Lecture 4
State Machines

Dr Peter T. Popov
Centre for Software Reliability

18th October 2018

Design - state machines

State machines before UML

- Finite-state machines, FSM, (often called a finite-state automaton) have been widely used long before UML.
- They model the behaviour of an entity (computer program, digital circuits with memory, etc) that can be in one of a ***finite number*** of states;
- The transitions between the states are governed by ***external stimuli*** and are dependent on the ***current state***.

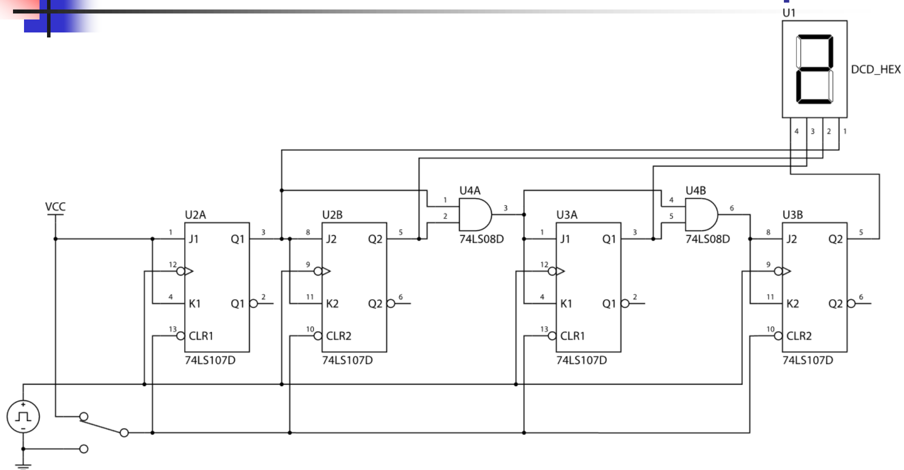
3

Examples of FSM

- Digital circuits with memory i.e. Registers implemented by flip-flops
  - Minimisation of the combinational logic used to drive the state changes is a very well established discipline
- Software programs
  - Replication (fail-over, database replication, etc.) are based on FSMs (proof of correctness of replication protocols explicitly refers to FSMs)
  - Consensus protocols in distributed systems
    - These guarantee for instance that all replicas are "on the same page", e.g. process change requests in the same order (known as "one copy serialisability")
- Markov/semi-Markov processes
- Petri Nets
- and many more.

4

Finite-state machine example



A 4-bit counter is an example of a simple state machine

5

Moore and Mealy state machines

- Moore machine uses ***entry actions*** only
  - An Output is produced when the state changes.
    - The output (action) depends on the state only, not on the input
- Mealy machine uses ***input actions*** only
  - Output is produced based on the ***input and the state***
- These two are functionally equivalent and mechanistic transformation exists from one to another

6

Mathematical model of FSM

- A *deterministic finite state machine* is a quintuple (Σ,S,s_0,δ,F), where:
  - Σ is the input alphabet (a finite, non-empty set of symbols).
  - S is a finite, non-empty set of states.
  - s_0 is an initial state, an element of S.
  - δ is the state-transition function.
    - stochastic formalisms may define δ stochastically
  - F is the set of final states, a (possibly empty) subset of S.
- O is often added to the above to represent the ***output alphabet*** – defined differently depending on the type of state machine

7

21.2

State machines in UML

- Some model elements (***classifiers*** to use the UML jargon) such as classes, components and subsystems, can have interesting ***dynamic*** behaviour
  - state machines can be used to model such behaviour
- Every state machine exists in the context of the particular model element that:
  - Responds to events dispatched from outside of the element
  - Has a clear life history modelled as a progression of *states, events and transitions.*
  - Its current behaviour depends on its past
- A state machine diagram always contains ***exactly one state machine*** for ***one model element***
  - If we want to model the behaviour of two different classes, then we develop ***two state machines***.
    - Q: How about sub-classes? **Can we model the superclass and its subclasses together, with the same state machine?**
    - A: No! Each sub-class will have its *own state machine*. However there will be *similarities* between the two, possibly large parts of the state-machines will be identical. UML offers advanced mechanism to model inheritance "SM extended".

8

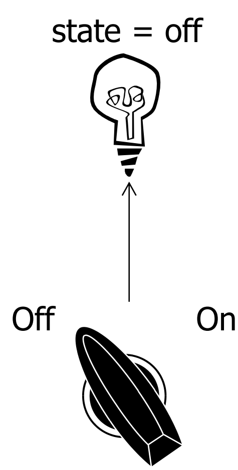
David Harel's work

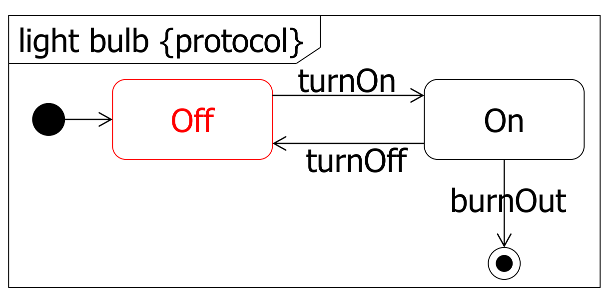
- UML's state machines are based on the work by David Harel
  - Harel generalised the previous models of FSM
  - In UML 1.X state machines were called 'state-charts', a term coined by Harel.
    - He considered state-charts as a *visual* formalism
  - Starting with UML 2.0 the term 'state-charts' was replaced with the term ***state machines***.
    - You can still find (especially in on-line resources) references to state-charts.

9

21.4

State machine diagrams



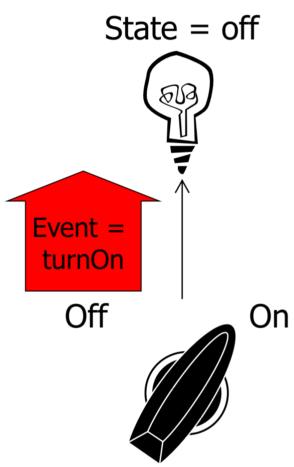


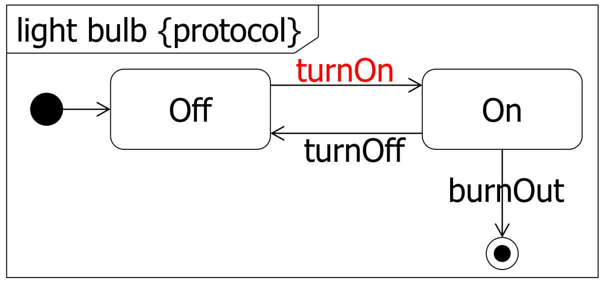
- We begin with the light bulb in the state "Off".

© Clear View Training 2010 v2.6 10

21.4

Light bulb turnOn

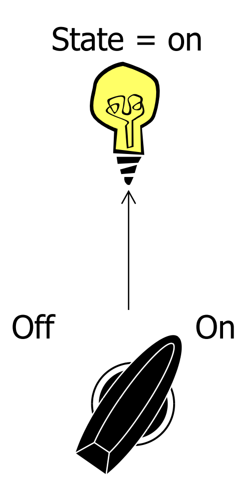


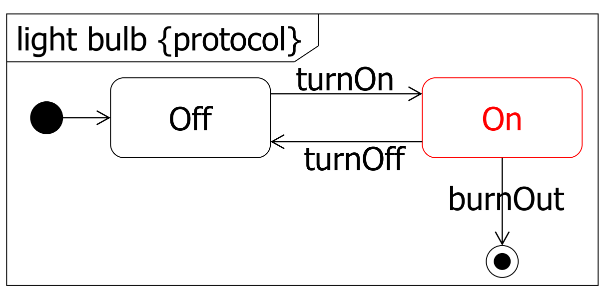


- We throw the switch to On and the event turnOn is sent to the lightbulb

21.4

Light bulb On

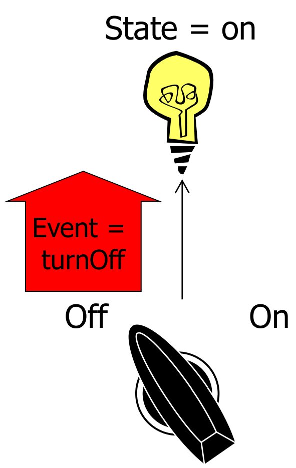


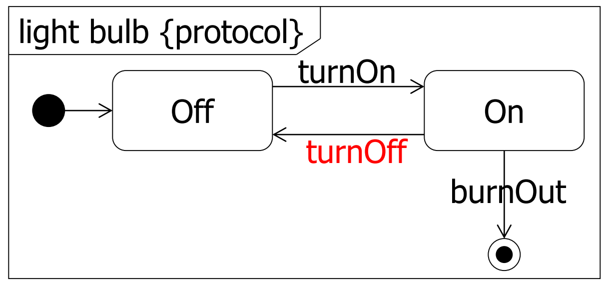


- The light bulb turns on

21.4

Light bulb turnOff

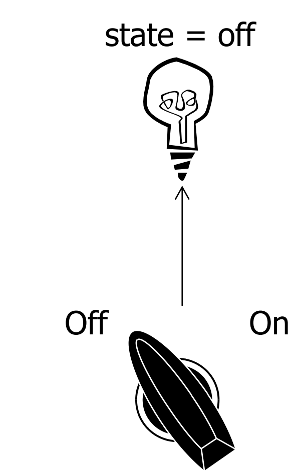


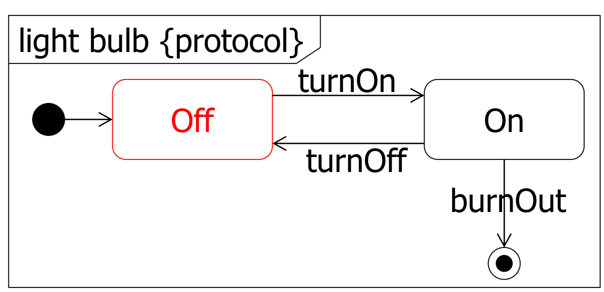


- We turn the switch to Off. The event turnOff is sent to the light bulb

21.4

Light bulb Off

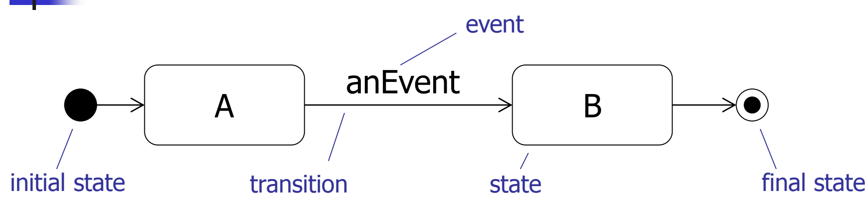




- The light bulb turns off

21.4

Basic state machine syntax



- Every state machine should have an *initial state* which indicates the first state of the sequence
  - E.g. The state in which an object is when it is instantiated
- Unless the states cycle endlessly, state machines should have a *final state* which terminates the sequence of transitions
- We'll look at each element of the state machine in detail in the next few slides!

 15

21.2.1

Kinds of state machine

- There are two different ***kinds of state machines***:
  - ***Behavioural*** state machines - define the behaviour of a model element e.g. the behaviour of class instances
  - ***Protocol*** state machines - Model the protocol of a classifier
    - The conditions under which operations of the classifier can be called
    - The ordering and results of operation calls
    - Can model the protocol of classifiers that have no behaviour (e.g. ***interfaces and ports***)
      - A protocol state machine linked to an interface may capture the expected order of messages that the users of the interface must respect. This order of messages may be an essential part of the contract defined by an interface.
    - The UML syntax of behavioural and protocol state machines is very similar:
      - Protocol state machines ***don't have actions***
      - Transitions are labelled differently

16

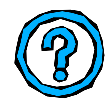
21.5

States

- "A condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event"
- The state of an object at any point in time is determined by:
  - The ***values*** of its attributes
  - The ***relationships*** it has to other objects
    - E.g. if class A has an association with 0..* multiplicity with class B, the state of an object of type A is affected by how many ***actual links*** this object has with objects of type B.
  - The ***activities*** it is performing
    - E.g. whether a particular method of a class is under processing or has been completed.

How many states?

Color
red : int green : int blue : int



© Clear View Training 2010 v2.6 17

21.5.1

State syntax

- Actions are ***instantaneous*** and ***uninterruptible***
  - Entry actions occur immediately on entry to the state
  - Exit actions occur immediately on leaving the state
- Internal transitions occur *within* the state. They do *not* transition to a new state
- Activities take a finite amount of time and are ***interruptible***

state name	EnteringPassword
entry and exit actions	entry/display password dialog exit/validate password
internal transitions	keypress/ echo "*" help/display help
internal activity	do/get password

Action syntax: eventTrigger / action
Activity syntax: do / activity

© Clear View Training 2010 v2.6 18

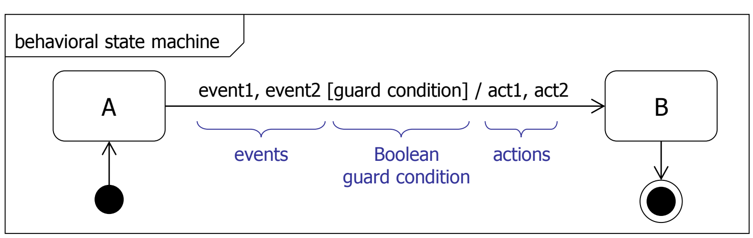
An example: States of Locker object

- Read the description of the Locker case study on slide 45.
- Locker object
  - Identify the states of a locker object
  - Look at entry and exit actions
  - Any useful activities (i.e. what the Locker does in its different states)

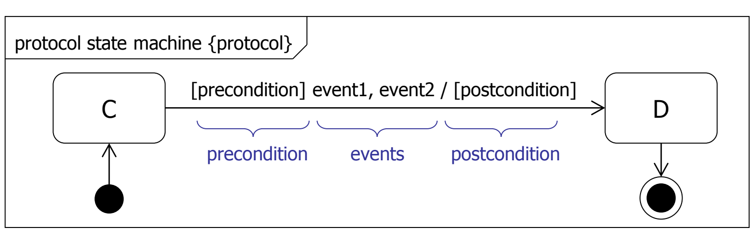
21.6

Transitions

behavioral state machine



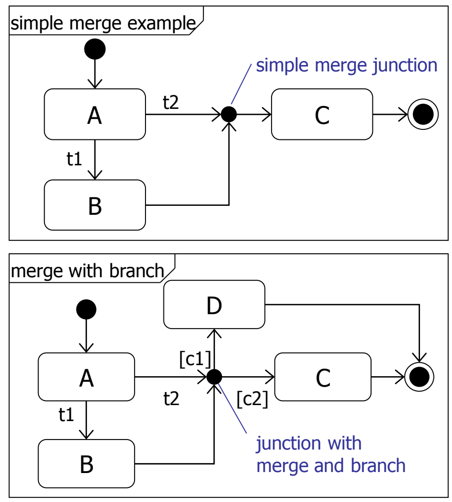
protocol state machine



21.6.1

Connecting - the junction pseudo state

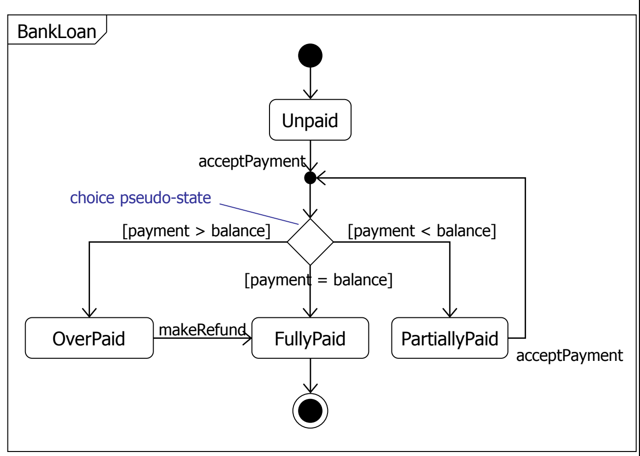
- The junction pseudo state can:
  - connect transitions together (merge)
  - branch transitions
- Each outgoing transition must have a ***mutually exclusive*** guard conditions
  - C1 and C2 must not overlap



21.6.2

Branching – the choice pseudo state

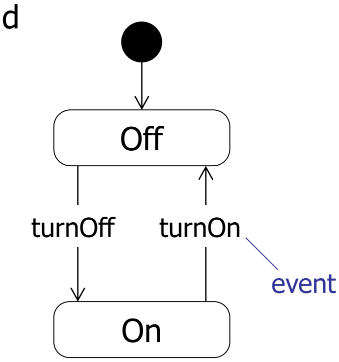
- The ***choice*** pseudo state directs its single incoming transition to one of its outgoing transitions
- Each outgoing transition must have a ***mutually exclusive*** guard condition



21.7

Events

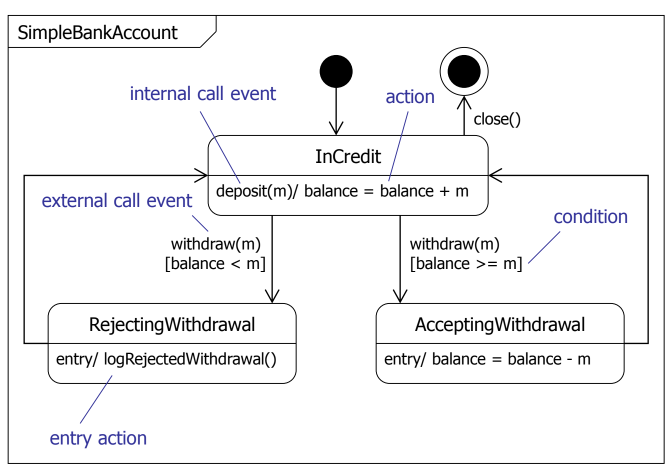
- "The specification of a noteworthy occurrence that has location in time and space"
- Events trigger transitions in state machines
- Events can be shown externally, on transitions, or internally within states (internal transitions)
- There are four event types:
  - Call event
  - Signal event
  - Change event
  - Time event



21.7.1

Call event

- A call for an operation execution
- The event should have the ***same signature*** as an operation of the context class
- A ***sequence*** of actions may be specified for a call event - they may use attributes and operations of the context class
- The return value must match the return type of the operation



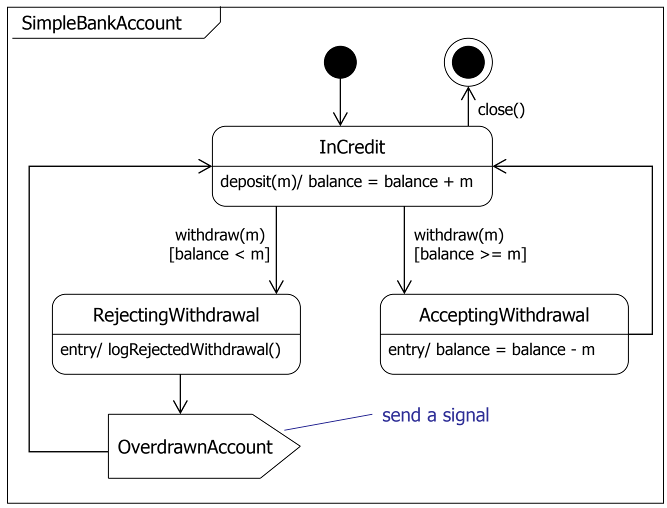
21.7.2

Signal events

- A signal is a package of information that is sent ***asynchronously*** between objects
  - the attributes carry the information
  - no operations

«signal»
OverdrawnAccount

date : Date
accountNumber : long
amountOverdrawn : double

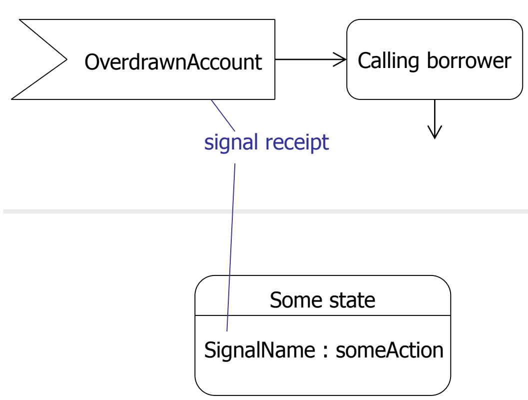


© Clear View Training 2010 v2.6 25

21.7.2

Receiving a signal

- You may show a signal receipt:
  - on *a transition* using a concave pentagon, or
  - as an *internal transition* state using standard notation

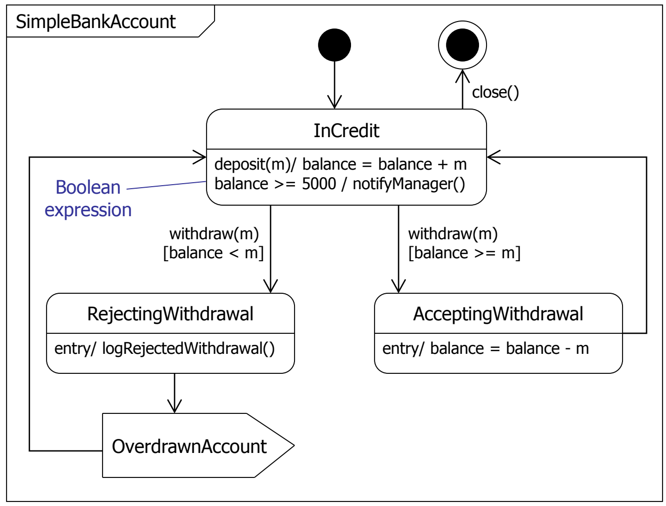


© Clear View Training 2010 v2.6 26

21.7.3

Change events

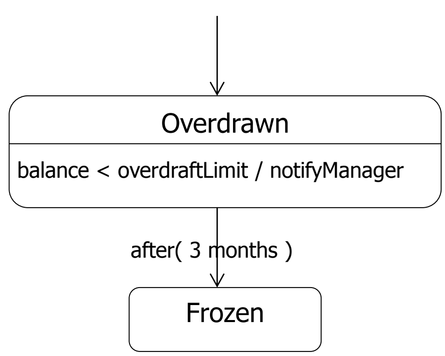
- The action is performed when the Boolean expression transitions *from false to true*
  - The event is *edge triggered* on a false to true transition
  - The values in the Boolean expression must be constants, globals or attributes of the context class
- A change event implies *continually testing* the condition whilst in the state



21.7.4

Time events

- Time events occur when a time expression becomes true
- There are two keywords, after and when
- Elapsed time:
  - after(3 months)
- Absolute time:
  - when(date =20/3/2000)



Context: CreditAccount class

Example: Events with the Locker

- Identify:
  - Events in the Locker case study that lead to transitions
  - Look for Time events

29

State machines vs Activity Diagrams

- These two diagrams look similar, but are really quite different. Similarity of syntax is misleading!
  - State-machines (SM) have:
    - states: may be mapped to values of attributes, existence of links with objects of other classes. A state defines which events (e.g. methods invocations) will be accepted (and responded to), any other events will be ignored. An SM remains in a state until a transition is successfully completed.
    - Transitions – are triggered by an explicit trigger, which is defined, and may have an action. ***Automatic transitions*** are possible (without explicit trigger), but they are ***NOT the norm***.
    - A guard, when false, may block a transition.
  - Activity diagrams have nodes: actions, objects, control nodes.
    - Actions are pieces of computation. When processing is complete, a transition takes place to the next node. There is no need for an explicit trigger. ***Automatic transitions are the norm***.
    - Conditions apply to control nodes (decision/merge and fork/join).

30

SM vs Activity Diagrams (2)

- The focus of diagrams is different:
  - State-machines typically model the behaviour of an instance of a ***single classifier*** (e.g. of an object) and defines how the state of the instance may change in response to ***ANY method*** defined for the classifier.
  - Activity diagrams are typically used to model:
    - Business processes (e.g. use - cases) in which different actors can participate ("swim lanes"). Instances of different classifiers may be involved in the process (objects, pins, etc.)
    - The logic of a ***single method*** of a classifier (e.g. a class)
  - We may use an Activity diagram to model the "do" activity defined for a state-machine.

In summary: Make sure that you study and understand state-machines!

31

21.8

Summary

- We have looked at:
  - Behavioural state machines
  - Protocol state machines
  - States
    - Actions
      - Exit and entry actions
    - Activities
  - Transitions
    - Guard conditions
    - Actions
  - Events
    - Call, signal, change and time

The lecture follows closely Chapter 21 of Arlow's book

 32

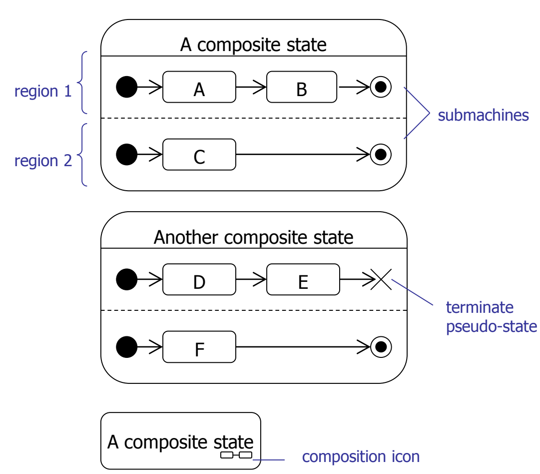
Design - advanced state machines

 33

22.2

Composite states

- Have one or more regions that each contain a nested submachine
  - Simple composite state
    - exactly one region
  - Orthogonal composite state
    - two or more regions
- The final state terminates its enclosing region – all other regions continue to execute
- The terminate pseudo-state terminates the whole state machine
- Use the composition icon when the submachines are hidden

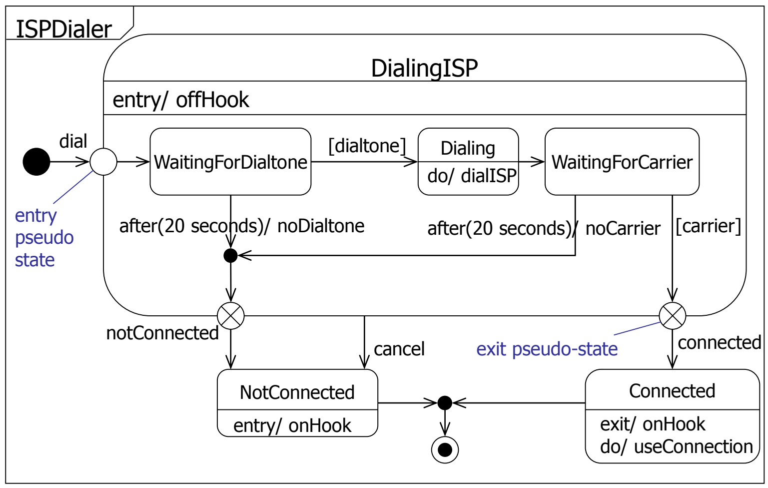


 34

22.2.1

Simple composite states

- Contains a *single region*
- The nested states inherit the cancel transition from DialingISP
  - i.e. can cancel from any sub-state.



© Clear View Training 2010 v2.6 35

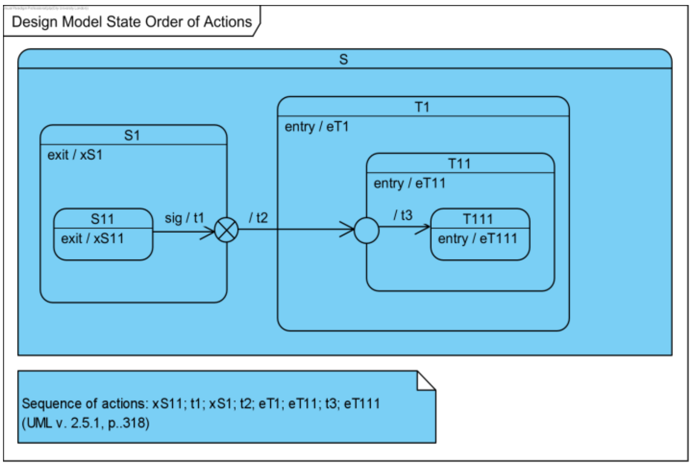
Example: Locker state machine

- Complete the behavioural state machine of the Locker:
  - Identify the states including the composite states
    - Identify entry, exit actions
    - Identify activities
  - Specify the transitions: events, guards and actions

36

Action Execution Order in SM

- Every Transition, except for **internal** Transitions, causes exiting of a source State, and entering of the target State. These two States, which may be composite, are designated as the *main source* and the *main target* of a Transition respectively.



Consider the following example (© OMG, UML v2.5.1, p. 318)

The SM is in state S11 and receives a trigger "sig".

What is the order of executing the *actions* (entry/exit and those attached to transitions)?

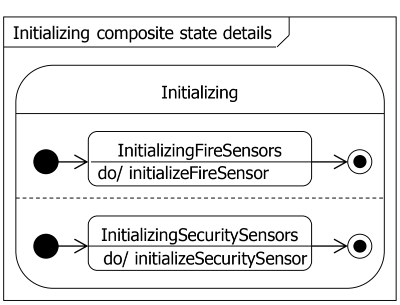
37

22.2.2

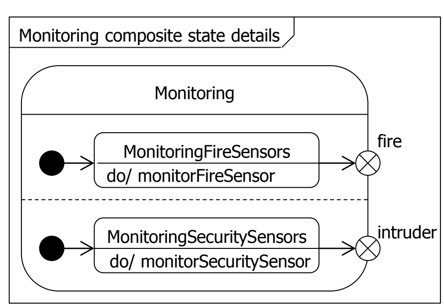
Orthogonal composite states

- Has two or more regions
- When we enter the superstate, both submachines start executing concurrently - this is an implicit fork

Synchronized exit - exit the superstate when *both* regions have terminated



Unsynchronized exit - exit the superstate when *either* region terminates. The other region continues

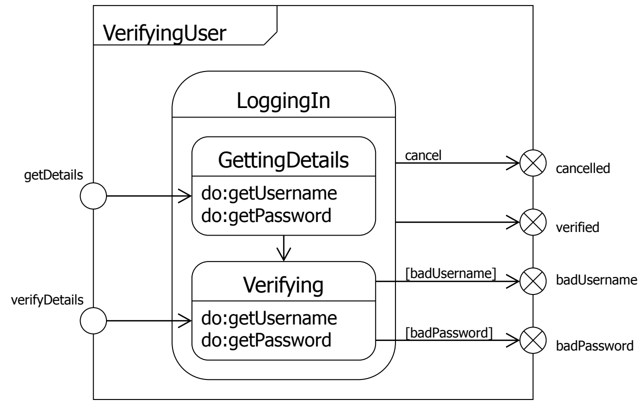


 38

22.3

Submachine states

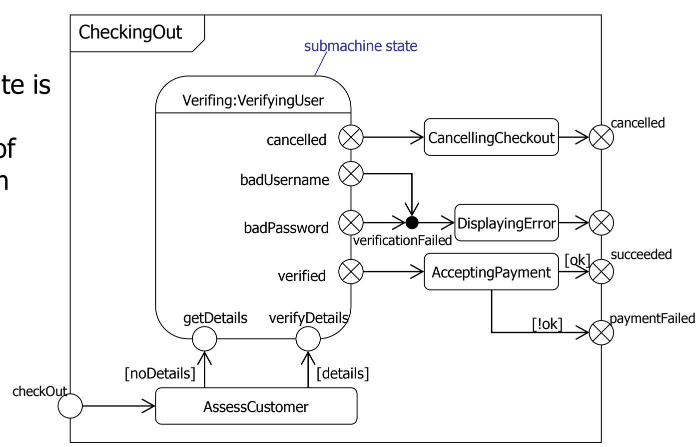
- If we want to refer to this state machine in other state machines, without cluttering the diagrams, then we must use a *submachine state*
- Submachine states reference another state machine
- Submachine states are semantically equivalent to composite states



22.3

Submachine state syntax

- A submachine state is equivalent to including a copy of the submachine in place of the submachine state

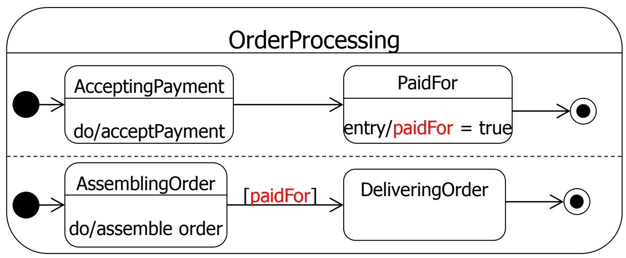


22.4

Submachine communication

- We often need two sub-machines to communicate
- Synchronous communication can be achieved by a join
- Asynchronous communication is achieved by one submachine setting a flag for another one to process in its own time.
  - Use attributes of the context object as flags

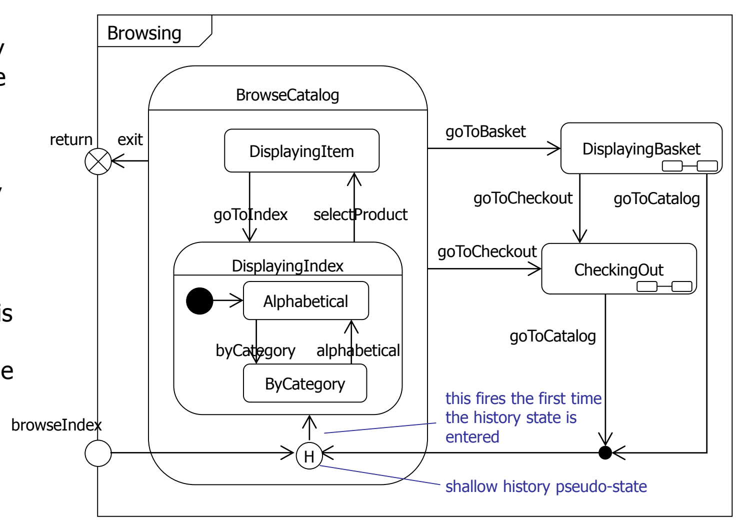
Submachine communication using the attribute PaidFor as a flag: The upper submachine sets the flag and the lower submachine uses it in a guard condition



22.5.1

Shallow history

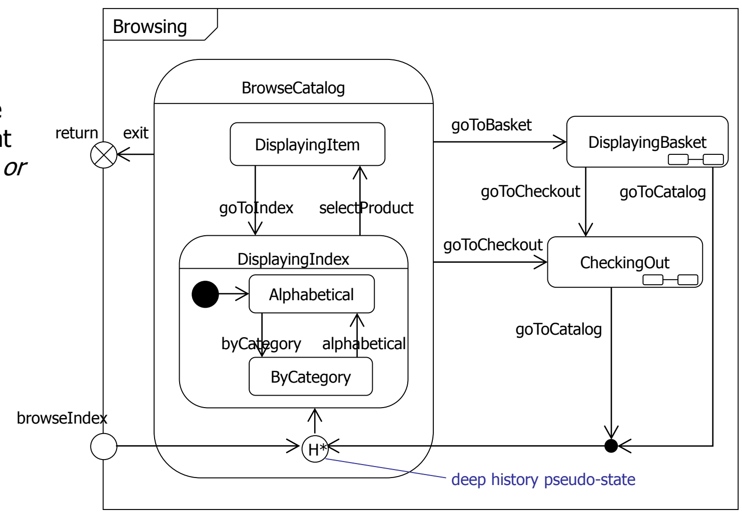
- Shallow history remembers the last sub-state at the *same level* as the shallow history pseudo state
- Next time the super state is entered there is an automatic transition to the remembered sub-state



22.5.2

Deep history

- Deep history remembers the last sub-state at the same level *or lower* than the deep history pseudo state



22.6

Summary

- We have explored advanced aspects of state machines including:
  - Simple composite states
  - Orthogonal composite states
  - Submachine communication
    - Attribute values
  - Submachine states
  - Shallow history
  - Deep history

- Advanced state machines are covered in Chapter 22 of Arlow's book.

Locker case study

The BAGS system will control racks of lockers, in which members of the public will store their luggage for a period of time. The Lockers offer the following functions:

- Users can select and use an empty locker. Empty lockers are indicated by a green light.
- After placing items in a locker and closing the locker door, the user selects the required length of time of hire and pay the necessary amount of currency (cash) or swipe a valid credit card. If a successful payment is received, the Locker's door is locked and the green light goes off. The user will be given a receipt with a unique security code.
- In order to retrieve deposited items, the user enters the security code and pays any additional charge due, in case the originally selected time has been exceeded. Then the Locker's door is unlocked and the green light will go on again.
- When the selected time of hire is exceeded the user is allowed to make an additional payment due to exceeding the period. The payment must be received within 5 min after typing in the security code. After typing the security code the green light starts flashing. If the payment is not received within 5 min, the locker's door remains locked and the green light goes off. In this case the user is prompted to type the security code again and try to complete the payment within another 5 min. Successful and sufficient payment received within 5 min unlocks the Locker door and the green light goes on.

At any time a locker may be either operational (i.e. offer the functionality listed above) or be out of service due to hardware failure. Fixing the failures will make the locker operational again.

45