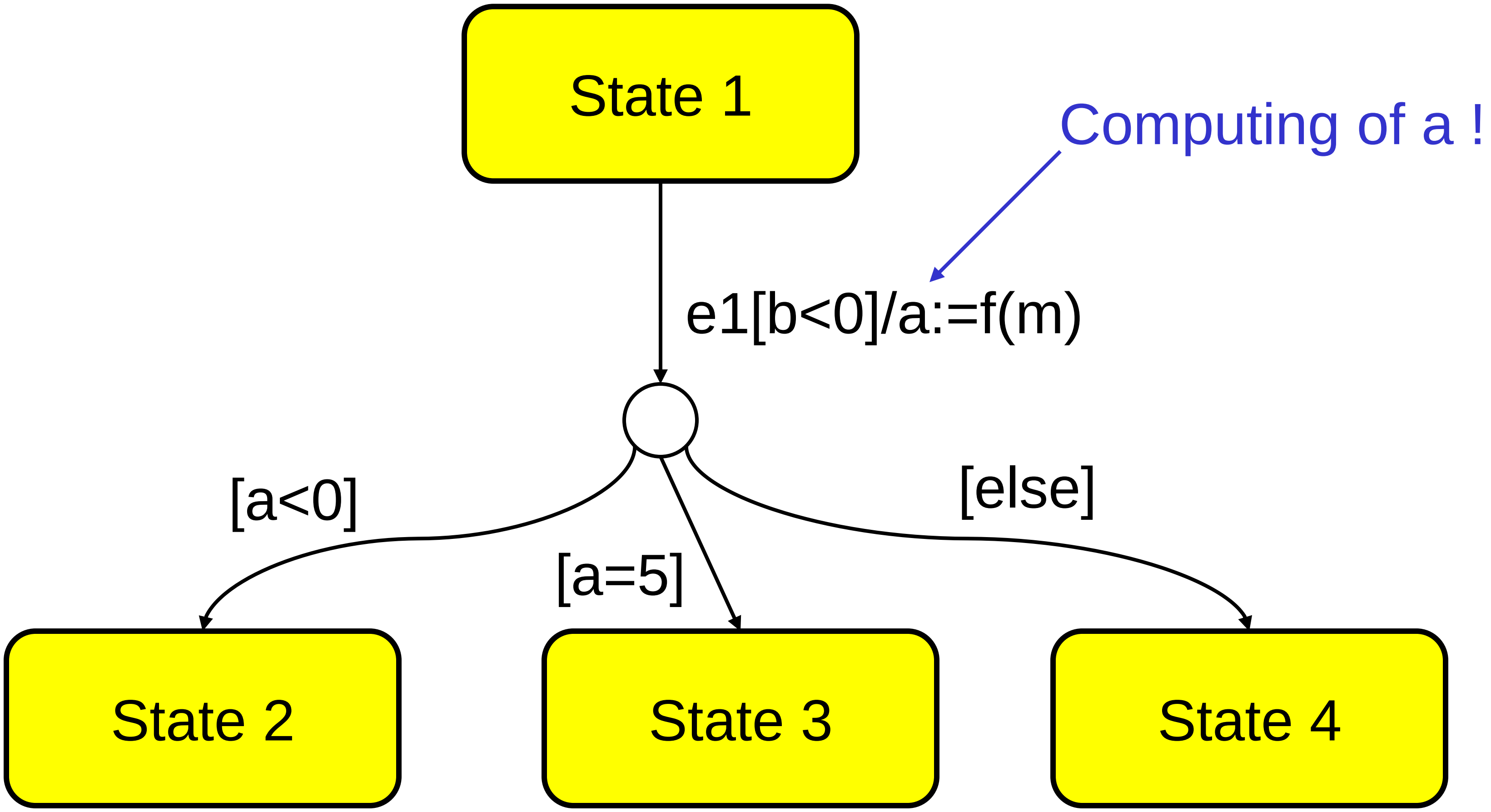


Junction Point

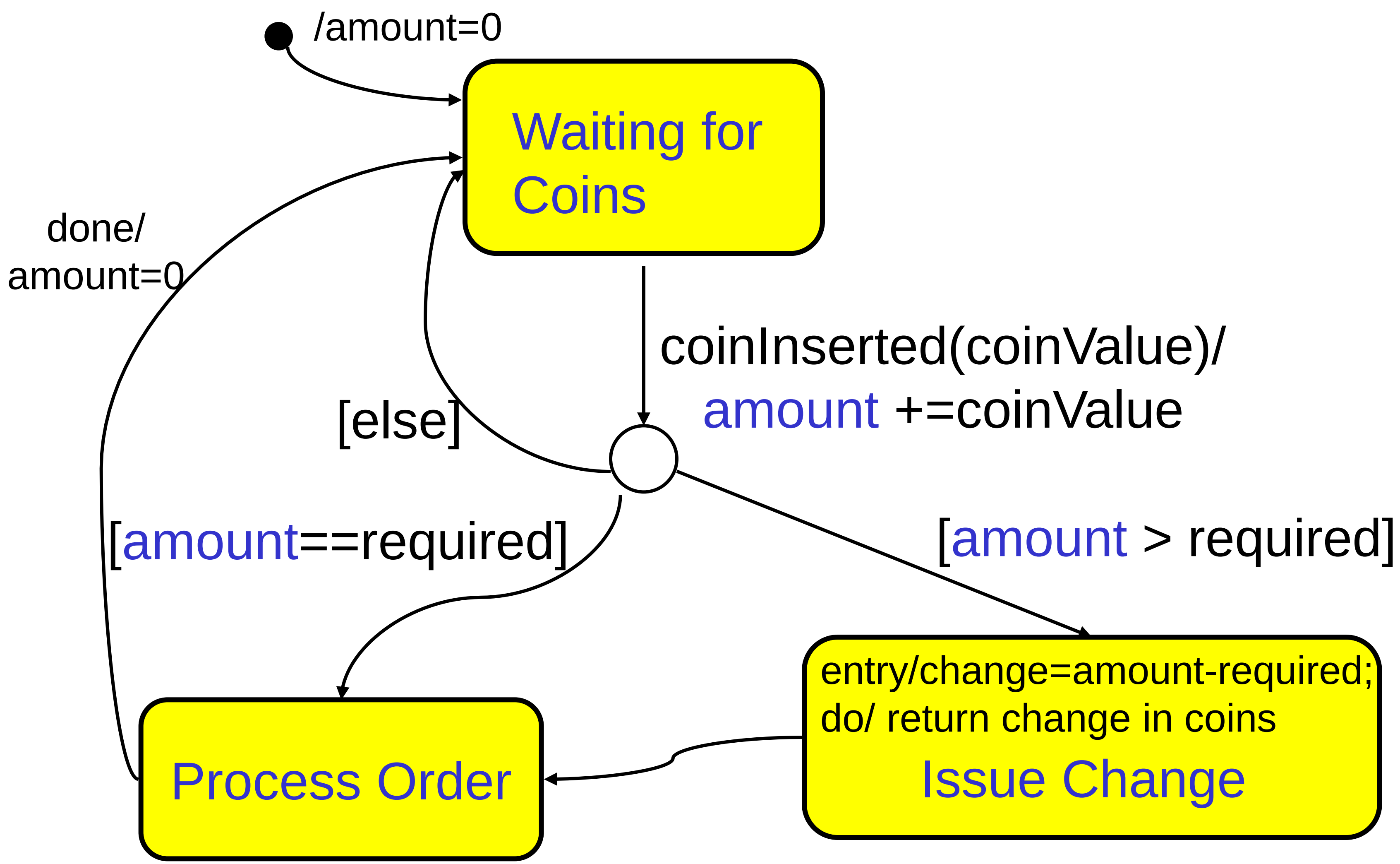


b<0 and one of the a guards must be true for a state shift to state 2, 3 or 4.

Dynamic Choice Point



Dynamic Choice Point - Example

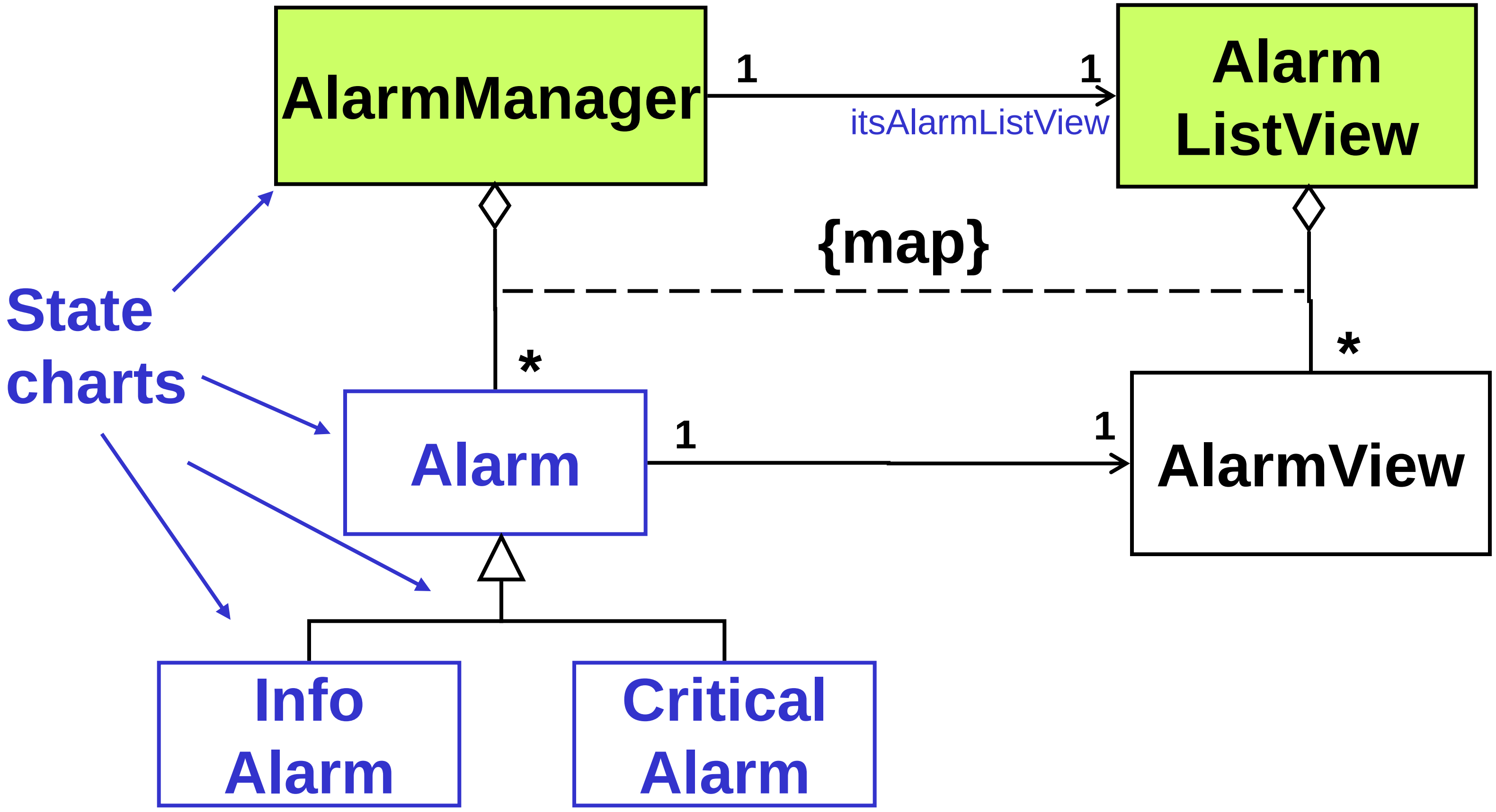


Common State Chart Mistakes

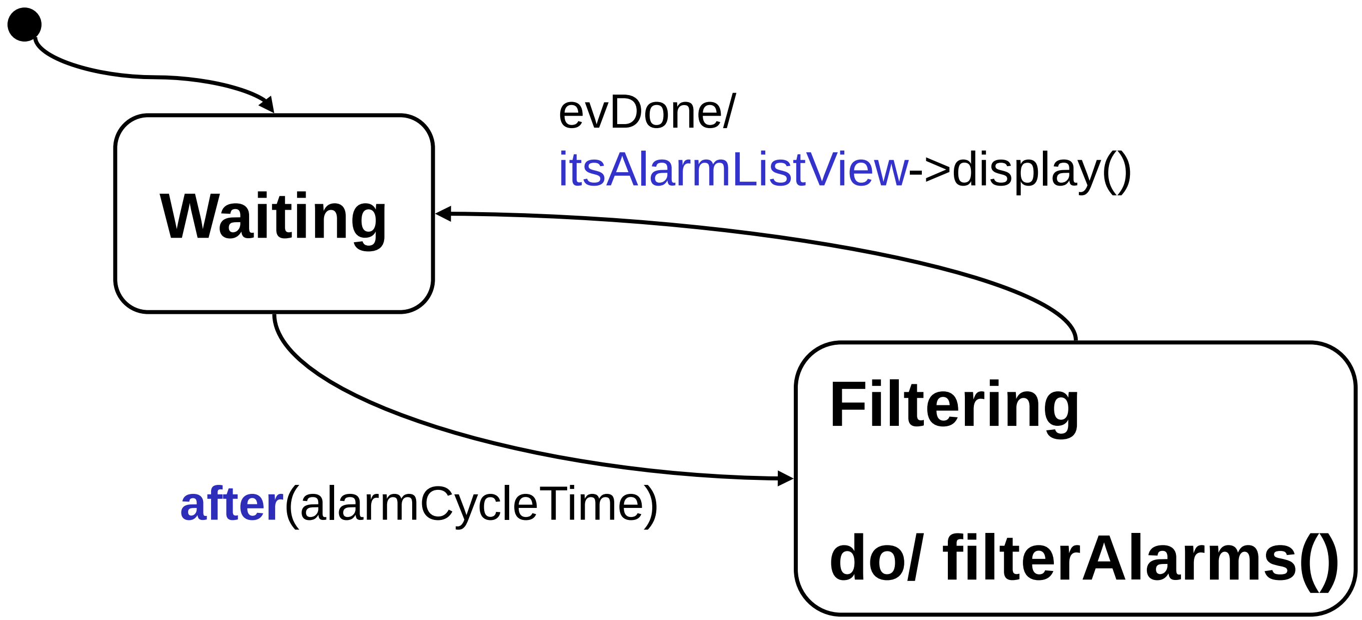
- No initial state within a context
- Conflicting transitions
- Overlapping guards
- Guards on completion-event transitions
- Guards on initial transitions
- Using action side effects
- Synch pseudostate within a single orthogonal region
- Breaking substitutability in subclasses

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

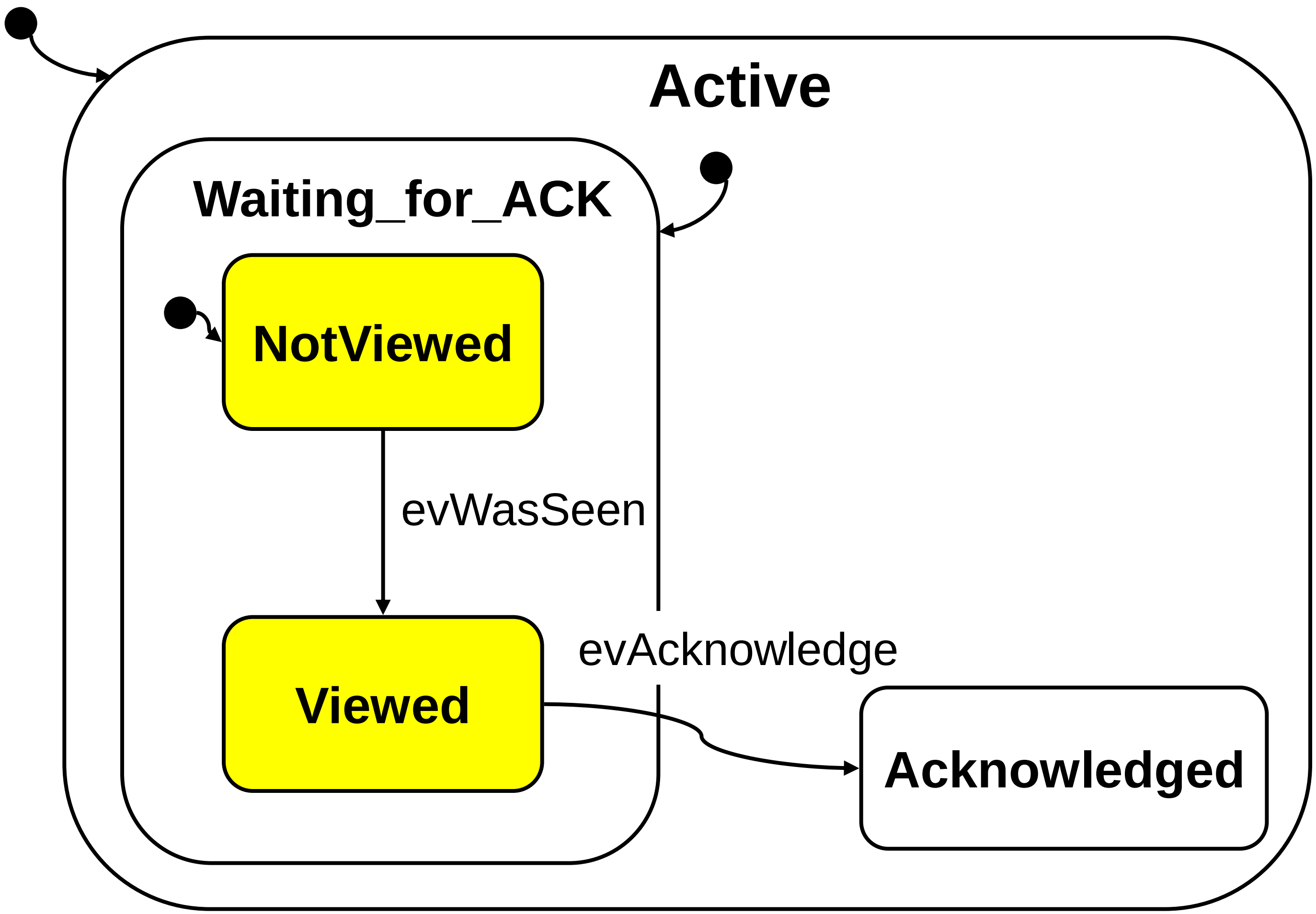
AACTS Alarm System



AlarmManager State Diagram

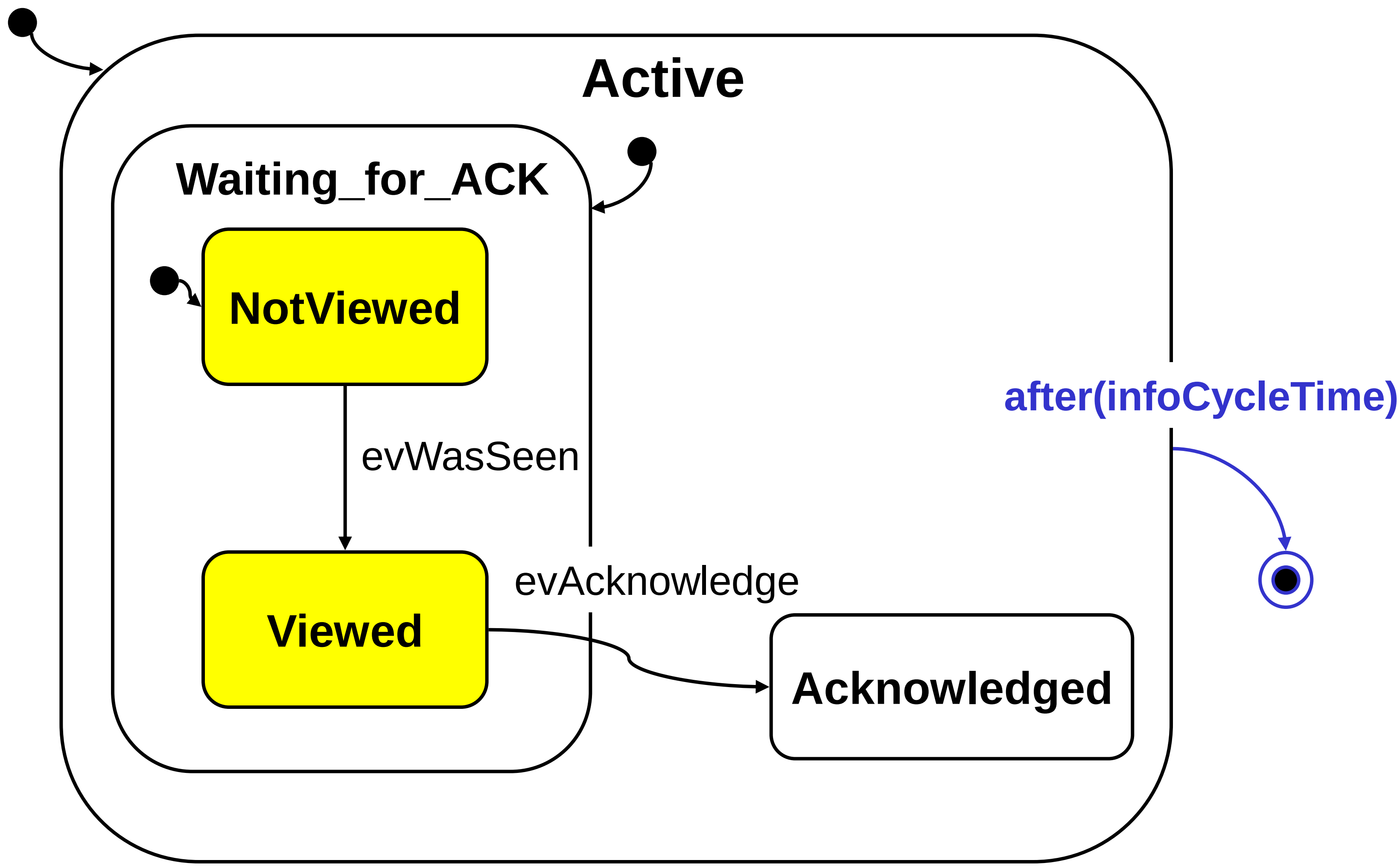


Alarm State Diagram



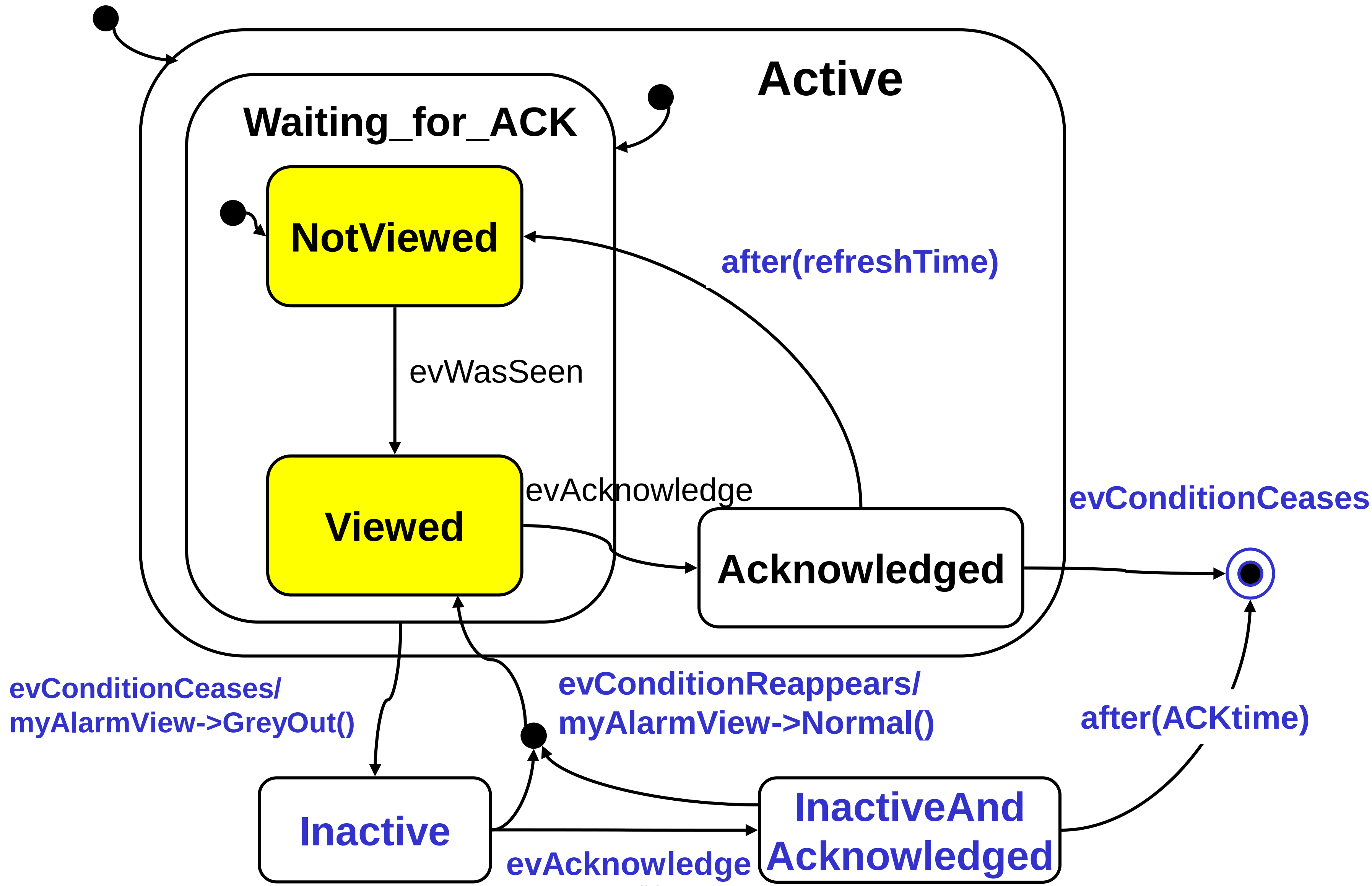
Handles caution alarms that must be explicit ack.

InfoAlarm State Diagram



Handles info alarms that are displayed for a short period

CriticalAlarm State Diagram



State-Event Tables

State-event tables is unfortunately not a part of UML, but can be used as a very valuable supplement for validation and test

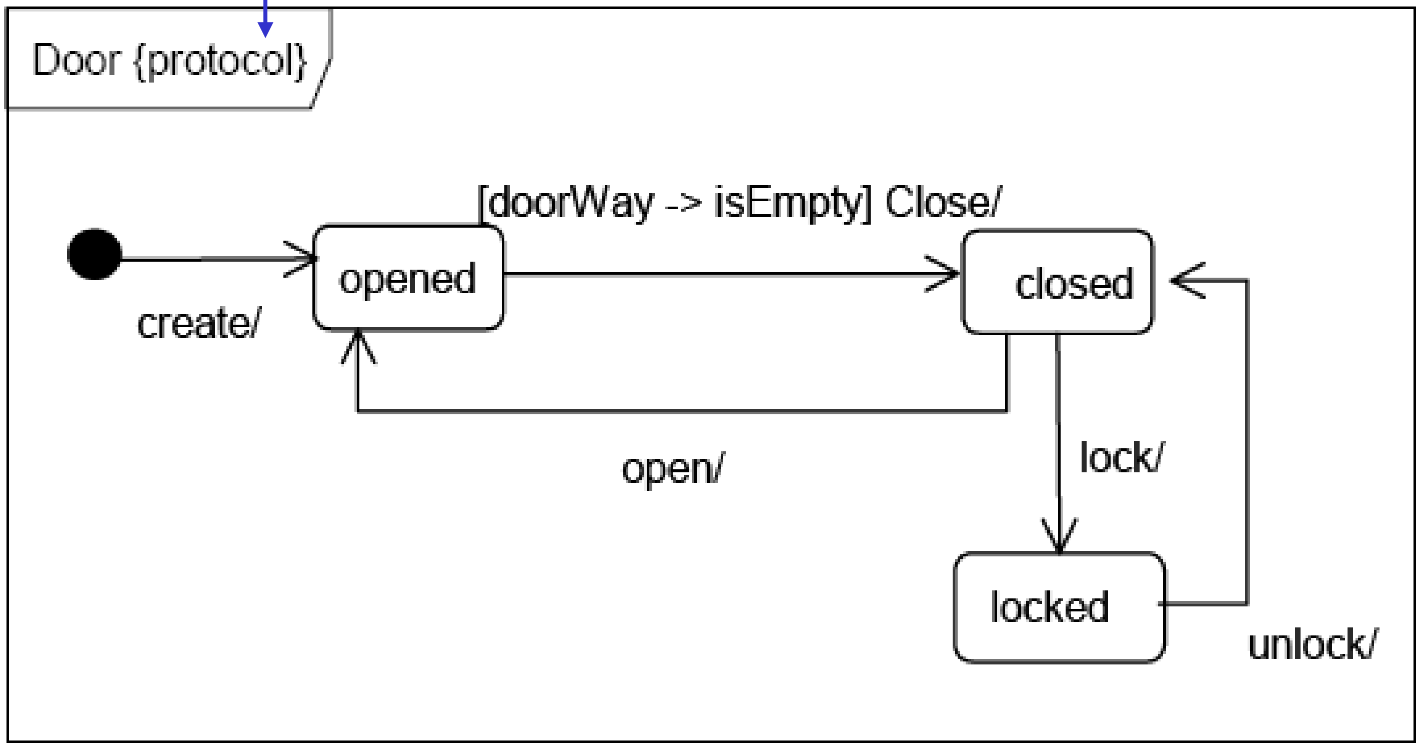
	Event 1	Event 2	Event 3	Event 4
State 0	New state / action	- / -		
State 1		State 3 / action17		
State 2	State 1/ -			
State 3				

Protocol State Machines (UML 2.0)

- A Protocol State Machine is always defined in the context of a classifier
- It specifies which operations of the classifier can be called in which state and under which condition
  - specifying the allowed call sequences of the classifiers operations

Protocol State Machine Example

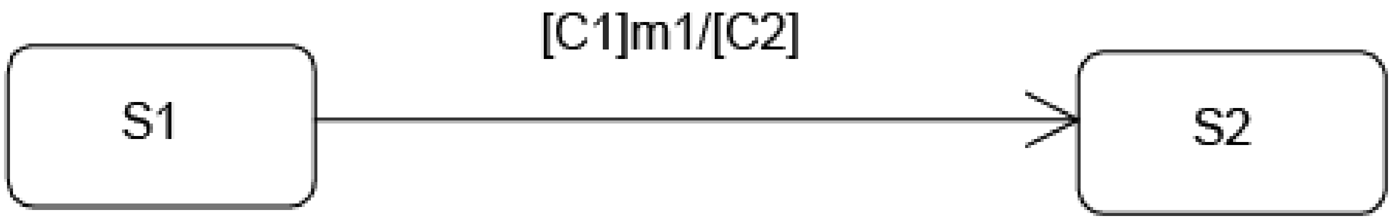
Notice!



Notation:

[preconditon] event / [postcondition]

Equivalences to Pre and Post Conditions



1. **the operation m1() can be called on an instance, when it is in state S1 under the condition C1 (the pre condition)**
2. **the final state S2 must be reached under the, condition C2 (the post condition)**

Protocol state machines provide a global overview of the classifier protocol usage, in a simple formal representation

Rail-car Case – see the following Article

Cover Feature

Executable Object Modeling with Statecharts

Statecharts, popular for modeling system behavior in the structural analysis paradigm, are part of a fully executable language set for modeling object-oriented systems. The languages form the core of the emerging Unified Modeling Language.

David Harel
The Weizmann
Institute of
Science and
i-Logix

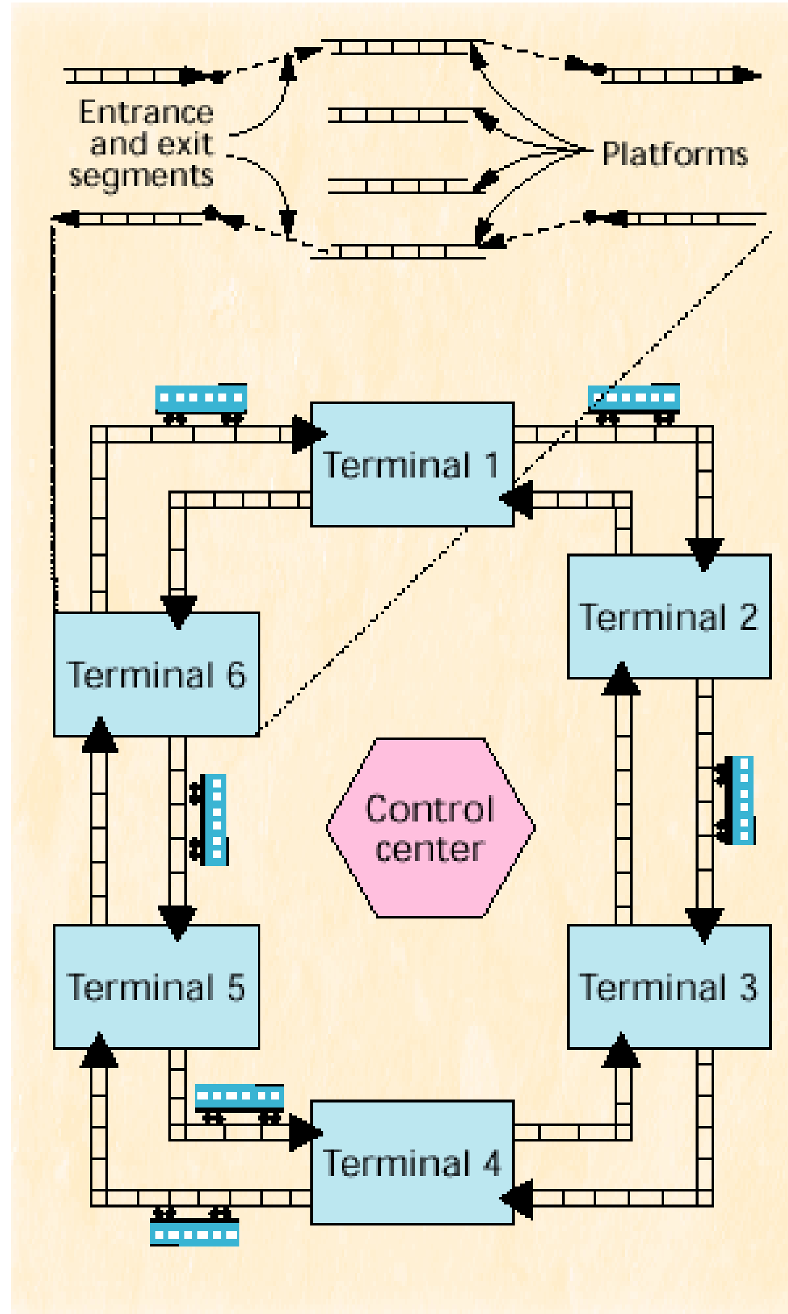
Eran Gery
i-Logix

IEEE Computer, July 1997

Automated Rail-Car System

A car can be requested at a terminal by pushing a destination button

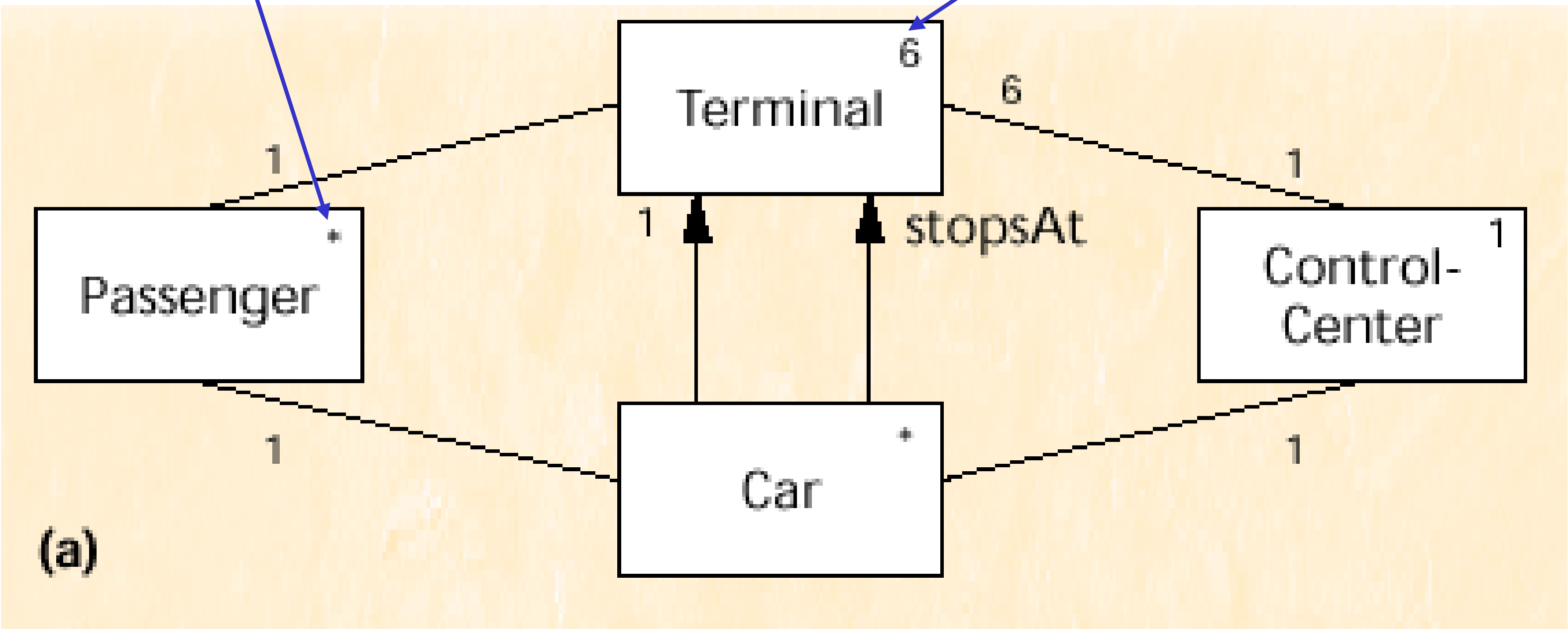
A car has also a destination board



Class Diagram (System Level)

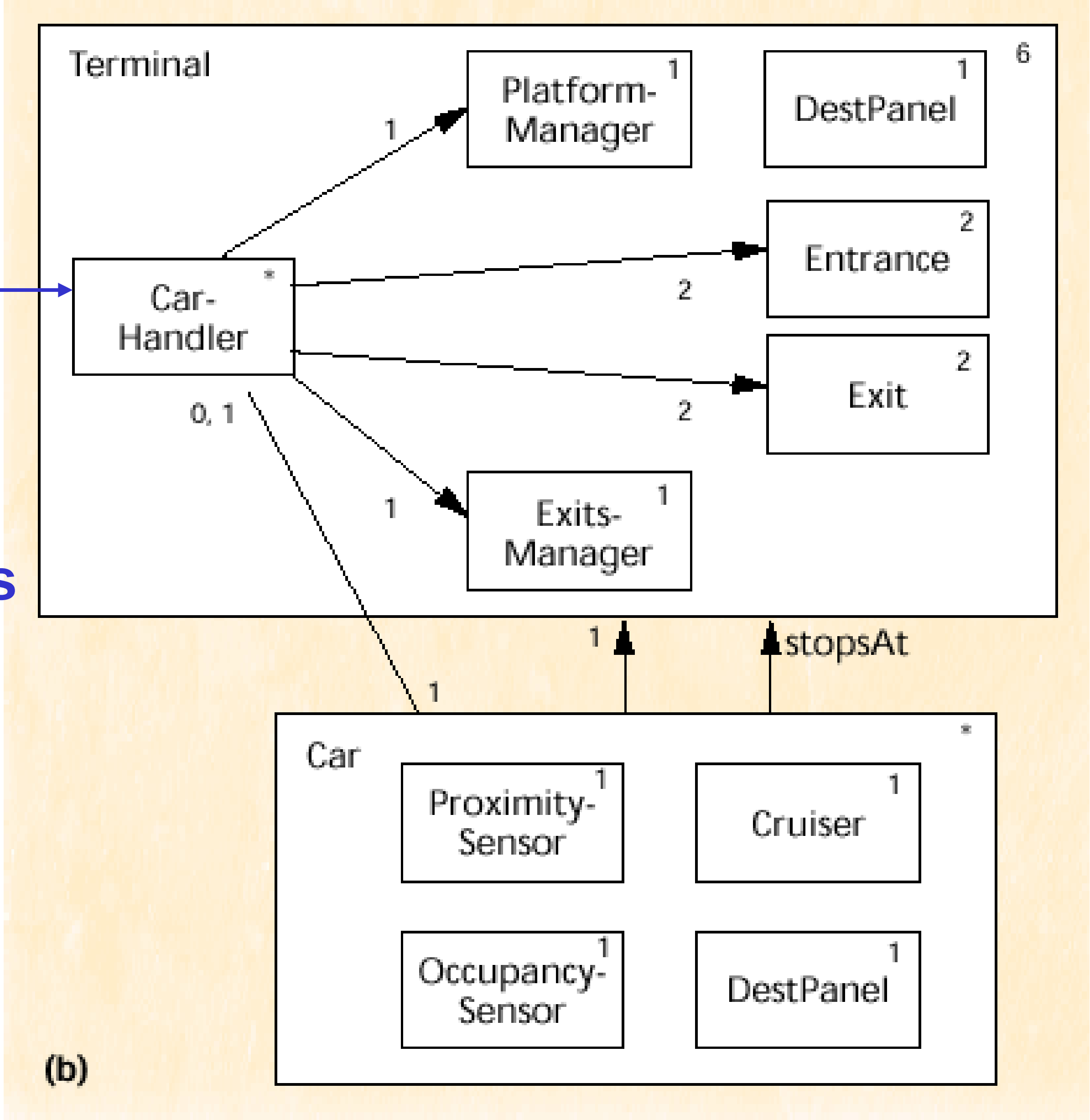
**unbounded
no of instances**

**bounded
no of instances**



Terminal and Car are composite structured classes (see next page)

Created (dynamic) to control a single car at a terminal, deleted when the car leaves



Terminal and Control Center State Diagrams (1)

- They are both modeless, containing reactions and forwarding only:
- **Terminal class behavior:**
  - Reactions:
    - **destSelected/**

      if (itsCar->isEmpty())

      itsControlCenter->gen(sendCar(this))
    - **arrivReq(car,dir)/**

      new CarHandler(car,dir)
  - Forwarding:
    - delegate(clearDest, DestPanel)

Terminal and Control Center State Diagrams (2)

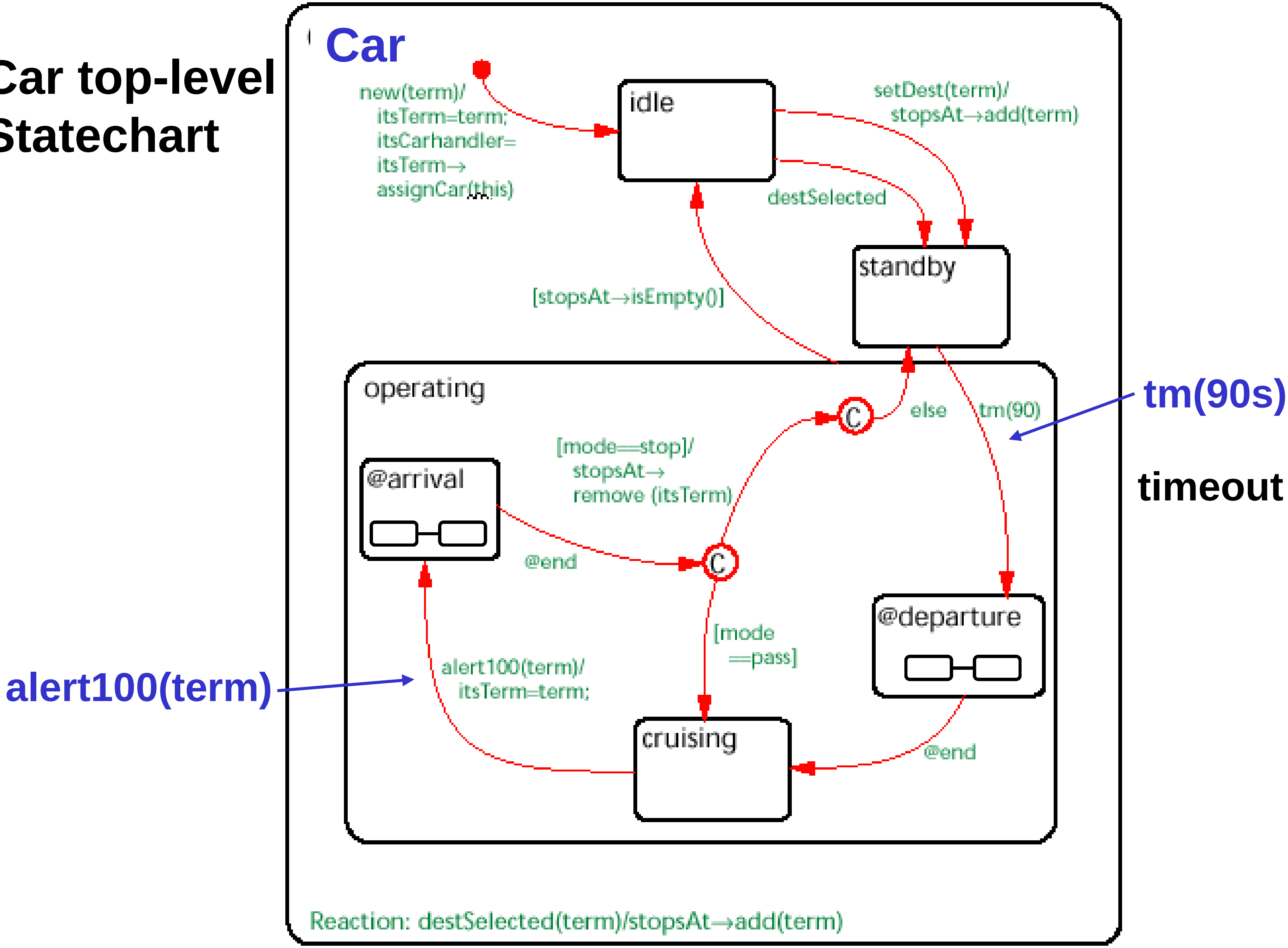
- **Control Center class behavior:**
  - Reactions:
    - sendCar/

      for each (car, itsCar)

        if (car->idle()

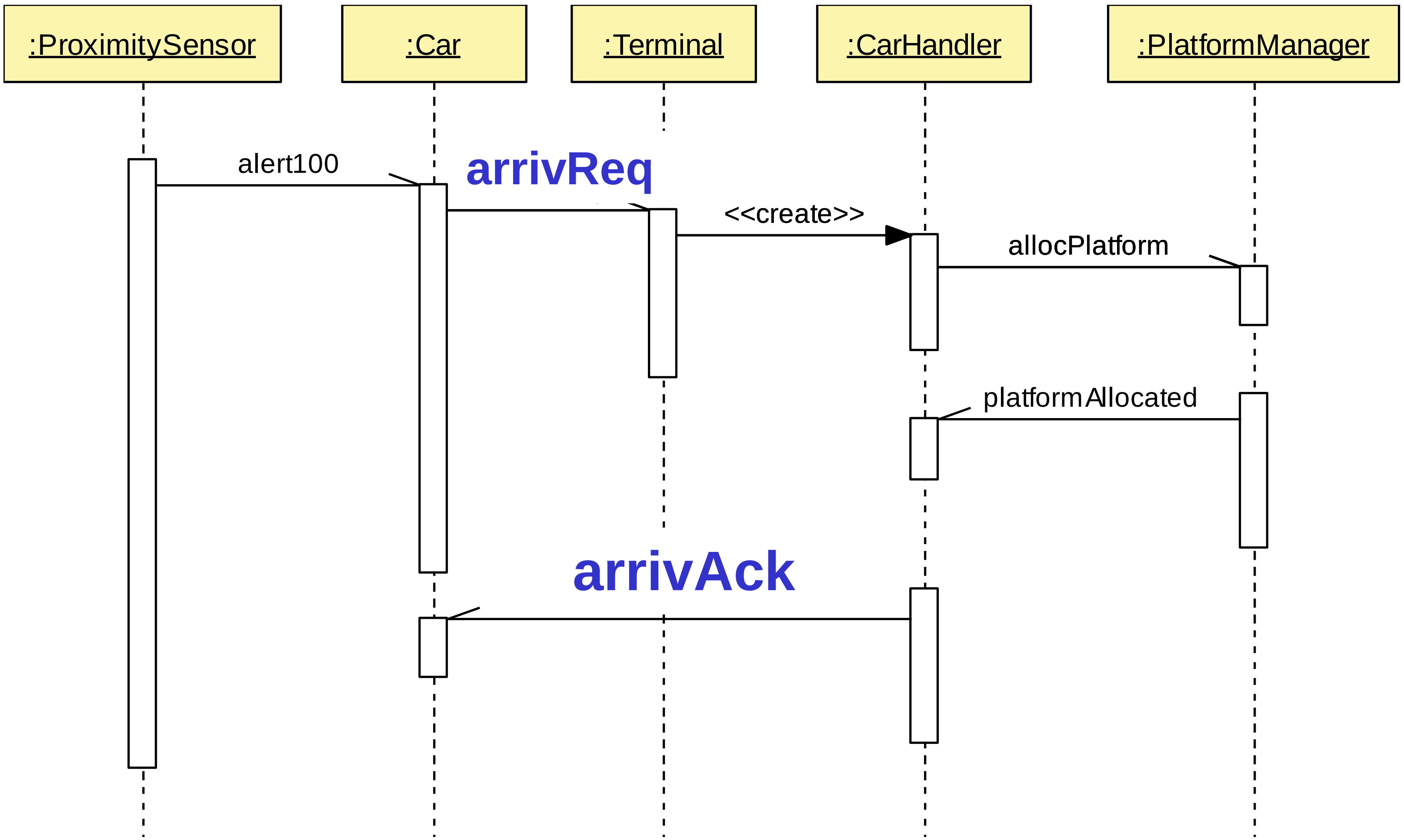
          car->gen(setDest(sendCar->term))
- These behavior specifications are necessary for executable of the model

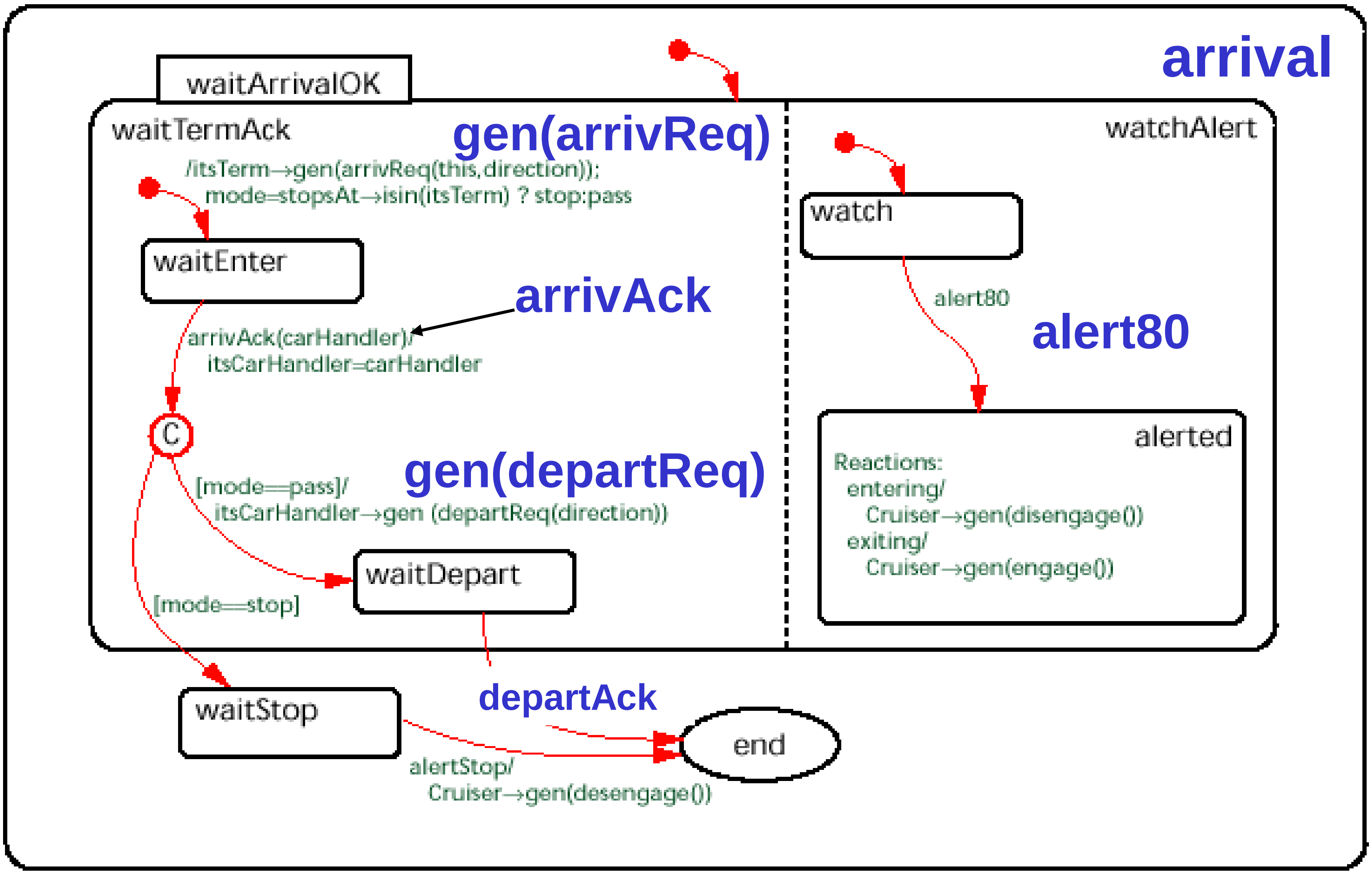
Car top-level Statechart



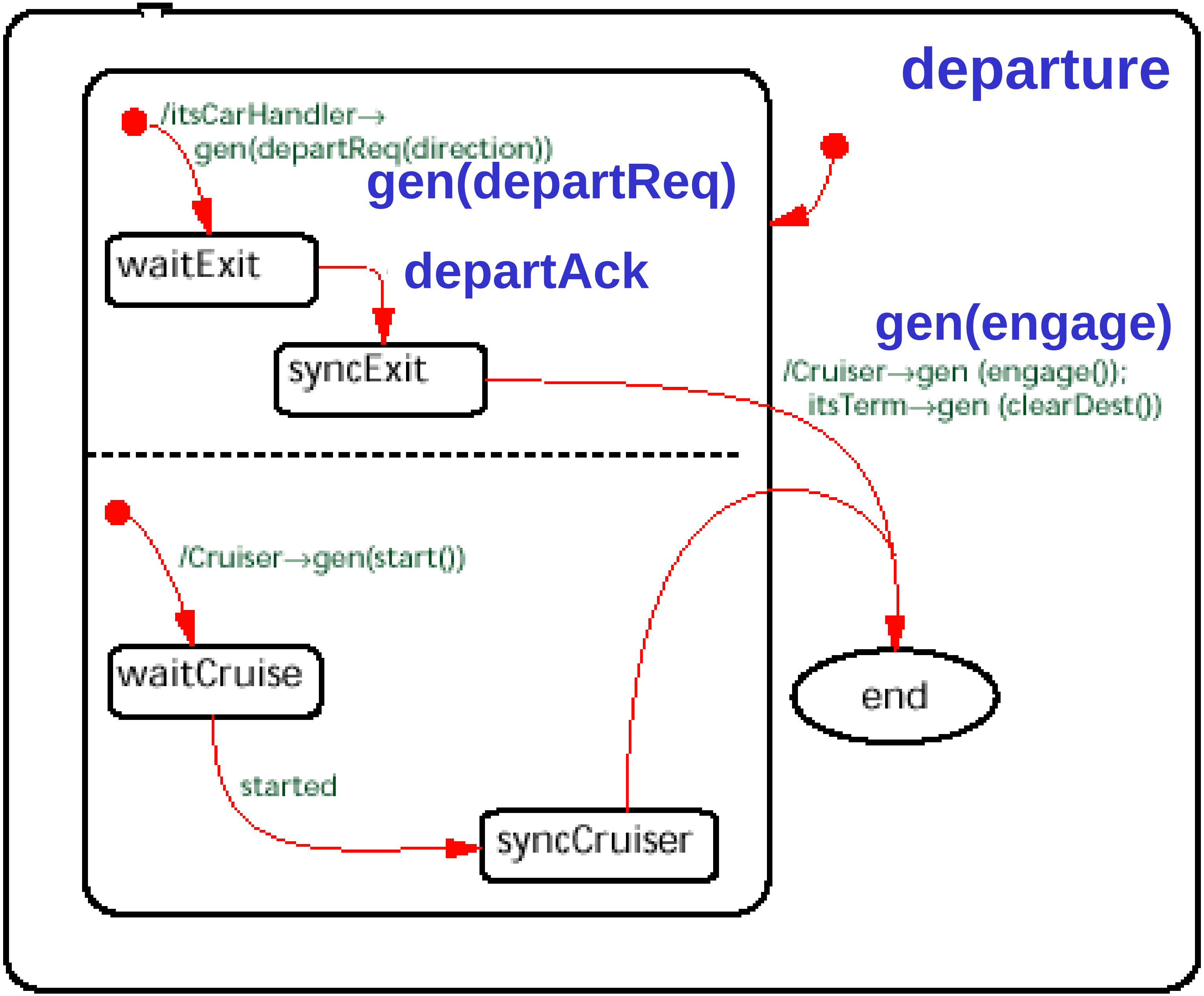
AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Sequence diagram



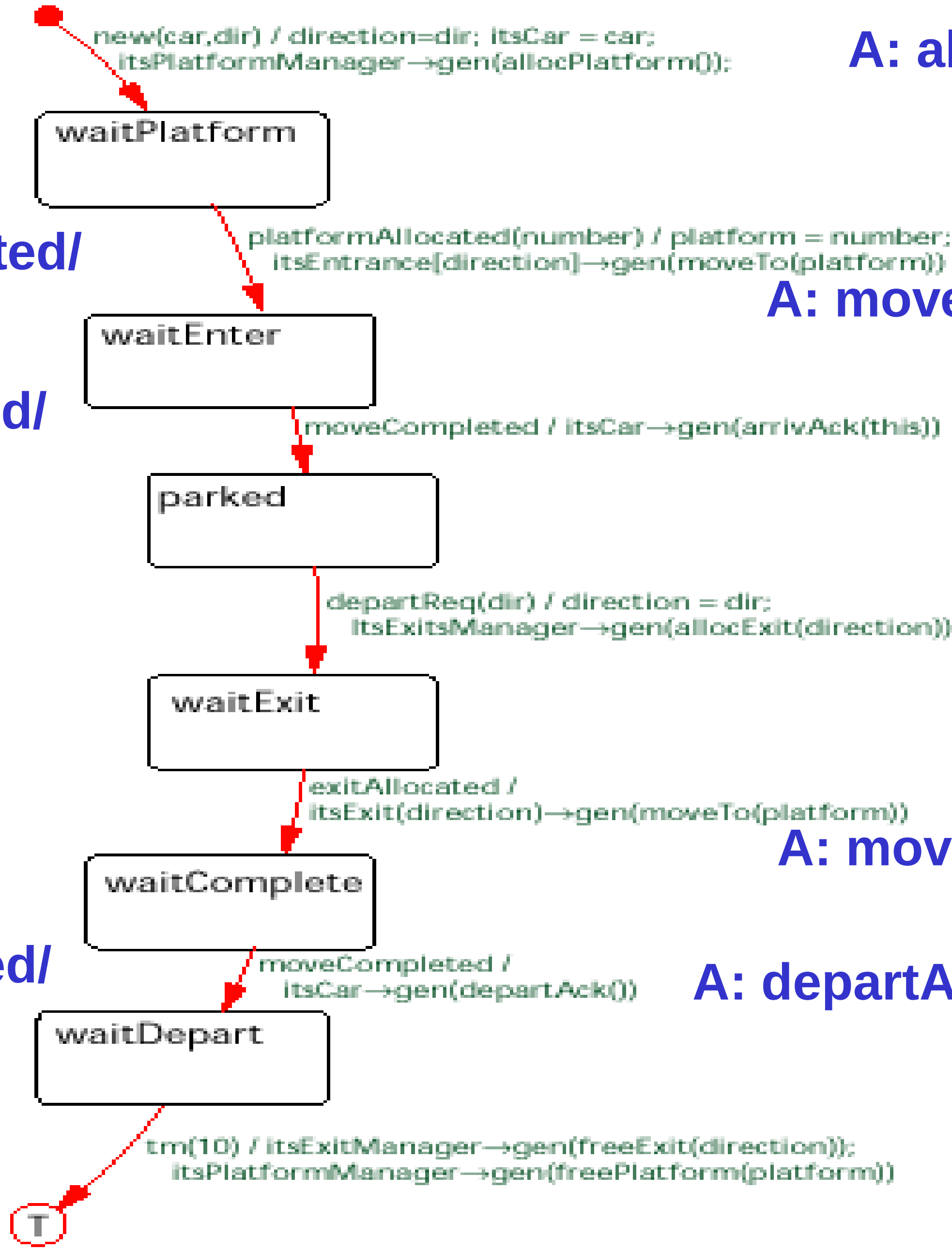

arrival
waitArrivalOK
waitTermAck
gen(arrivReq)
/itsTerm→gen(arrivReq(this,direction));
mode=stopsAt→isin(itsTerm) ? stop:pass
waitEnter
arrivAck
arrivAck(carHandler)/
itsCarHandler=carHandler
C
[mode==pass]/
itsCarHandler→gen (departReq(direction))
gen(departReq)
waitDepart
[mode==stop]
waitStop
departAck
end
alertStop/
Cruiser→gen(desengage())
watchAlert
watch
alert80
alert80
alerted
Reactions:
entering/
Cruiser→gen(disengage())
exiting/
Cruiser→gen(engage())

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING



AARHUS UNIVERSITY
DEPARTMENT OF ENG

CarHandler Statechart



A: allocPlatform

E: platformAllocated/

A: moveTo(platform)

E: moveCompleted/

A: arrivAck

E: departReq/

A: allocExit

E: exitAllocated/

A: moveTo(platform)

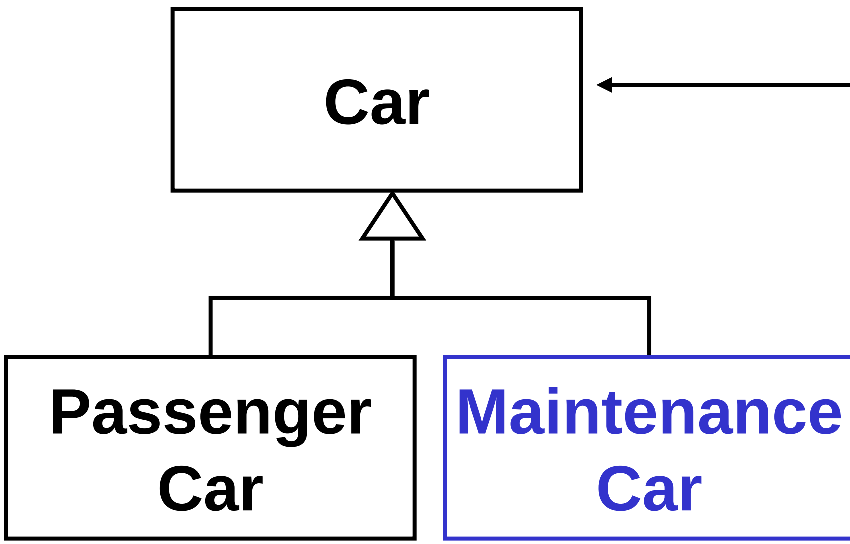
E: moveCompleted/

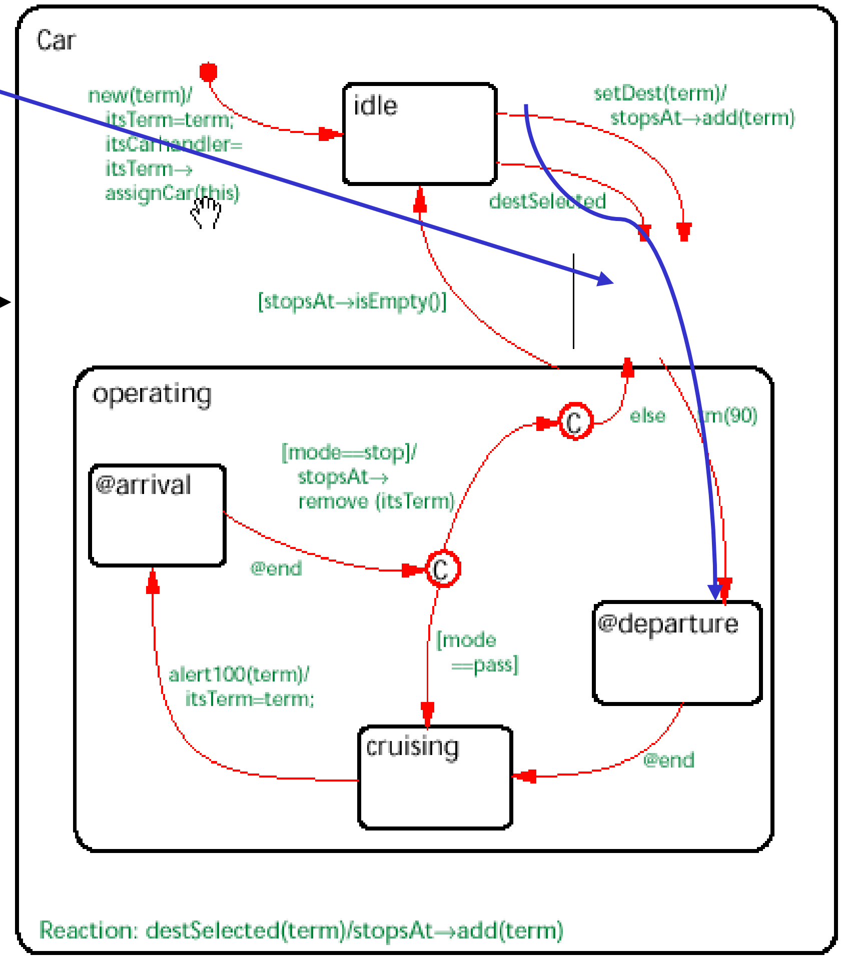
A: departAck

E: tm(10s)/

State Machine Inheritance

standby removed



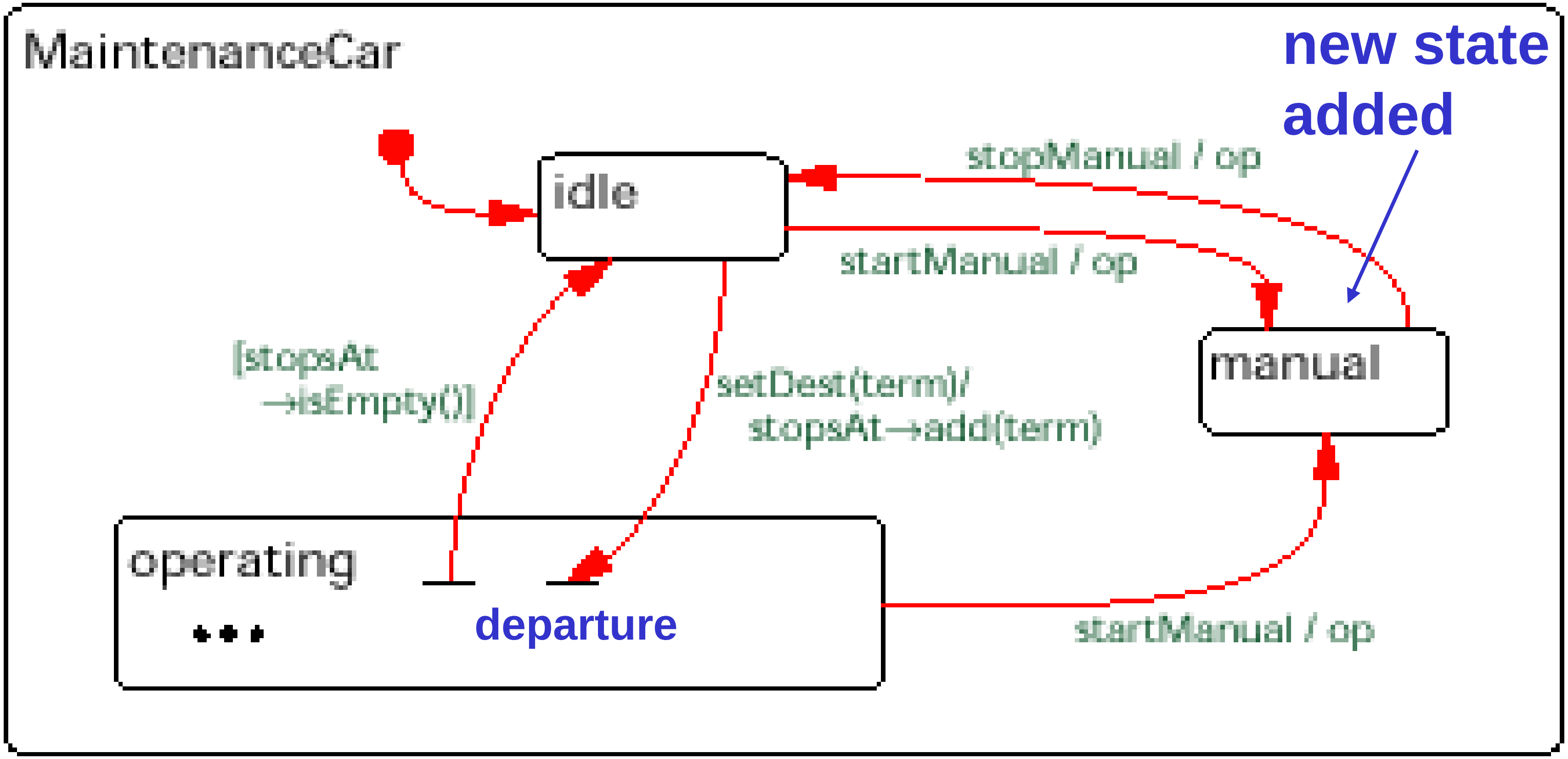


as previous with a standby state

new class

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

State Machine for MaintenanceCar



Summary

- State Machines and UMLs State Diagrams are very important tools for specifying and implementing embedded real-time systems
- UML State Diagrams have important concepts for specifying hierarchical and concurrent State Machines