SYSC-3120—Software Requirements Engineering

Part IV – State-Based Modeling

State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

Different Kinds of Object Behaviours [Douglas, Wagner et al]

- **Simple behaviour**: object performs services on request and keeps no memory of previous services
  - e.g., a simple math function such as sine or square root
  - function returns the last value measured from a sensor
  - function returns the value of object attribute.
- **State behaviour** (a.k.a., state-driven, reactive): the way the object performs services depends on what happened in the past (memory), i.e., what other services have occurred before
  - e.g., a cruise control
    an elevator control
- **Continuous behaviour**: current output depends on the previous history in a way that does not lend itself to discretization (as in state behaviour)
  - e.g., digital filter

Purpose

- What do we model with a state machine?
  - behavior of complex entity classes (e.g., customers, accounts)
  - behavior of control classes
    - State(s) of execution of a use case (e.g., conditional messages)
  - behavior of a subsystem or system, but often too complex to represent with a single state-machine
    - Need to abstract away from details, or
    - Use many communicating state-machines (one for each object)
- We will assume we model the state-based behaviour of a class
  - Rather the behaviour of any instance of the class
- What are the benefits?
  - systematic ways of implementing classes based on their state machine description; lends itself to automatic code generation
  - state-based behaviour can be automatically verified.

State Machine

- Shows a behaviour that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to events.
  - describes the pattern of events, states and state transitions undergone by objects of a class
  - describes the life **history** of instances of a class

- State machine
  - Is not concerned with algorithmic behaviour or internal control
  - Is concerned with “when” operations execute, rather than “what” operations do, or “how” they are implemented
  - Not needed for stateless objects which always respond in the same way to each stimulus, regardless of their state

State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

Notion of State in a Finite State Machine

- State = information about past history;
  condition that persists for a significant period of time
- All states represent all possible situations in which the state machine may ever be.
- Specifies a kind of memory: how the state machine can have reached the present situation.
- As the application runs the state changes from time to time, and outputs may depend on the current state as well as on the inputs.
- States are distinguishable: i.e., we can observe that they differ from one another in either one (or several) of:
  - the events they accept
  - the transition they take as a result of accepting those events
    - a transition is a response to an event that causes a state change
  - the actions they perform.

State

- **State**: a condition or situation in the life of an object during which
  - the object satisfies some condition,
  - the object performs some activity (in response to events),
  - the object waits for some event.
- Condition satisfied in a state = state invariant
  - each state is defined by one unique condition (state invariant)
  - different states must satisfy different conditions
  - state invariant described in terms of (condition on what?):
    - attribute values
    - links to other objects
    - other objects' states
- The current state of an object is the state in which the state invariant is currently satisfied
  - the current value of the object's attributes and the links that it has with other objects (and possibly their state)

State (example 1)

- Class `StaffMember` has an attribute `startDate` which determines whether a `StaffMember` object is in the *probationary state*:
  - The `StaffMember` object is in the `Probationary` state for the first six months of employment.
  - While in this state, a staff member has different employment rights and is not eligible for redundancy pay in the event that they are dismissed by the company.

- Some attributes and links of an object are significant for the determination of its state while others are not.
  - `staffName` and `staffNo` attributes of a `StaffMember` object have no impact upon its state (according to the specification above)
  - `startDate`, and more so the comparison between `startDate` and the current day is the condition that defines the probationary state.

State (example 2)

- A 1L bottle is either *empty*, *full*, or *partially full*
- These are three states that specify different behaviours of the bottle, i.e. different ways for the bottle to respond to events
  - In state empty or partially full, the bottle can accept more liquid
  - In state full, the bottle cannot accept more liquid
  - In state partially full or full, we can remove liquid from the bottle
  - In state empty, we cannot remove liquid from the bottle
- Assuming the bottle has an attribute `quantity`
  - `quantity=0` is the state invariant of state empty
  - `quantity=1` is the state invariant of state full
  - `quantity>0 and quantity<1` is the state invariant of state partially full
- A bottle cannot be in two states at the same time, i.e., two state invariants cannot be true at the same time: state conditions are distinguishable.

State vs. Class Invariant

- State condition = State invariant
  - A condition that does not vary (invariant) while the object is in the respective state
  - Attribute values may vary, but the condition remains.
- Class invariant = true in all the state**s** an object can be in.

- Bottle:
  - The state is either empty, full or partially full
  - In state partially full, the quantity in the bottle is not 0 and is not the maximum allowed
    - this condition does not vary
    - but the quantity (if one is pouring water in the bottle) does.

State and Class Invariant Specification

- Often one adds a **`state`** attribute that is an enumeration, to facilitate the specification of state invariants and the class invariant.
- State invariant for state empty (three alternatives):

```
(state=BottleState::empty) = (qty=0)
```

or

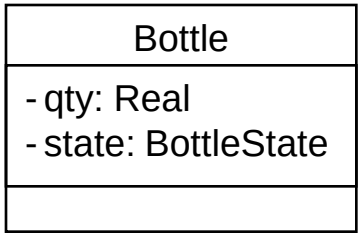
```
(state=BottleState::empty) implies (qty=0)
```

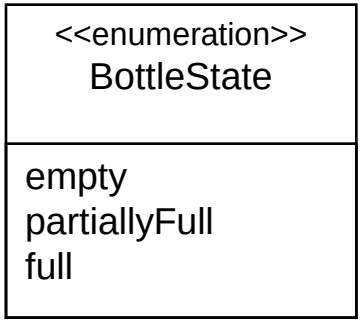
or

```
(state=BottleState::empty) and (qty=0)
```

- Class invariant

```
(state=BottleState::empty and qty=0)
xor
(state=BottleState::full and qty=1)
xor
(state=BottleState::partiallyFull and qty>0 and qty<1)
```





Class vs. State Invariant

```
Class ImmediateJob
Inv:  pickuploc<>””
  and pickupTime=””
  and bookedCab=performedBy
  and
  (
    ( jobState=JobState::dispatched
      and self.performedBy->notEmpty
      and self.performedBy.currentJob=self
      and self.performedBy.cabState=CabState::busy
    )
    xor
    ( jobState=JobState::completed
      and self.performedBy.currentJob->isEmpty
      and self.performedBy->isEmpty
    )
  )
```

state invariant (dispatched)

state invariant (completed)

class invariant

UML Symbols for States

- Graphically

(named) state

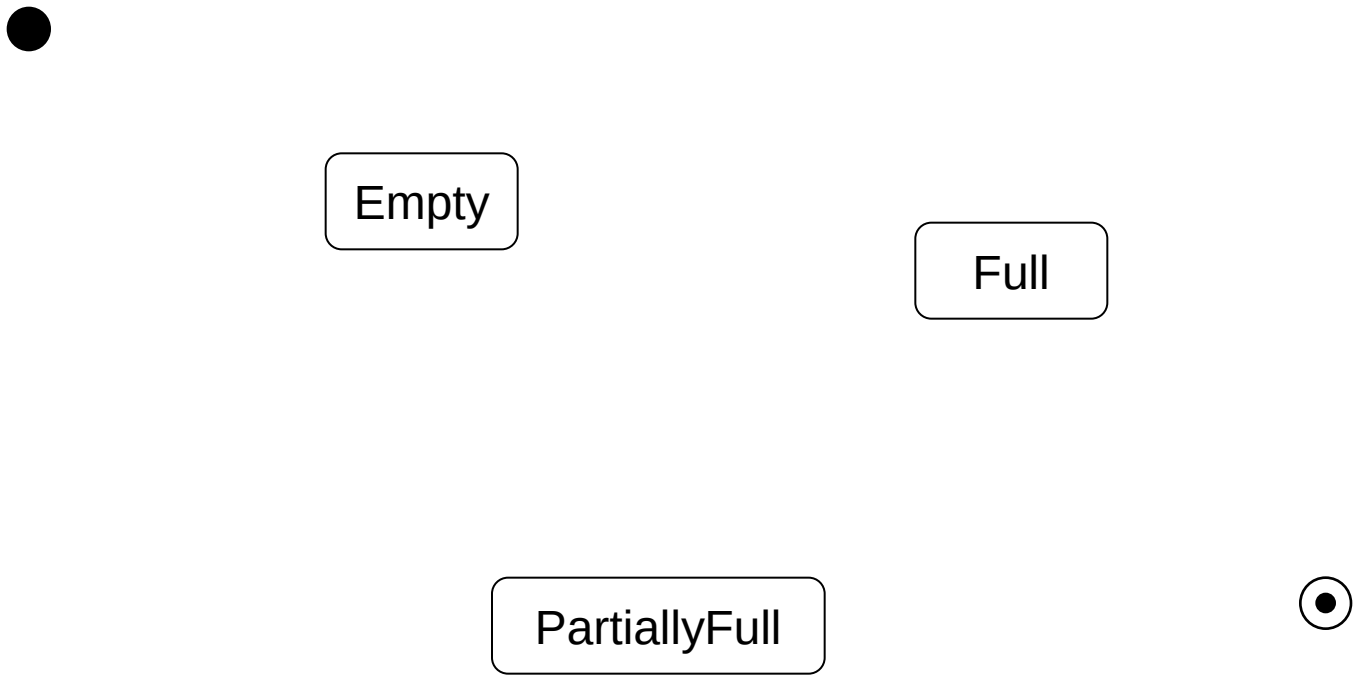
- Special states
  - Initial state: indicates the default starting state
    - State before any behaviour specified by the state machine can actually happen.
    - Indicates a state before the element modeled by the state machine is actually created.

initial state

  - Final state: indicates that the execution of the state machine has been completed
    - Specifies that the element which behaviour is being modeled by the state machine has reached the end of its life

final state

Bottle States

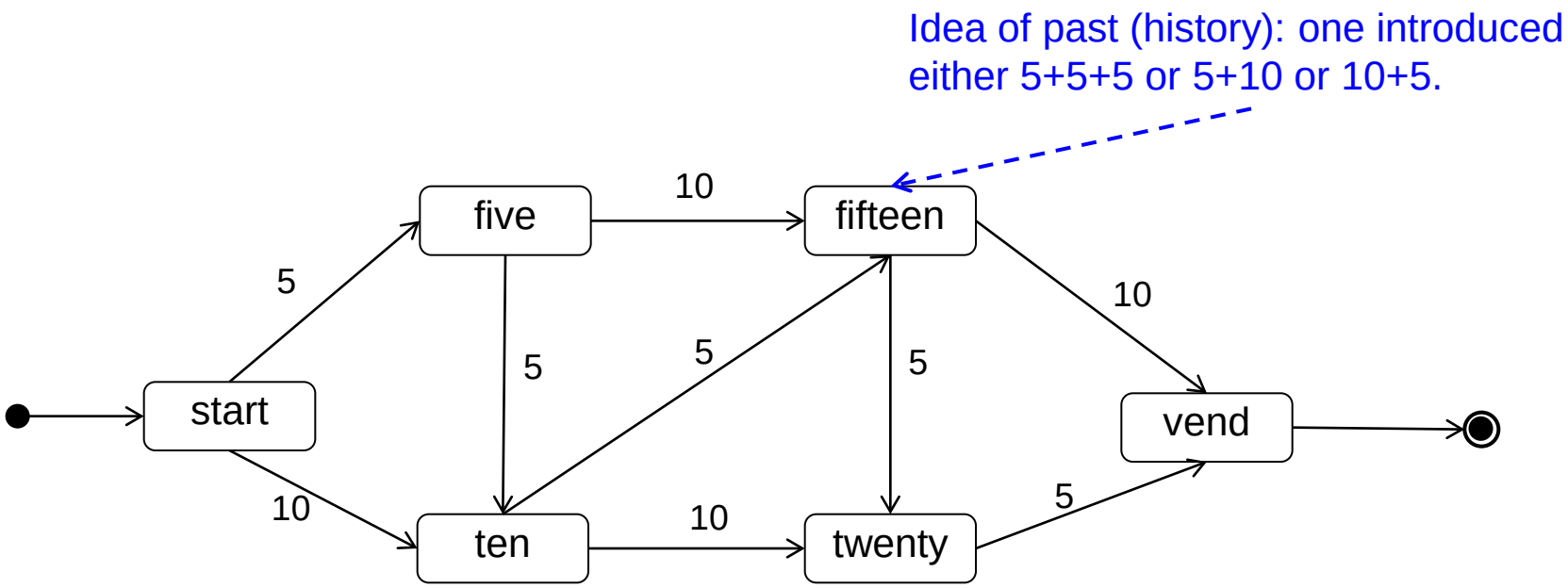


Finite State Machine

- The object (or component) being modeled can only assume a finite number of existence conditions called states
- The object behaviour in a given state is (distinguishable from other states' behaviour) essentially identical and defined by:
  - The messages and events accepted
  - The actions associated with each incoming event
  - The state's reachability graph (i.e., how state can change)
  - The set of transitions
- An object spends all its time in states
  - I.e., transitions take (approximately) zero time
- The object may change state only in a finite number of well-defined ways, called transitions
- Transitions are enabled by events: a response to an event that causes a change in state
- An object cannot be in two different states at the same time.
  - One (and only one) state condition holds at a given instant

A Simple Finite State Machine

A control system has to count the amount of money dropped into a vending machine. Only 5 and 10 cent coins are accepted. The correct, recognized sum (e.g., to deliver a drink) is 25 cents.



State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

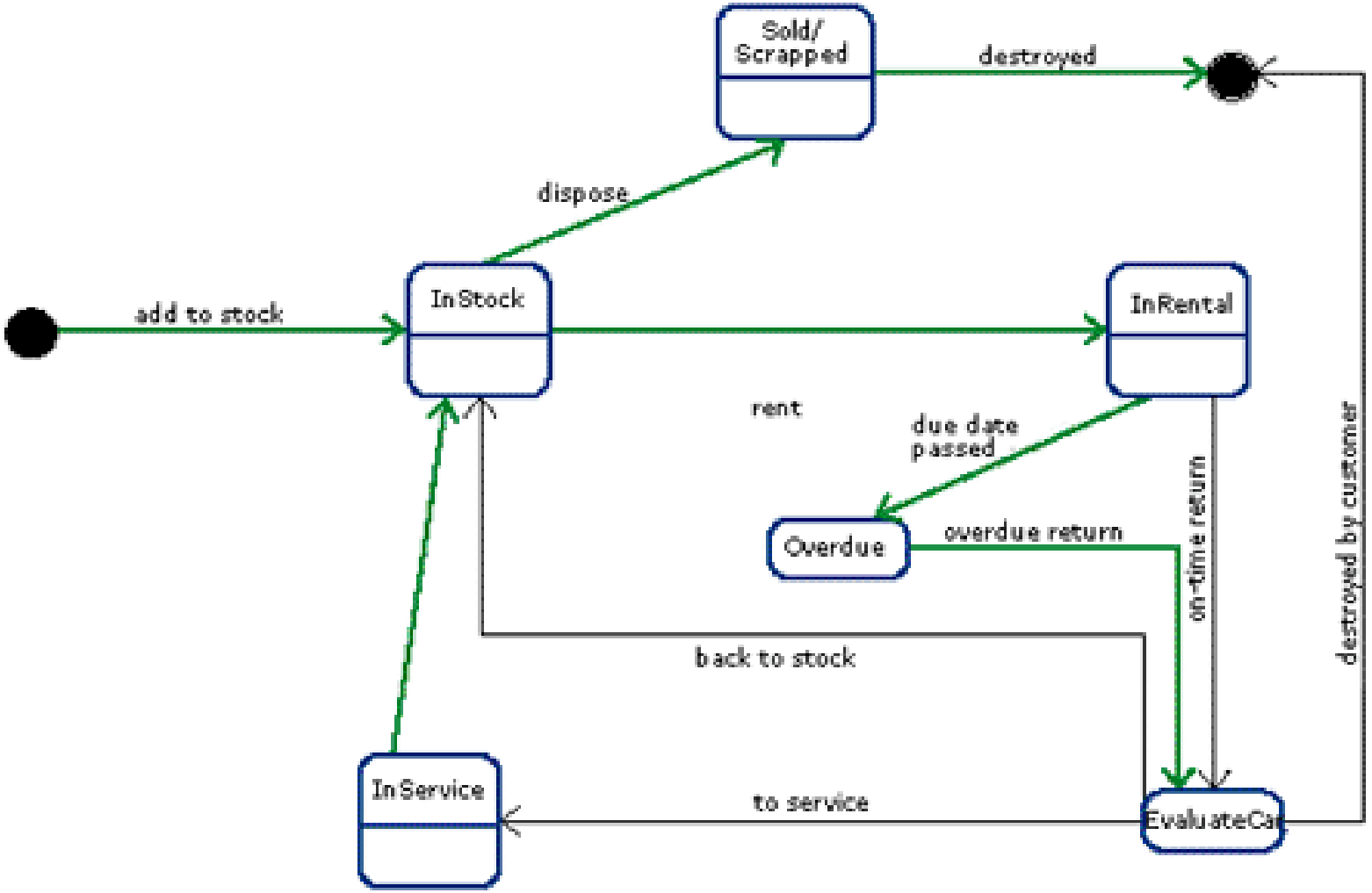
Event

- **Event**: the occurrence of a stimulus that can trigger a state transition.

- **Transition**: a relationship between two states indicating that an object in the first (initial) state will perform certain actions and enter the second (target) state when a specified event occurs and specified conditions are satisfied.

- State changes are caused by events

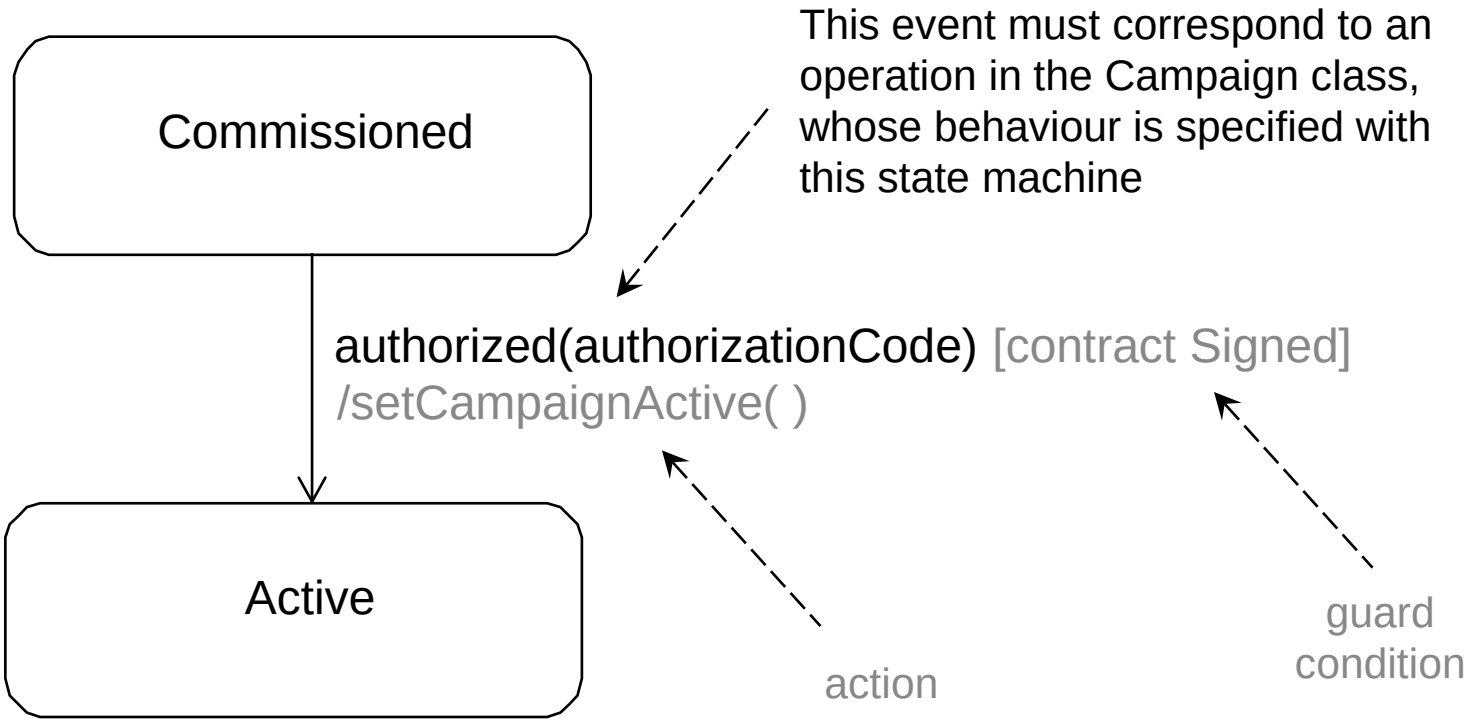
Example: state of a rental object



Transitions and Events

- Transition: the act of changing state
- A transition is initiated by an event (also called trigger).
- Four kinds of events in UML:
  - **Signal event**:
    - An occurrence of interest arising asynchronously from outside the scope of the state machine, by means of a signals (a signal can carry data)
  - **Call event**:
    - An explicit synchronous notification of an object by another
  - **Change event**:
    - An event based on the changing of an attribute value
  - **Time event**:
    - Either the elapse of a specific duration or the arrival of an absolute time
- Warning: an event has negligible duration
  - Something that occurs at a particular time instant
    - Recall: transitions take (approximately) zero time

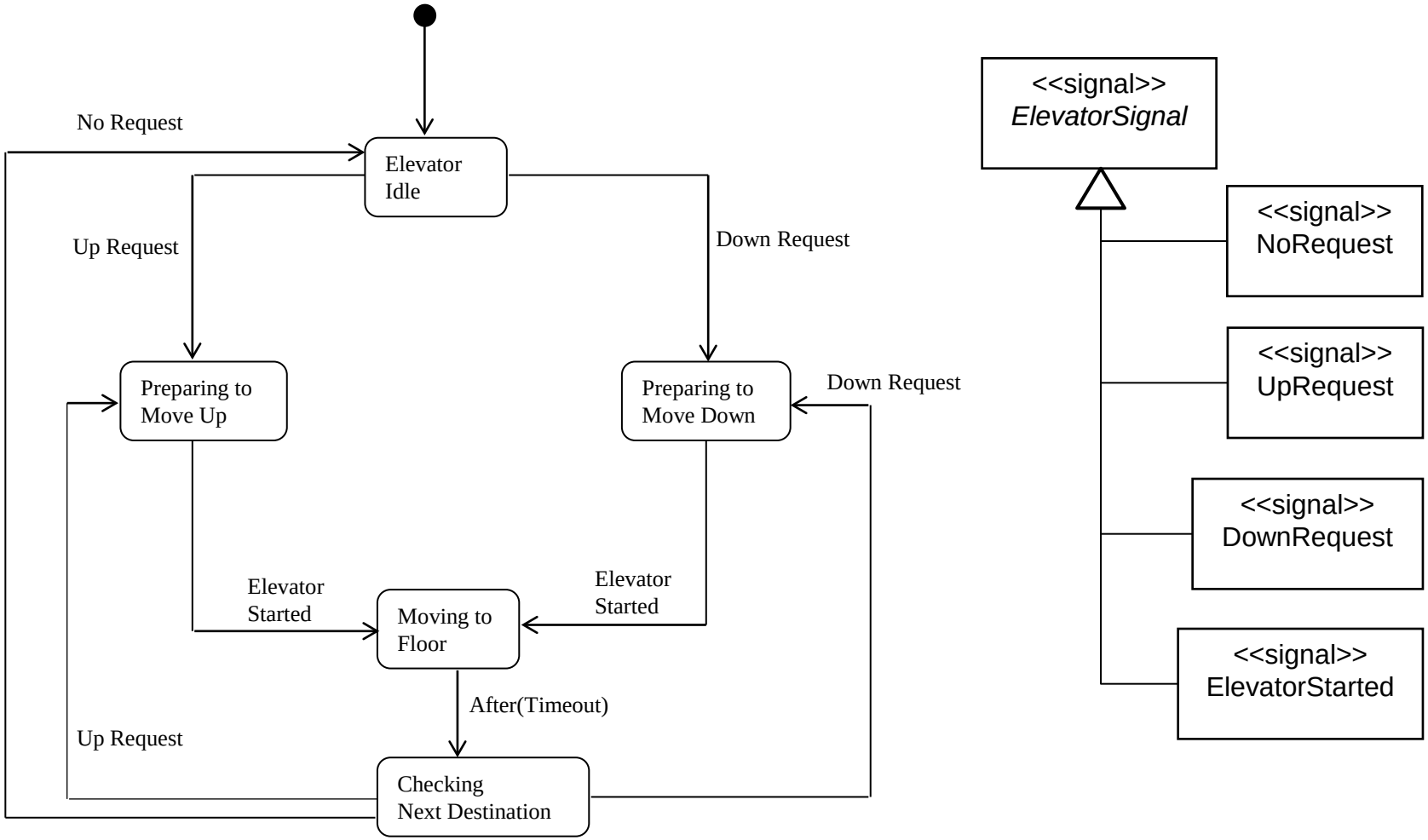
Call Events


Commissioned
This event must correspond to an operation in the Campaign class, whose behaviour is specified with this state machine
authorized(authorizationCode) [contract Signed]
/setCampaignActive( )
Active
action
guard condition

Signals

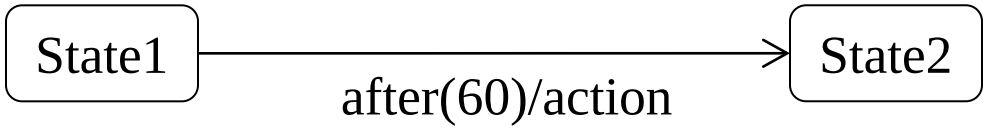
- UML definition: “named event that can be raised”
- Signals have properties => *can be modeled as objects with attributes*
  - typical signal attributes are priority, time sent, sender ID
- Passed synchronously or asynchronously between objects
  - Messages can pass a signal object, instance of a «signal» class.
- All the signals in a real-time system are modeled in a class hierarchy.
  - Signal classes must not be related to “normal” classes.

Signal Events



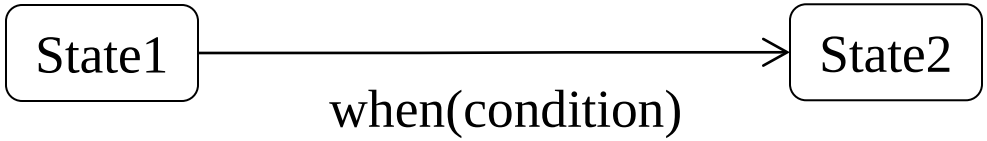
Elapsed-Time / Timeout transition

- A specific kind of event can be used to specify a timeout:
- The *after(x)* event
  - The UML doesn't define the units, but commonly they are milliseconds or microseconds.
- This indicates a timeout event which fires some specified period of time after the source state is entered.
- The timeout event is cancelled if the source state is exited prior to the timeout, e.g., with a signal event.



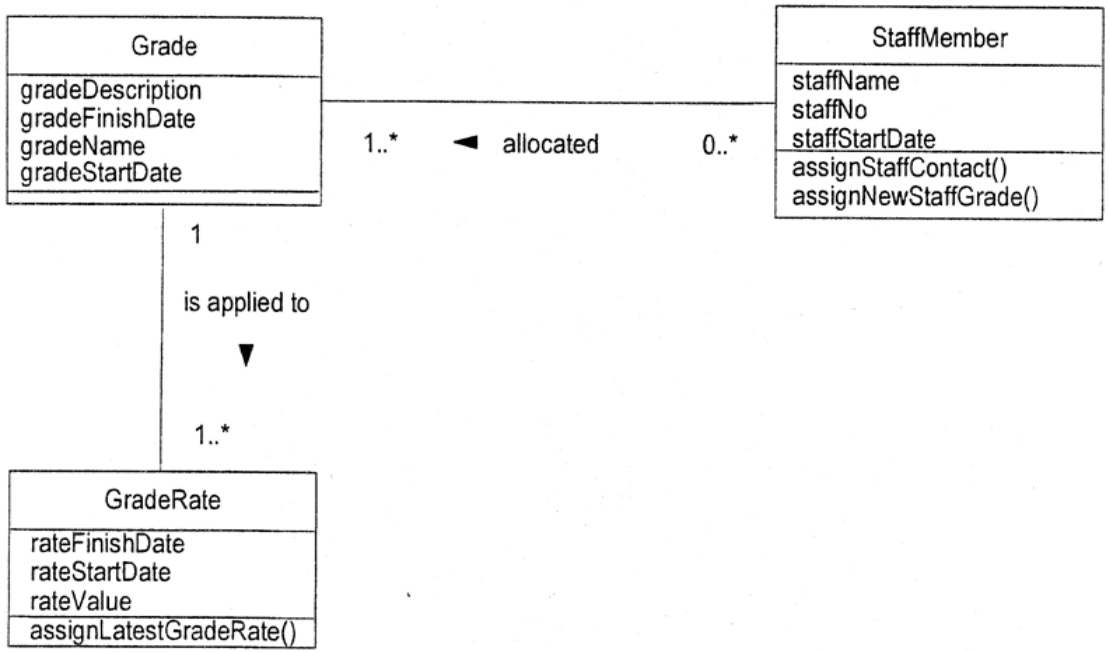
Change Events

- Represents the notification that a condition has become true
- Is specified using the “when” keyword
- Must have a Boolean expression (e.g., OCL) enclosed in parentheses designating the condition that must become true in order for the transition to fire
  - Unless the source state is exited prior to the this, e.g., with a signal event

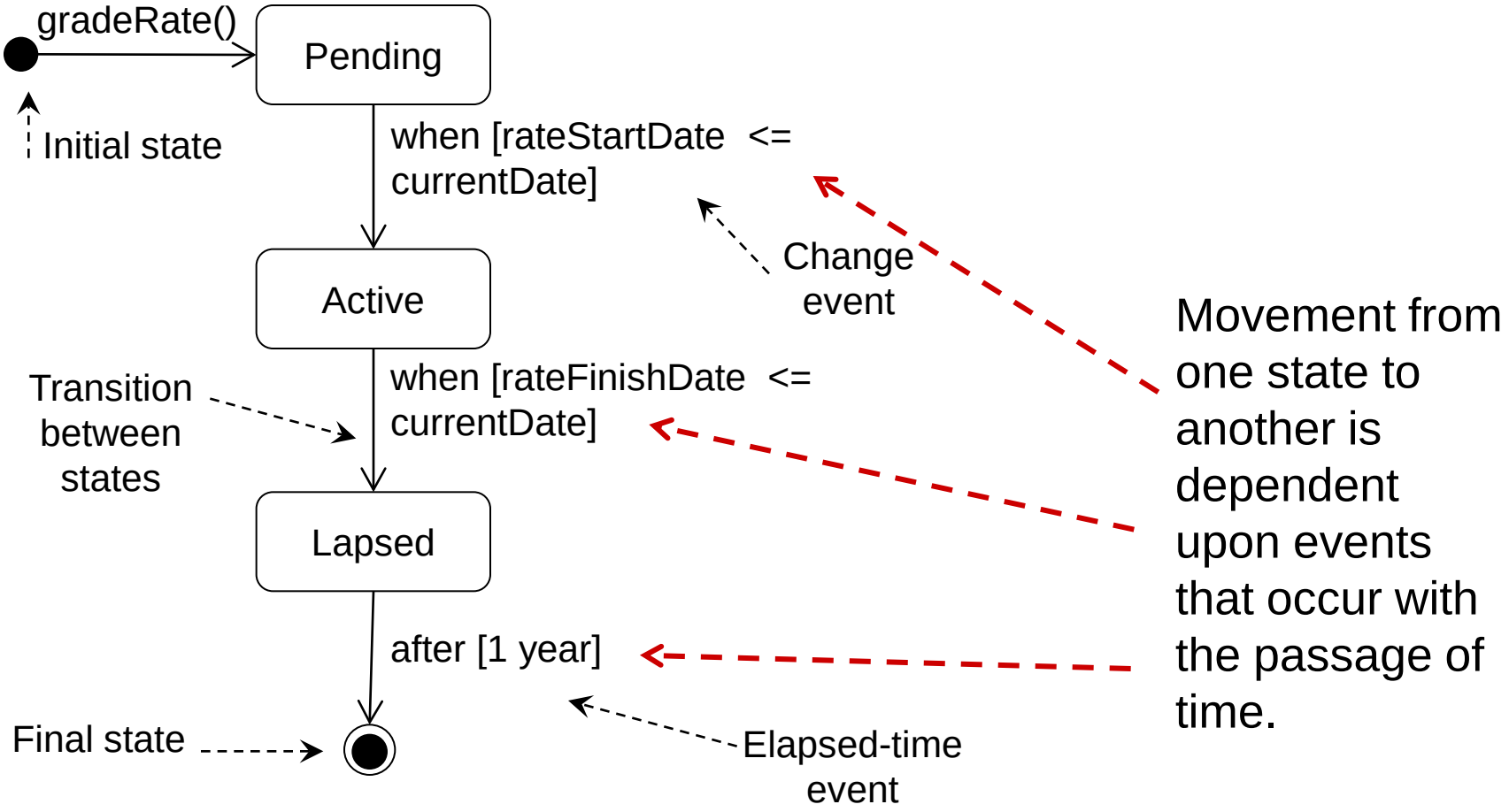


Example: elapsed-time and change events

- This example illustrates elapsed-time events and change events
- The current state of a **`GradeRate`** object can be determined by the two attributes **`rateStartDate`** and **`rateFinishDate`** (state variables)
- An enumerated state variable may be used to hold the object state, possible values would be **`Pending`**, **`Active`** or **`Lapsed`**



Class GradeRate states



Transitions

- A transition has
  - a **trigger**, i.e., the event that can fire the transition and make the state machine change state,
  - an **optional guard condition** that is a condition controlling the firing of the transition,
  - an **optional action** (or series of actions), a.k.a., transition action(s), that happens when the transition fires.
- The label of a transition has the following form:

  triggeringEvent [guardCondition] / actionSequence
- In a nutshell:

  if the state machine is in the source state,

  and the triggeringEvent is received,

  and the guard condition is true,

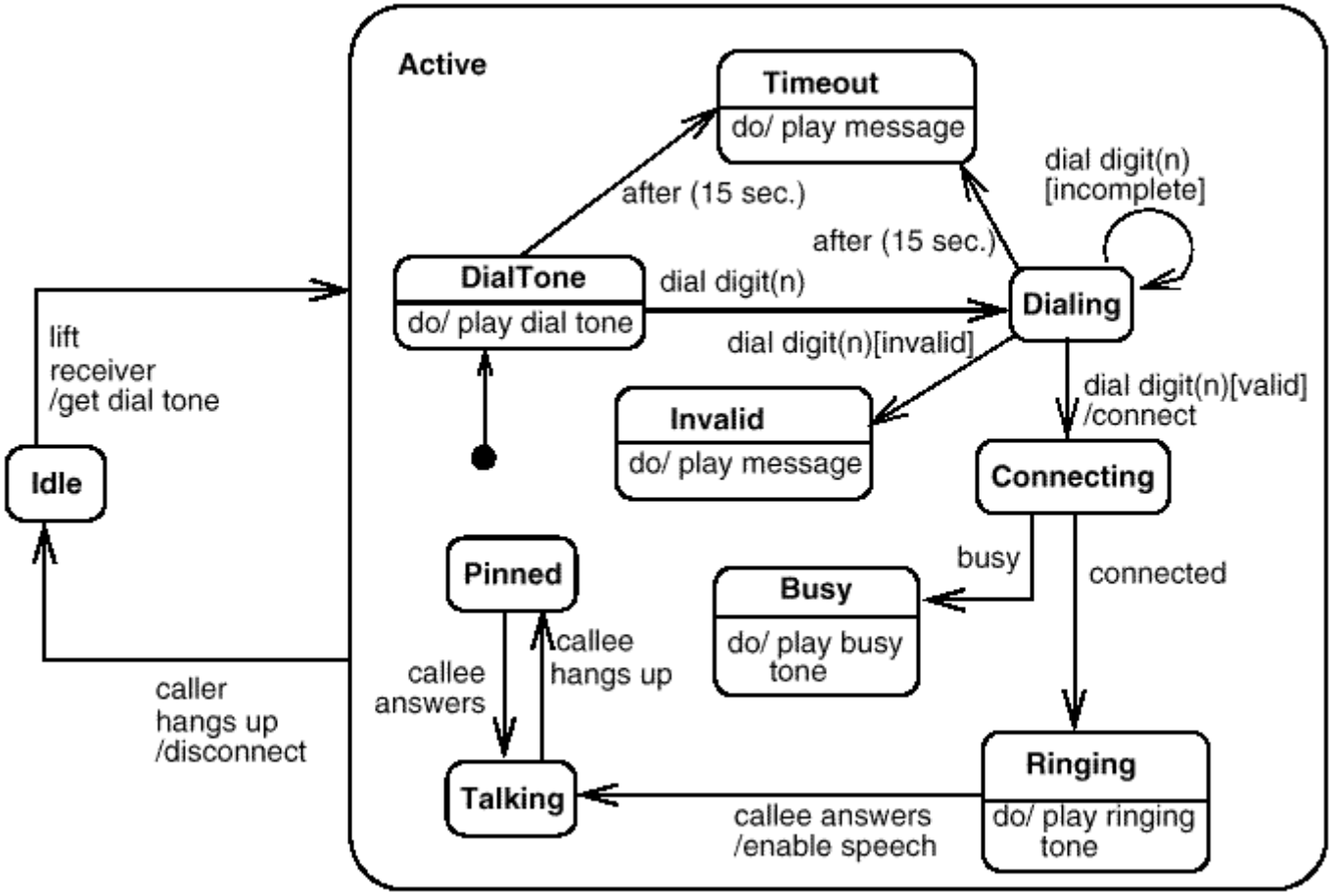
  then the action sequence is executed

  and the state machine changes state to the target state

Guard Condition

- A Boolean expression
- No side effect: i.e., it does not change the state of the system in any way
  - even if it calls some (query) class/object operations
- The model elements that can be evaluated in a guard condition:
  - the attributes of the class that is modeled,
  - links and their contents
  - states of linked objects
  - the arguments of the triggering event in case of a call event or a signal event.
- Guard condition vs. change events:
  - Guard conditions are evaluated when the event fires.
  - Change events fire when a condition becomes true; they are continuously checked when an object changes.

Example: Phone call statechart



State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

Execution Semantics

- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.
- When an event occurrence is detected and dispatched, the transitions out of the current state (also called the current state configuration, or active state configuration) are candidate for firing.
  - First, the transitions (there might be more than one) with a triggering event that matches the received event occurrence are identified.
  - If no transition outgoing from the active state configuration is identified, the received event occurrence is discarded (lost).
  - Second, the guard conditions of the matched transitions are evaluated.
  - If a guard condition evaluates to true, the corresponding transition is enabled and fires.
    - Since conditions are distinguishable, only one condition can be true at the same time and therefore only one transition fires
  - If no guard condition evaluates to true, the received event occurrence is discarded (lost).

Execution Semantics (cont.)

- Once a transition is enabled and it fires, the following occurs:

1. The source state is properly exited,
   - i.e., its activity is interrupted in case it is still running
   - and its exit action is executed.
2. The transition actions are executed in the order they are specified.
3. The target state is properly entered
   - i.e., its entry action is executed.

Notions to be discussed later

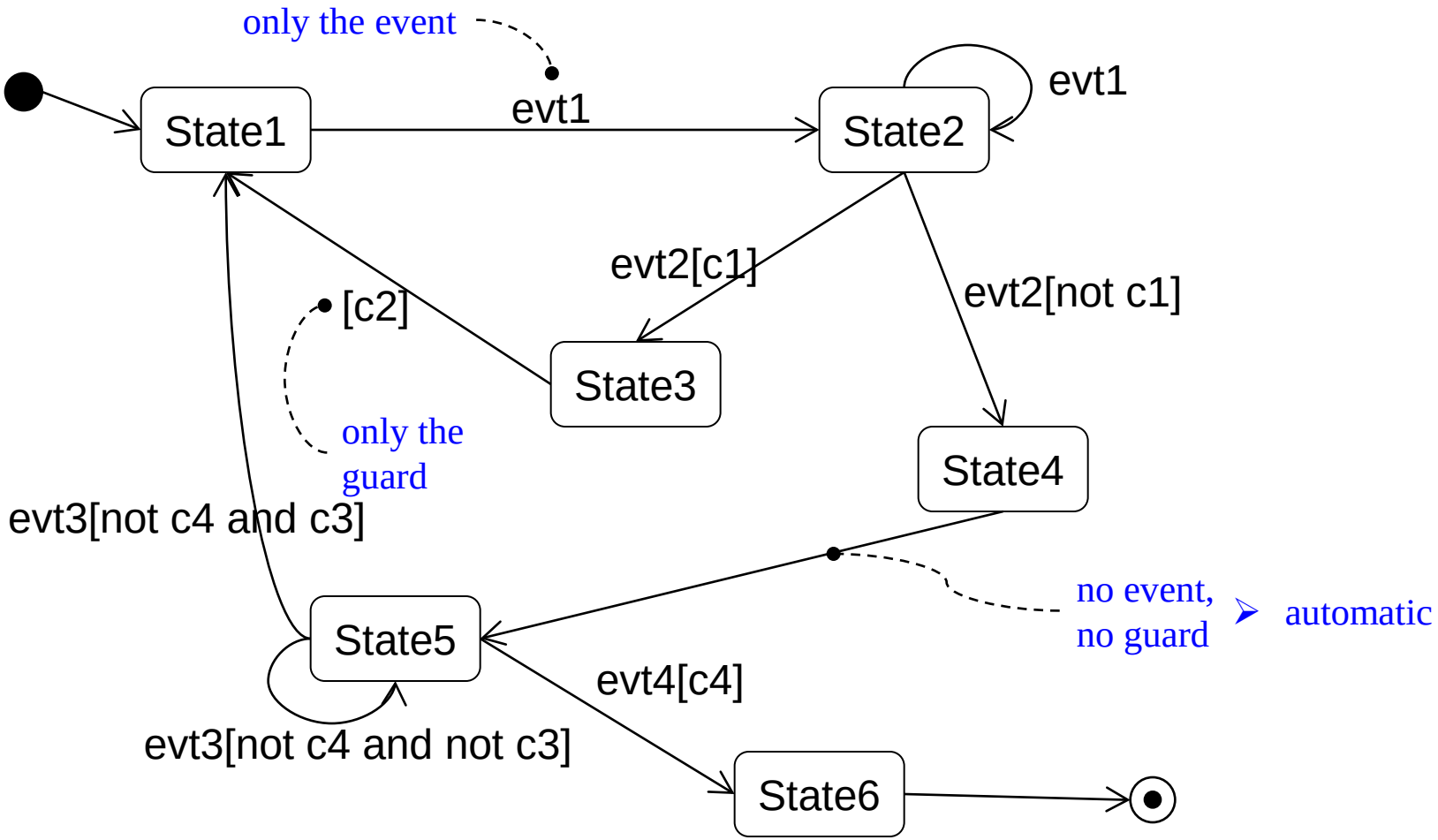
Execution Semantics (cont.)

- Dispatching events?
  - Signal event, time event or change event, dispatching the event is simply that: an event is dispatched.
  - What about call events?
- Dispatching a call event is simply that: an event is dispatched
  - This is simply a request to execute
  - The request may not be granted!

Execution Semantics (cont.)

- Assuming a transition with a trigger matching a call event can be found out of the active state
- The guard is evaluated:
  - Until now this is still a request to satisfy a call!
  - The receiving object is evaluating whether it can grant the request as specified in the state machine.
- If the transition is enabled, i.e., the guard condition evaluates to true, then the source state is existed, the operation corresponding to the call event is executed, the transition actions executes, and finally the target state is entered
- If the transition is not enabled, i.e., the guard condition evaluates to false, then the event occurrence is discarded (lost)

Abstract Example



State Transition Table

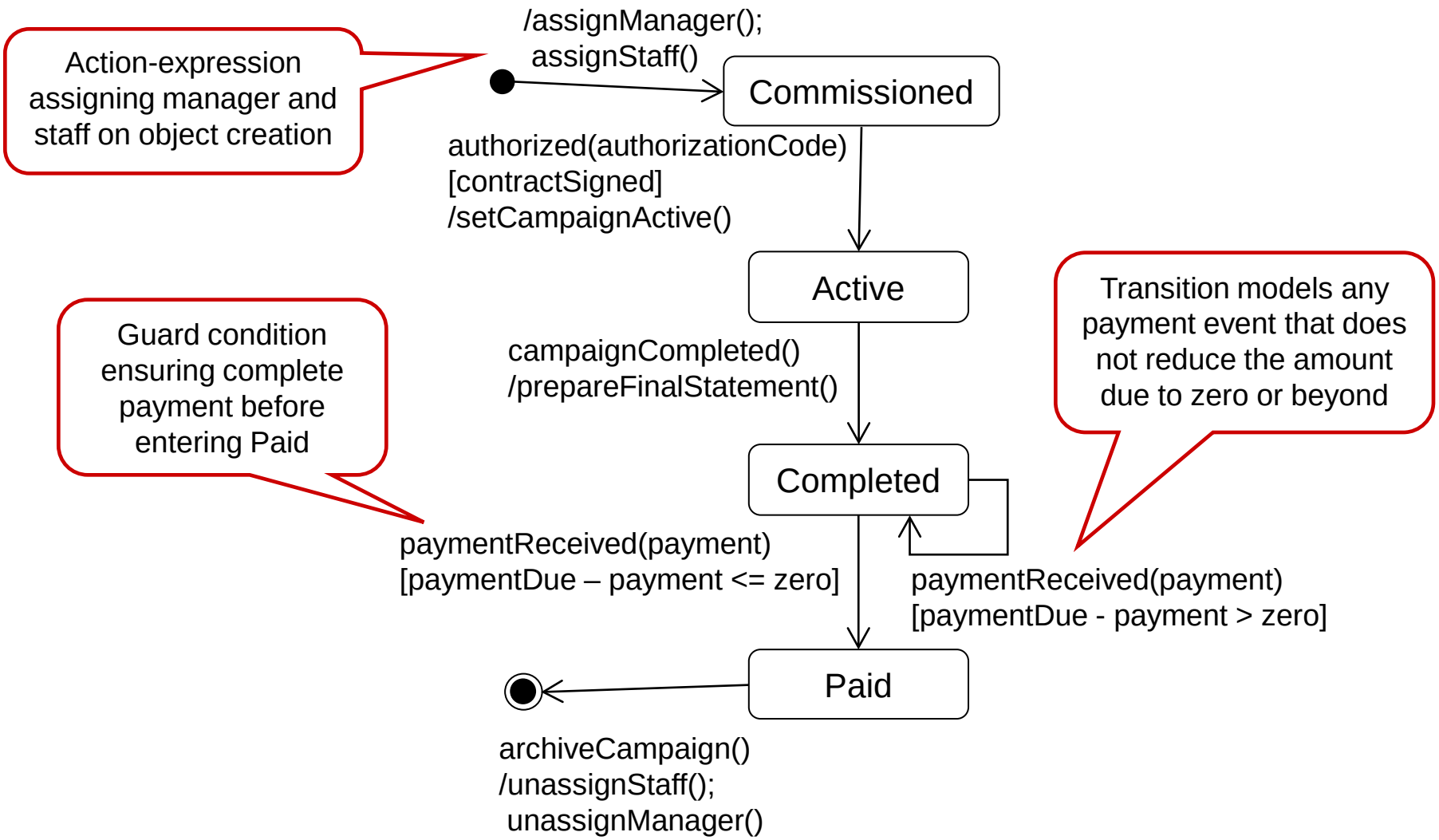
State	Condition (if any)	Event received	Result
State1		evt1	State2
		evt2	ignored
		evt3	
		evt4	
State2		evt1	State2
	C1 = true	evt2	State3
	C1 = false	evt2	State4
		evt3	ignored
		evt4	
State3		evt1	ignored
		evt2	
		evt3	
		evt4	
	C2 = true		State1

Actions in Transitions

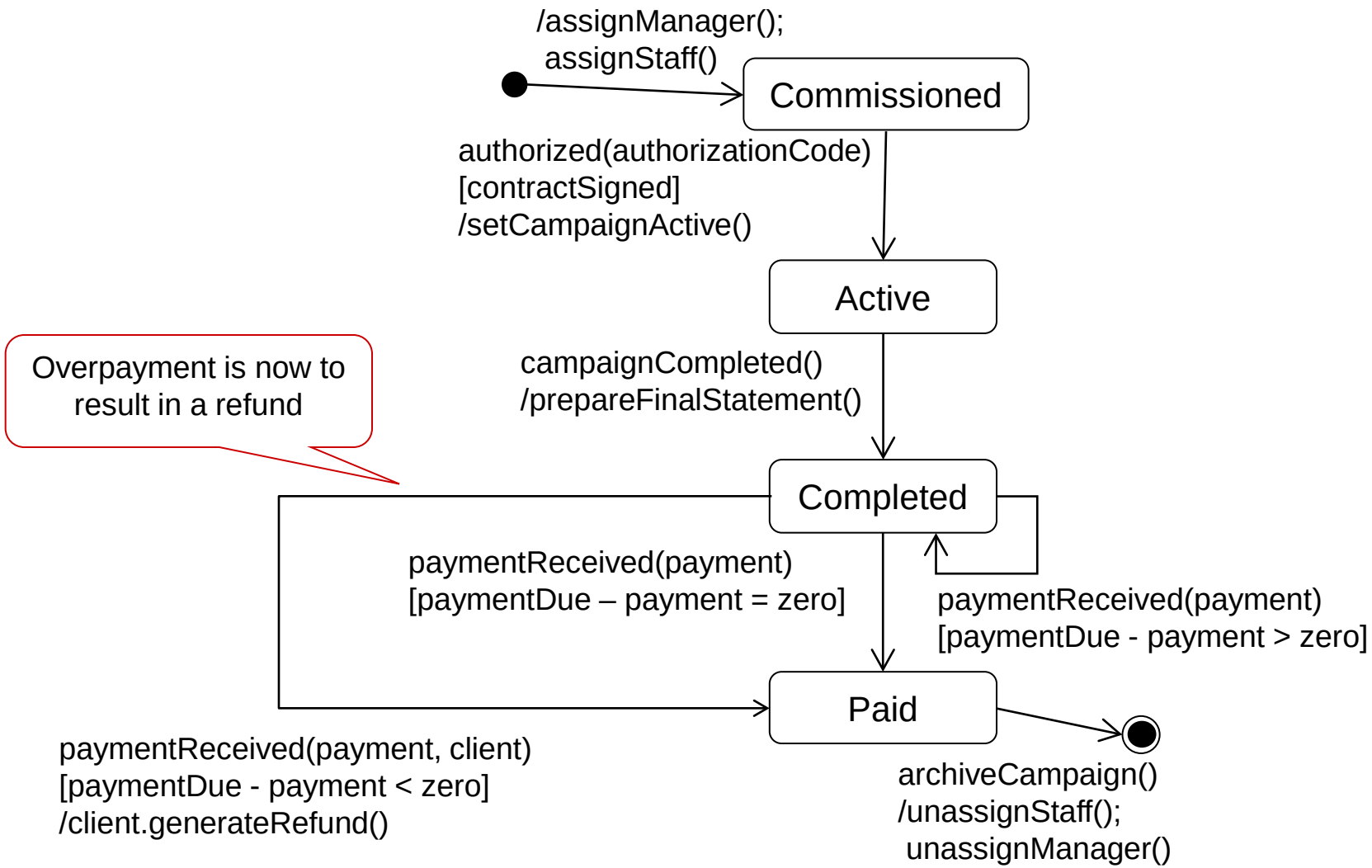
- Action in a transition:
  - An executable atomic computation that may directly act on the object that owns the statechart, and indirectly on other objects that are visible to this object
  - An action may include: operation calls, creation or destruction of another object, assignment, sending of a signal to an object
- Atomicity: cannot be interrupted by an event and therefore runs to completion
- Event syntax:

  event(arguments) [guard condition] / action(arguments)

Example: class Campaign



Example: revised statechart for class Campaign



State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

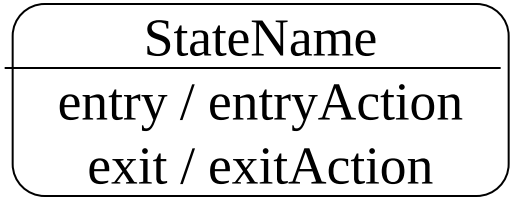
Actions in State

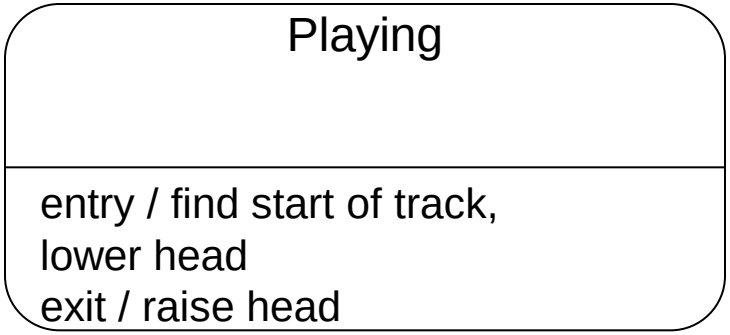
A state can have:

- An **entry action**, executed when entering the state
- An **exit action**, executed when exiting the state
  - If a signal is discarded, no exit action triggered (as if the signal never happened)

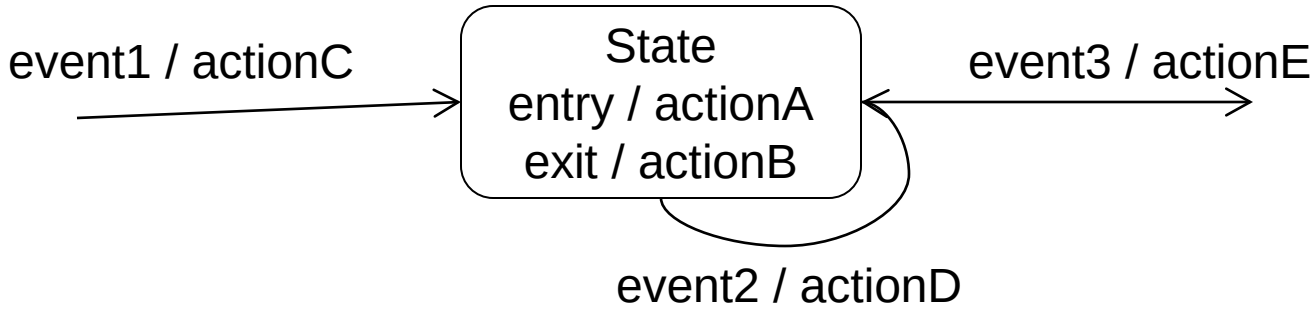
This helps:

- Dispatching the same action whenever we enter (resp. exit) a state, no matter which transition led us there (resp. away)





The Order of Action Executions



Sequence of events received

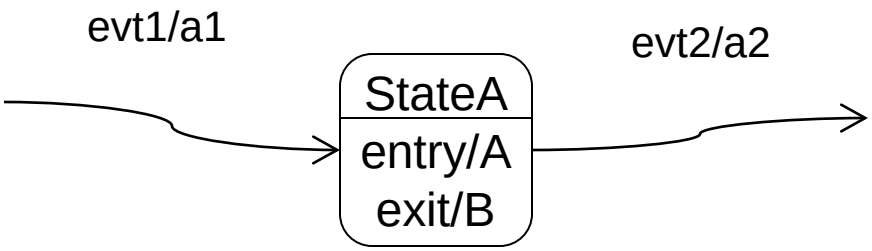
event1, event2, event3

Actions executed

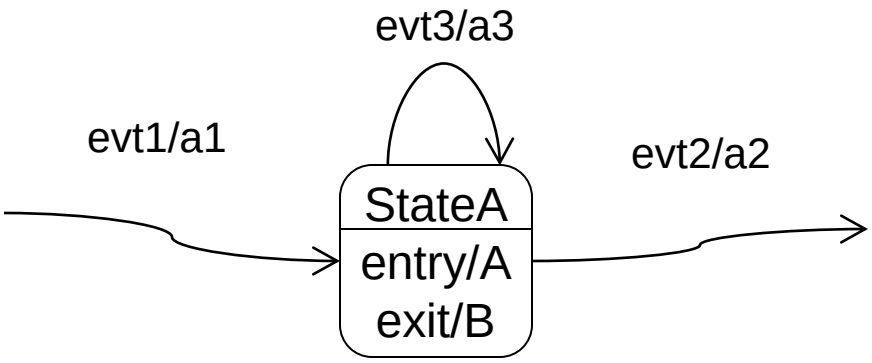
actionC, actionA, actionB, actionD, actionA, actionB, actionE

Example

Result if sequence evt1,evt3,evt2 is received



- a1, A, B, a2 (evt3 is ignored)

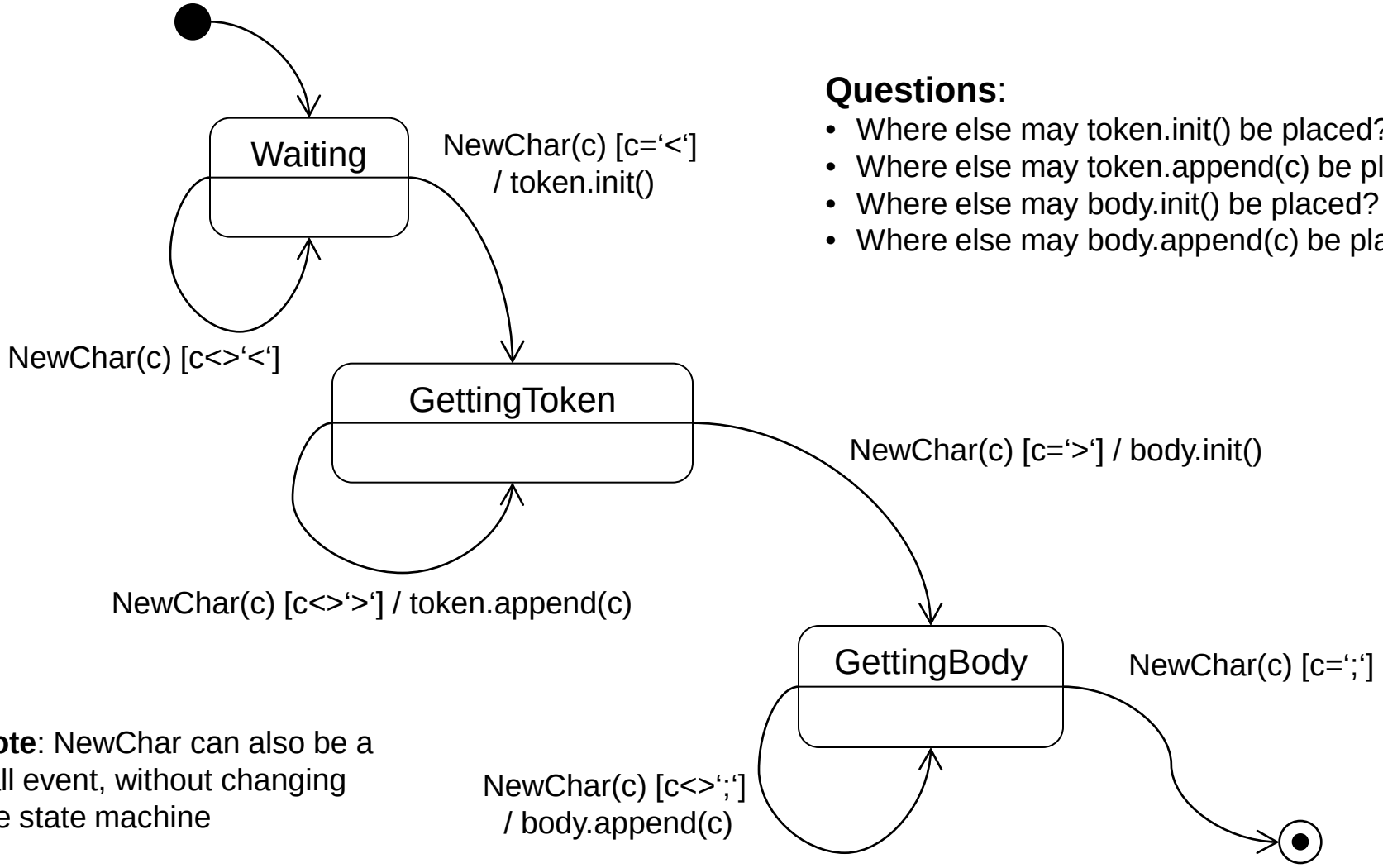


- a1, A, B, a3, A, B, a2

Example—Parsing

- Statechart for parsing a simple language such as: ‘<’ string ‘>’ string;
  - Do not do anything until character ‘<‘ is found
  - Add the string between ‘<‘ and ‘>’ to a `token` string:
    - token.init() empties the string
    - token.append(c) appends character c to string token
  - Add the string after the ‘>’ (but before ‘;’) to a `body` string
- We model a reactive class:
  - The class only waits for signal events to trigger behaviour, i.e., state changes
  - The event is (a <<signal>> class) called NewChar with one attribute
  - Recall the discussion on sequence diagrams:
    - the trigger can be the name of the signal (class) with arguments which match the attributes of the class definition
    - Such arguments are available to the guard condition, as well as to actions

Example—Parsing (cont.)



Questions:

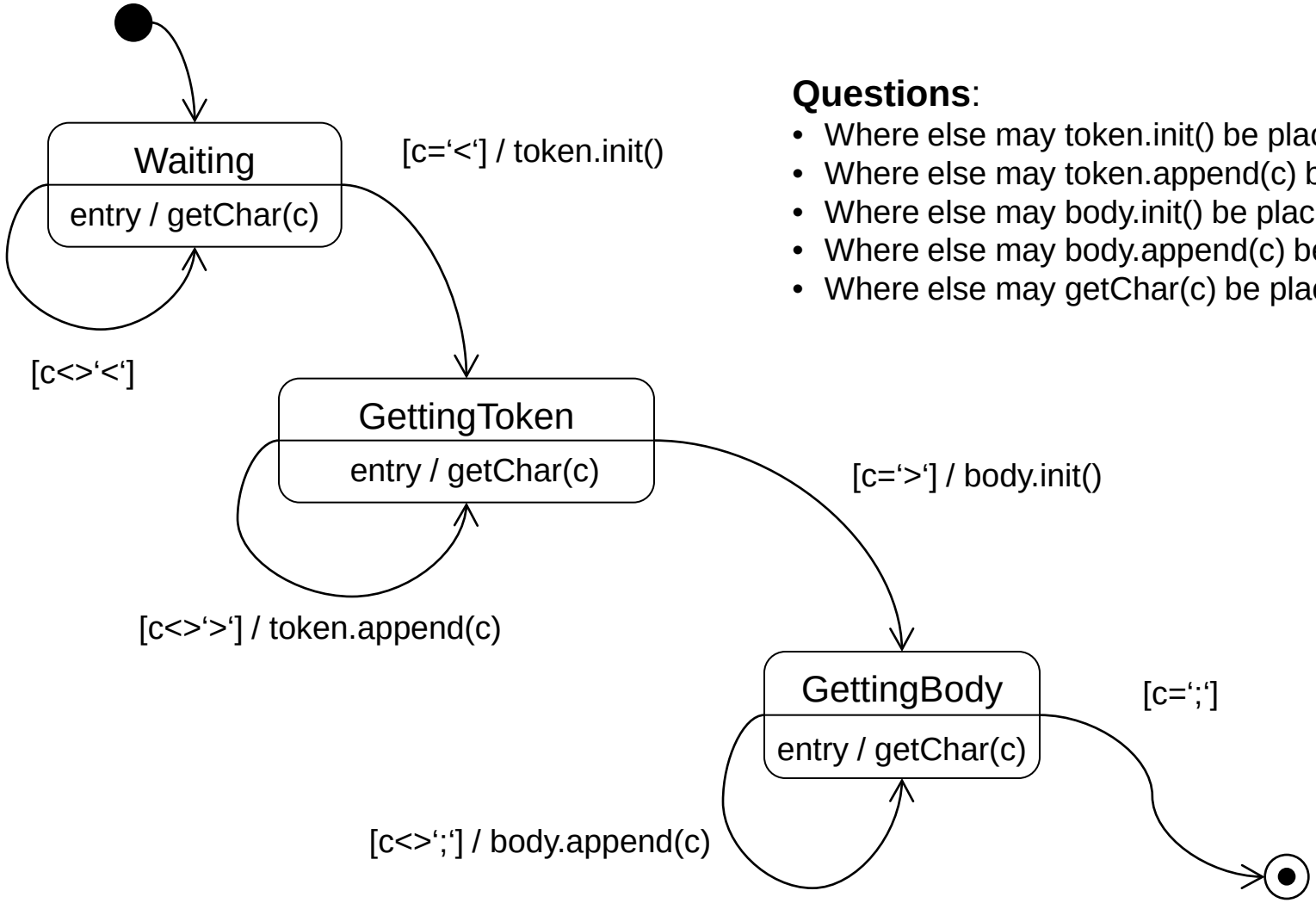
- Where else may token.init() be placed?
- Where else may token.append(c) be placed?
- Where else may body.init() be placed?
- Where else may body.append(c) be placed?

Note: NewChar can also be a call event, without changing the state machine

Example—Parsing (cont.)

- Same behaviour to model
- We model a class which, once started, can handle parsing by itself:
  - It does not need to wait for signals
  - It can get characters, one at a time, using action getChar(c)

Example—Parsing (cont.)

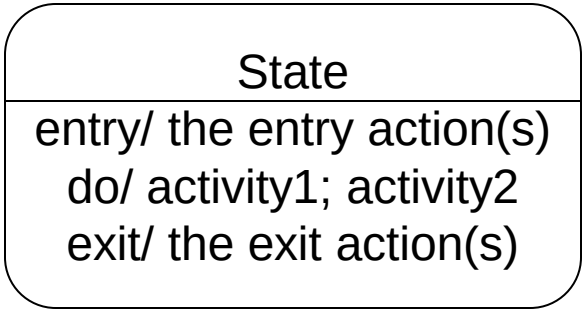


Questions:

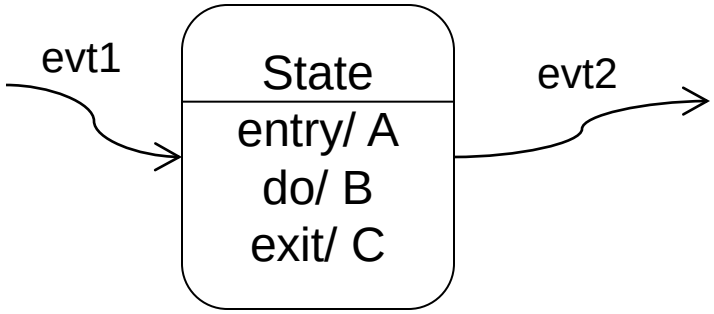
- Where else may token.init() be placed?
- Where else may token.append(c) be placed?
- Where else may body.init() be placed?
- Where else may body.append(c) be placed?
- Where else may getChar(c) be placed?

State Activity

- State Activity: models an ongoing work in a state.
  - The object does some work
    - that will begin after the entry action is finished,
    - that continues until it is interrupted by an event,
    - or that terminates (if not interrupted) before the exit action is triggered .
- A state activity can be interrupted by an event



Example

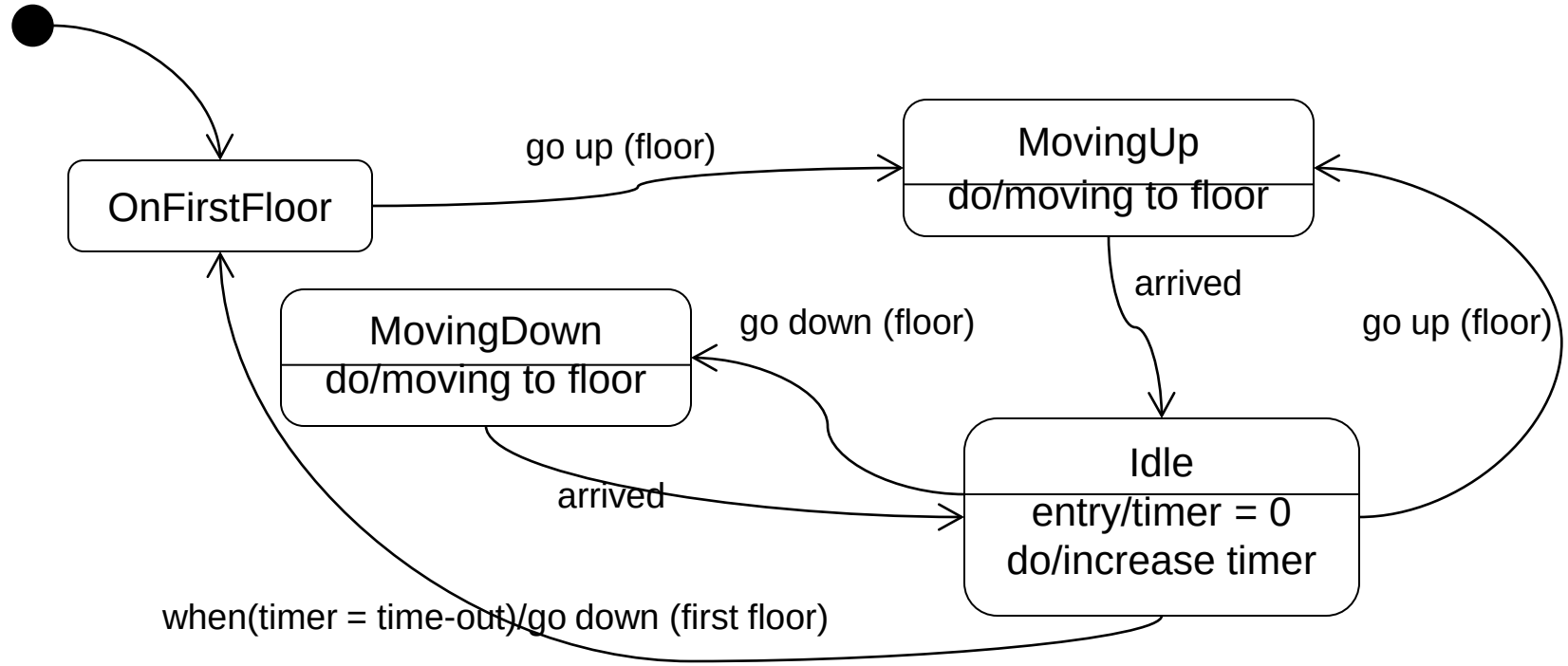


Event evt1 is first received and

- Event evt2 is received after completion of activity B
  - A, B, C
- Event evt2 is received before completion of activity B
  - A, part of B, C

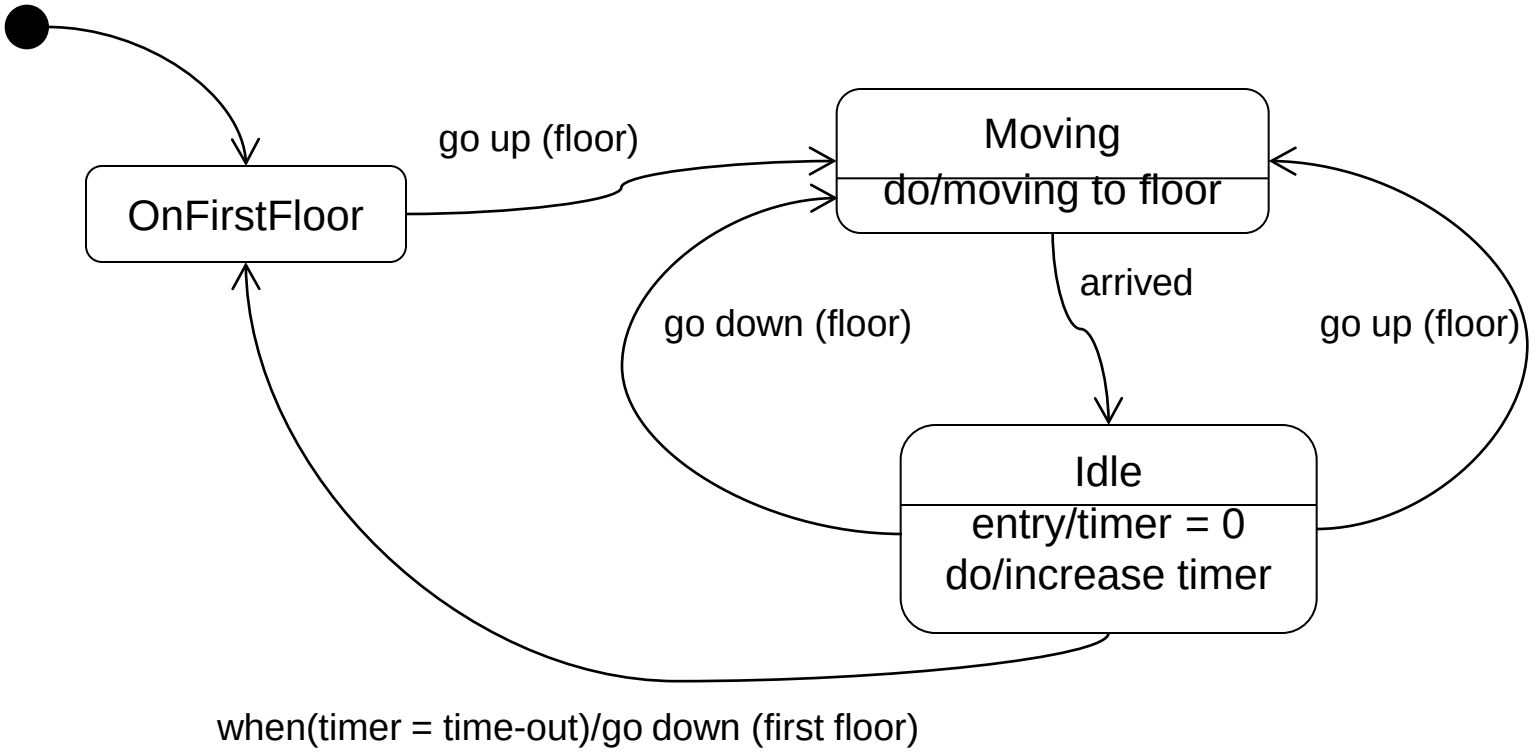
Example—Elevator

A statechart for an elevator. The elevator starts at the first floor. It can be moving up and down. If the elevator is idle on one floor, a time-out event occurs after a period of time and moves the elevator back to the first floor.



Example—Elevator (cont.)

Two states are merged



Completion Event

- A *completion event* is generated when all entry and do (activity) behaviors within the state are complete.
- If the state is connected to another state by a transition that has no label, then this transition (a.k.a., completion transition) fires.
  - the object automatically makes a transition to the state that comes after executing any exit action.
  - A completion transition can have a guard, actions.
- Recall the parsing example.

State-Based Modeling

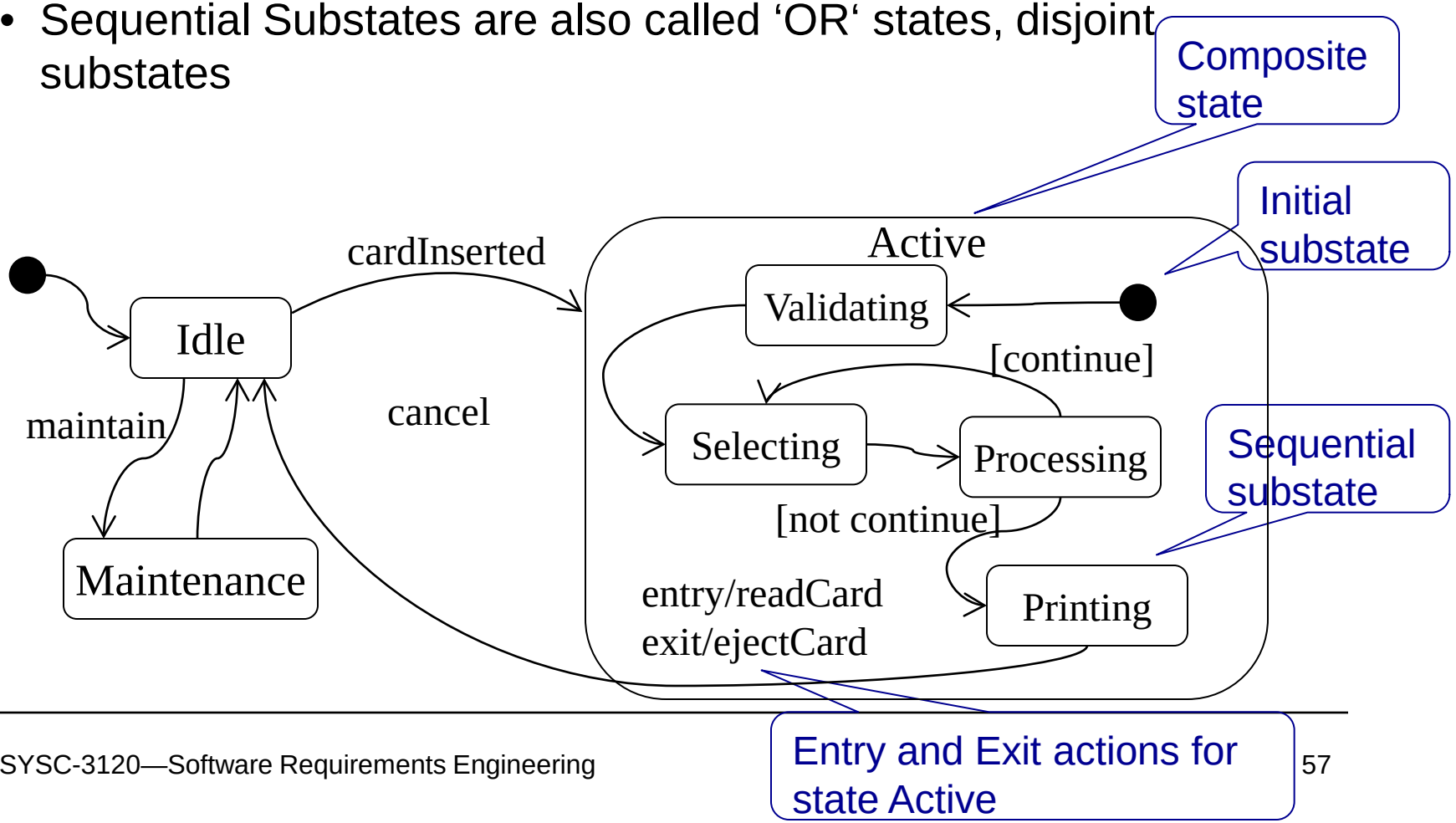
- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

Substates

- A substate is a state nested inside another state
  - A **simple state** is a state without any substate
  - A state with substates is a **composite state**
- There are two different kinds of substates (i.e., composite states):
  - Sequential substates
  - Concurrent substates
- Common features
  - Substates must have a default starting (initial) substate, and one or several final (terminating) substates
  - All substates fulfill the state invariant from their parent state
    - The invariant of the sub-sate implies the invariant of the parent state
  - All substates inherit outgoing transitions from their parent
    - Same responses
  - But: incoming transitions for parents are not inherited

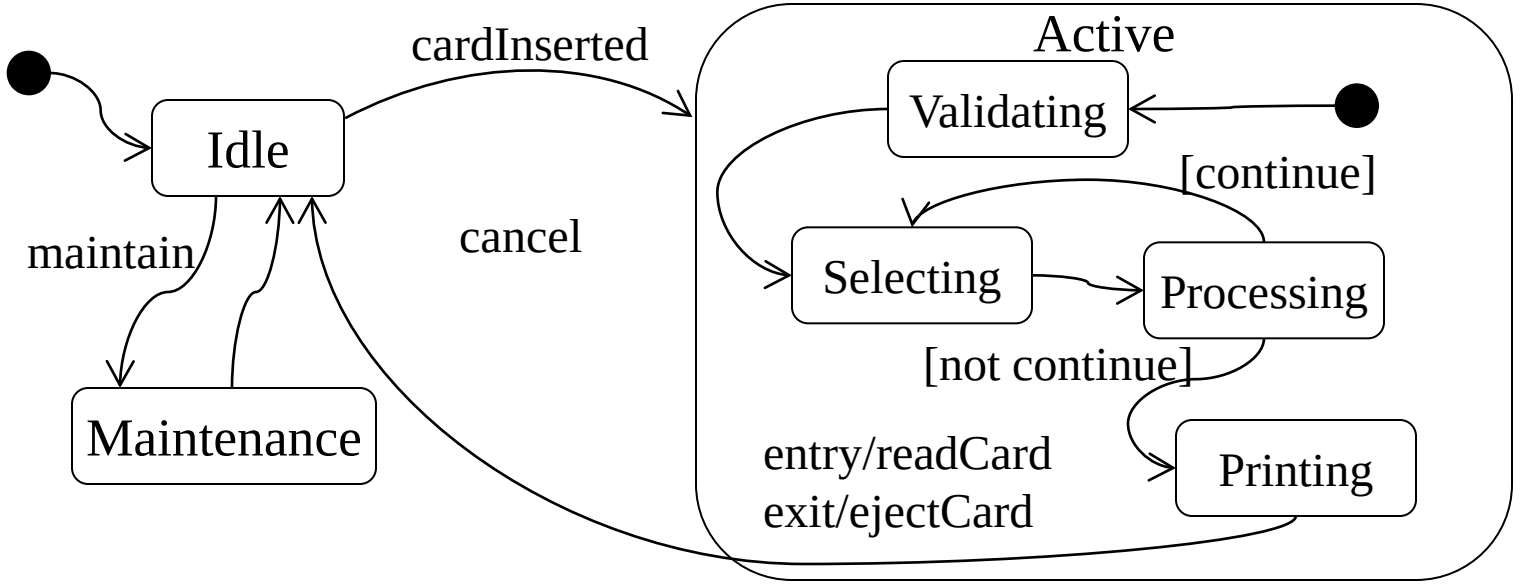
Sequential substates

- An object in a composite state can only be in one of that state's substates at the same time (mutually exclusive, distinguishable)
- Sequential Substates are also called 'OR' states, disjoint substates



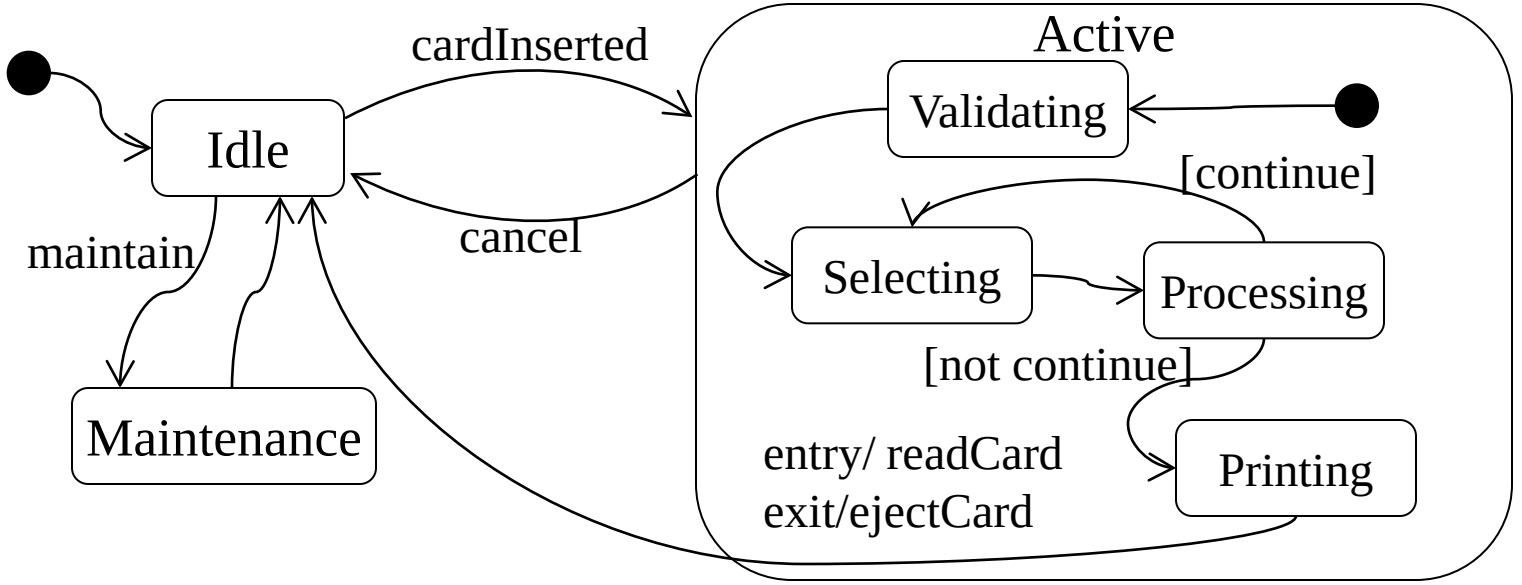
Sequential substates (cont.)

- The state machine is either in Idle, Maintenance, or Active (composite state)
- While in Active, the state machine is either in Validating, Selecting, Processing, or Printing (sequential substates)
- Overall, the state is either Idle, Maintenance, (Active)Validating, (Active)Selecting, (Active)Processing, or (Active)Printing

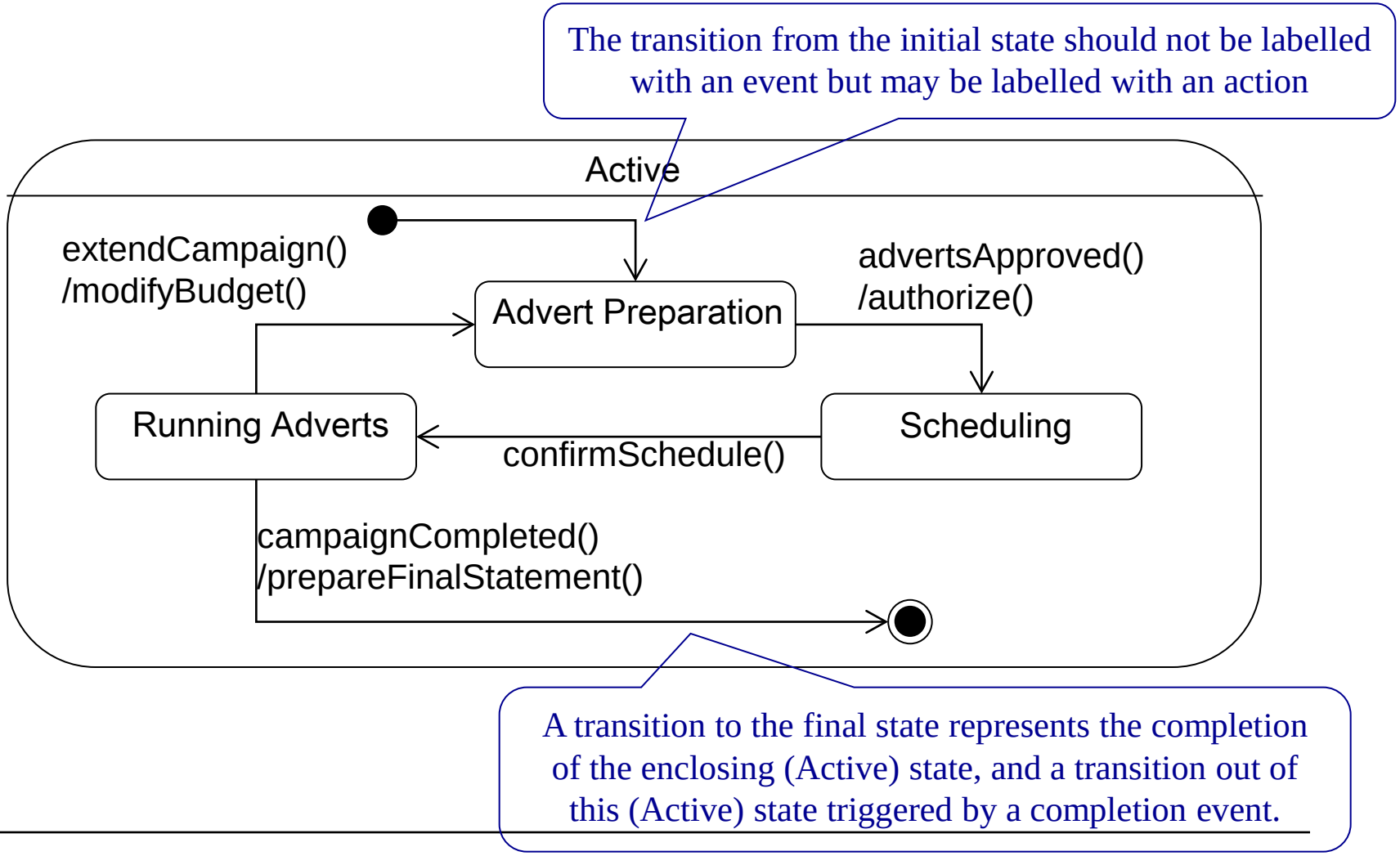


Sequential substates (cont.)

- If, while in Idle, event cardInserted is received, the new state is the initial state in composite state Active, leading to state (Active)Validating
- If, while in Active, event cancel is received, the new state is Idle
  - Being in Active means being in either Validating, Selecting, …
  - ➢ If, while in Validating, Selecting, …, event cancel is received, the new state is Idle
- After finishing Printing, the new state is Idle (after ejecting the card)

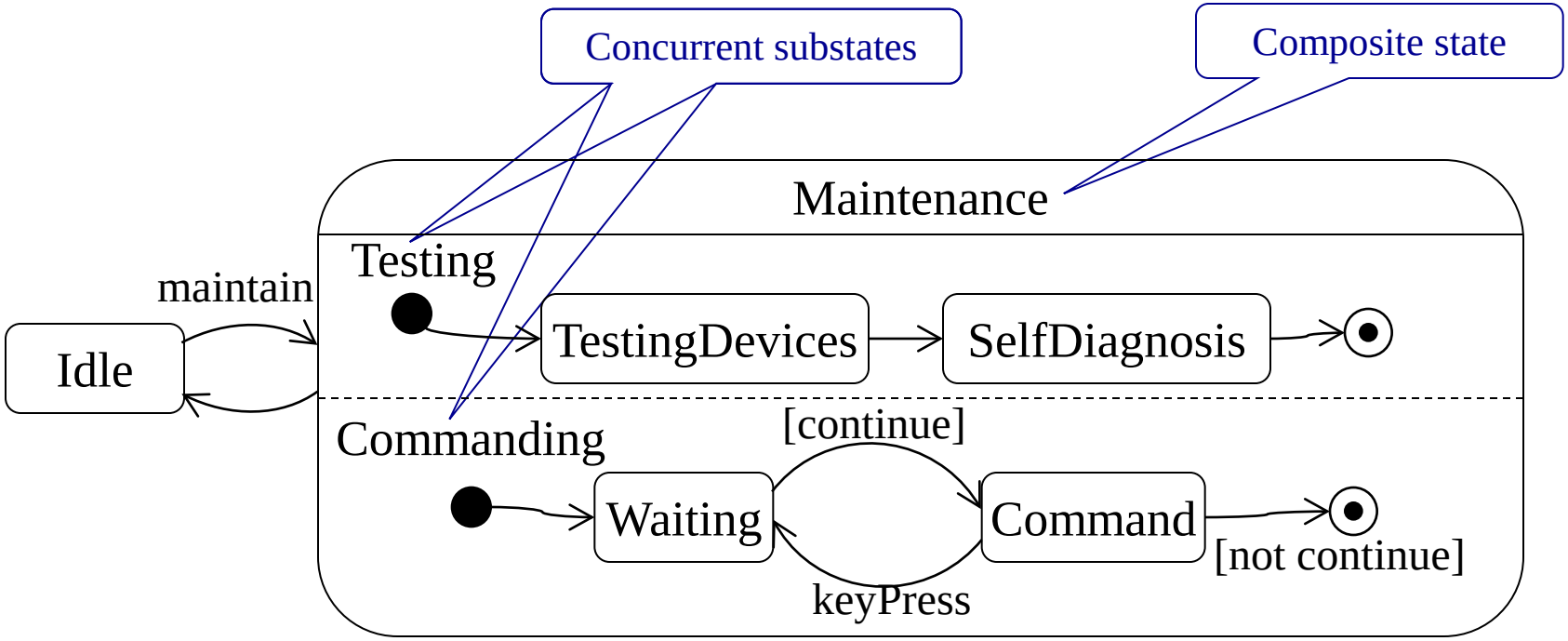


Initial state of Composite State



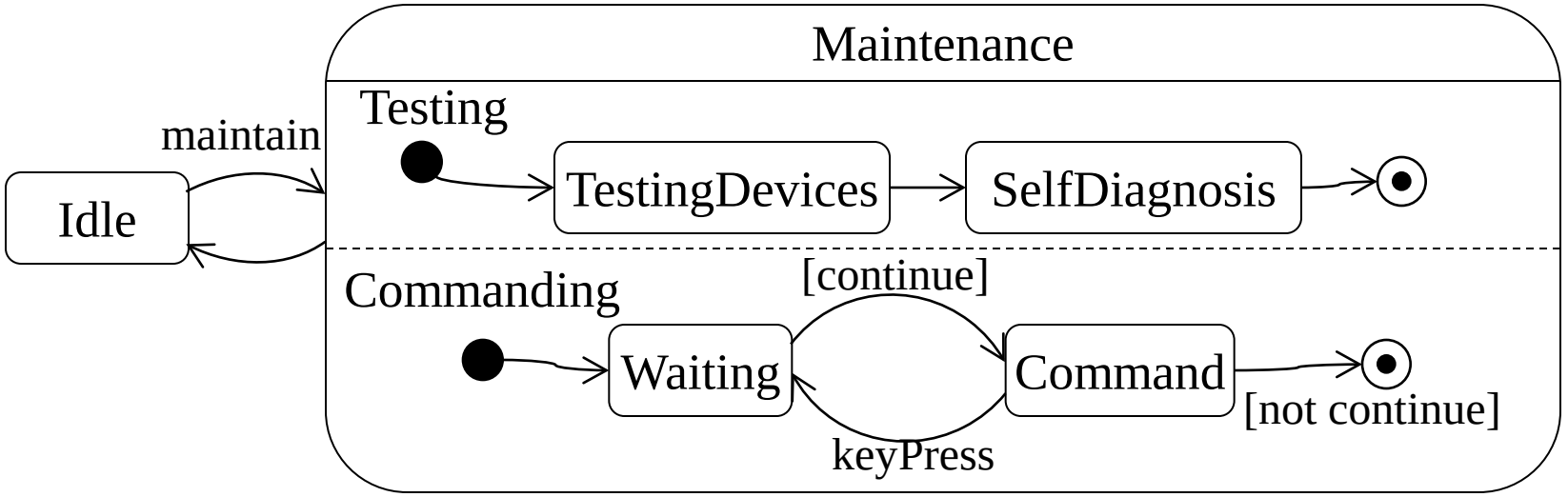
Concurrent substates

- If an object can be in a composite state but can also be in more than one of that state's substates at the same time
  - Subtates active simultaneously
- Concurrent Substates are also called 'AND' states, orthogonal substates

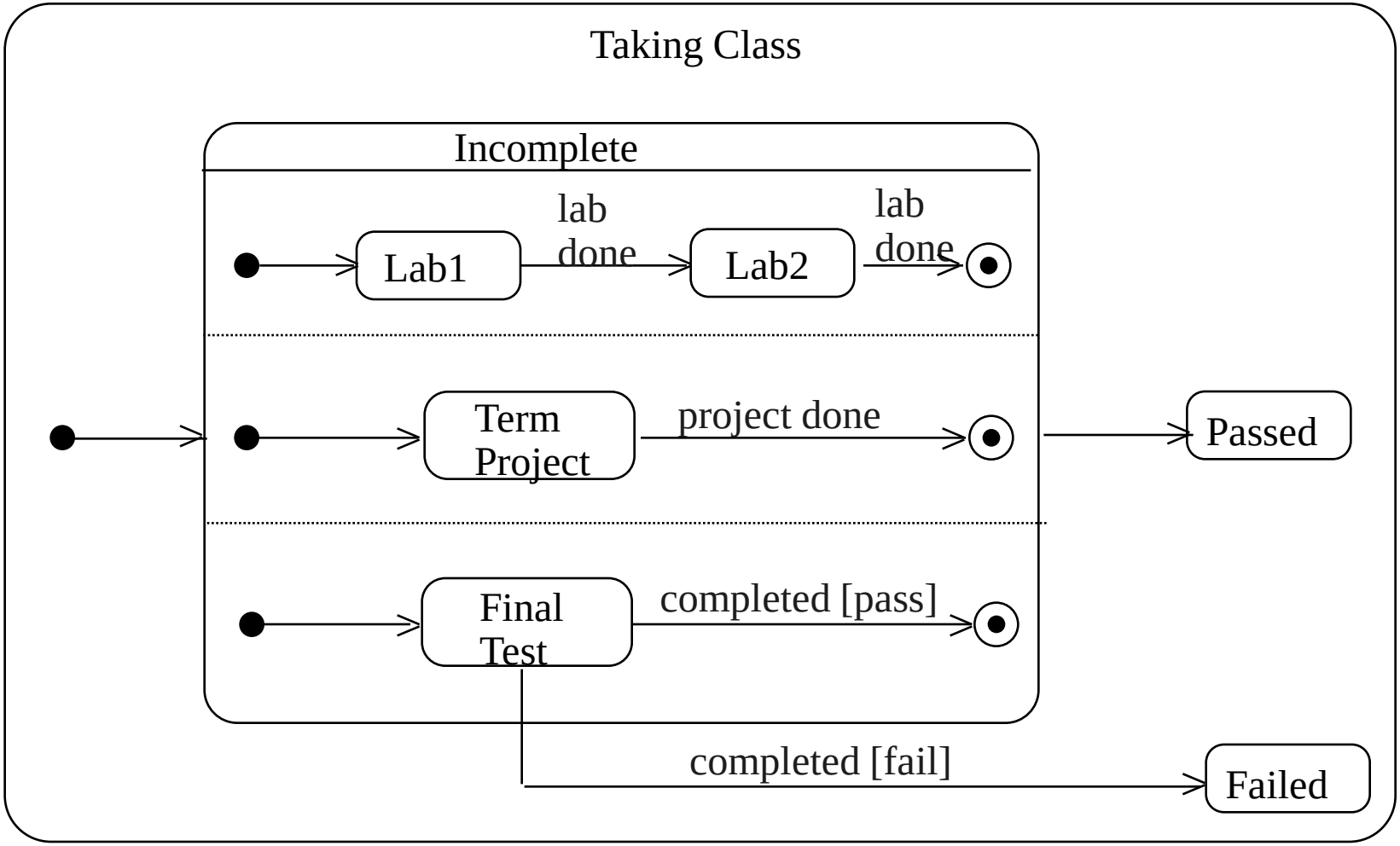


Concurrent substates (cont.)

- While in state Maintenance, the state machine is **at the same time** in states Testing and in state Commanding
- While in state Testing, the state machine is **either** in TestingDevices or SelfDiagnosis



Example

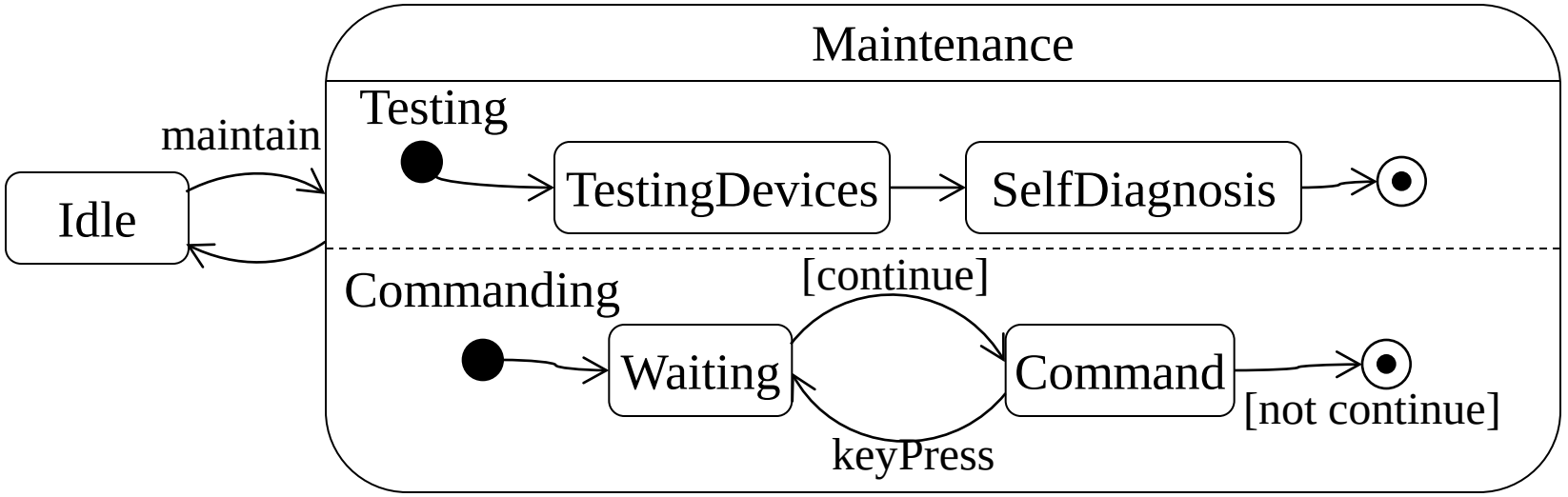

Taking Class
Incomplete
Lab1
lab done
Lab2
lab done
Term Project
project done
Passed
Final Test
completed [pass]
completed [fail]
Failed

Concurrent States—Initial state

- A transition to a composite state made of concurrent substates is equivalent to a simultaneous transition to the initial states of each concurrent statechart
- An initial state must be specified in both nested (concurrent) state machines in order to avoid ambiguity about which substate should first be entered in each concurrent region

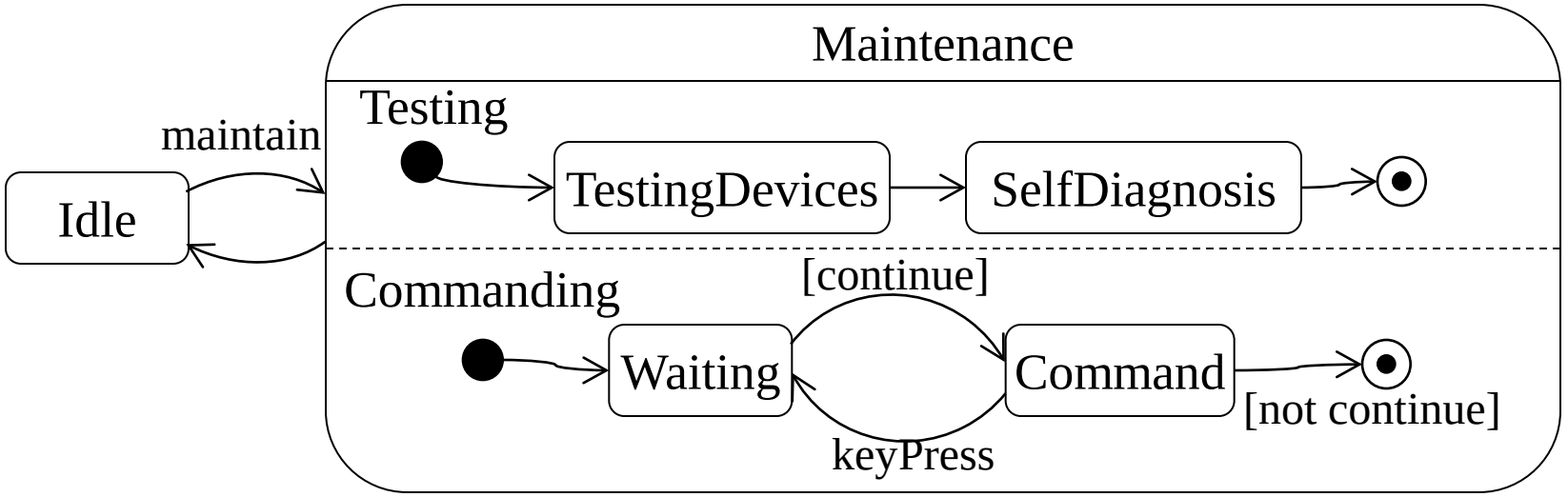
Concurrent States—Initial state

- While in state Idle, if event maintain is received, the new state is Maintenance.
- This means entering simultaneously into Testing and Commanding
- Which means being simultaneously in TestingDevices and Waiting

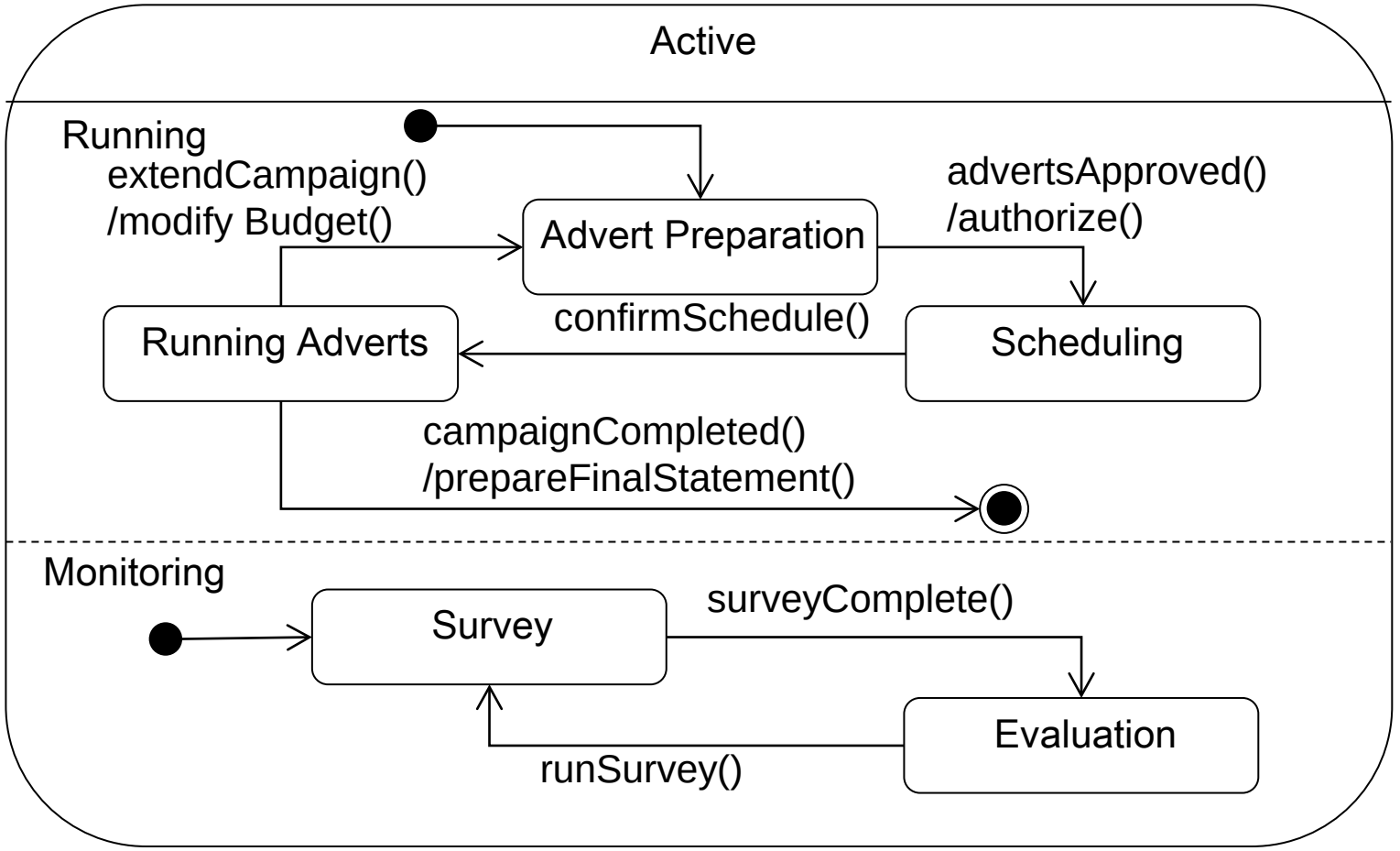


Concurrent States—Concurrent Beheviour

- Once the composite states is entered a transition may occur within
  - either one of the concurrent regions without having any effect on the state in the other concurrent region
  - in all concurrent regions at the same time
- If the current state is TestingDevices+Waiting, the next state may be TestingDevices+Command.



The `Active` State with Concurrent Substates



Composite State—Entry/Exit

- A composite state can have an entry action, an exit action
  - A composite state does not have a do activity

  (The behaviour while in the composite state is specified by its substates!)
- Each substate can have an entry action, an exit action

- Entering a composite state:
  - entry action of the composite state first
  - entry actions of substates in order of nesting
- Exiting a composite state:
  - exit action of the substates in (reverse) order of nesting
  - exit action of the composite state

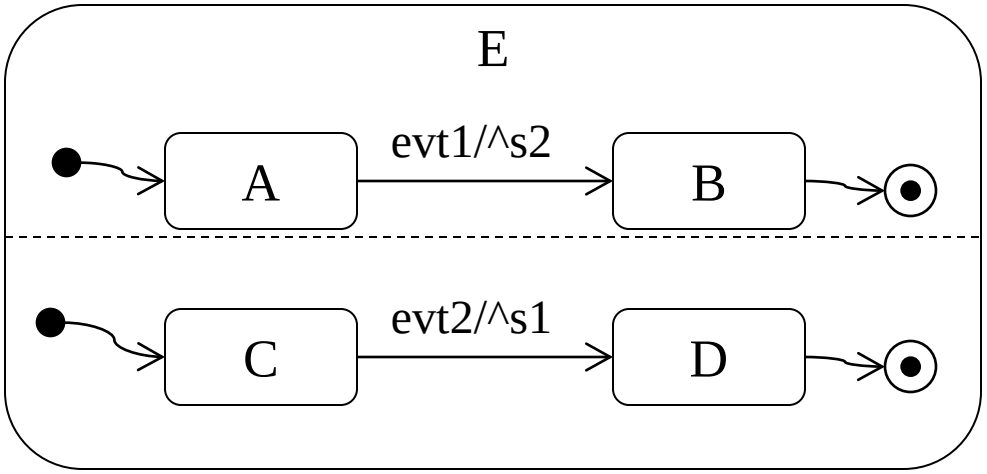
State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

Events Triggered

- Transitions are triggered by events (see previous slides)
- Transitions can also send signals as specified in the *send clauses*, specified after actions and a '^' separator.
- Syntax of a transition:

  eventReveived(args) [condition] / [target.]action(s), … ^
  [target.]signalSent(args), …



Semantics of Send Clauses

- An object can send a signal to any set of objects which it knows about
- May have a designation string (target), e.g., OCL navigation expression, that that indicates the object or class that will receive the signal

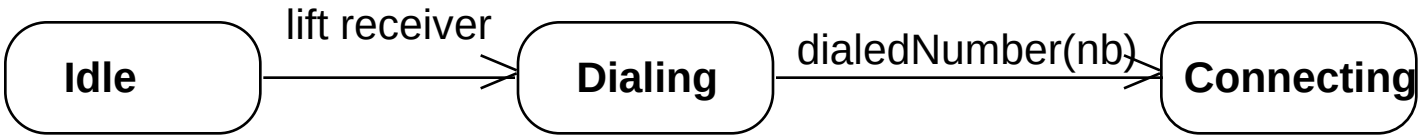
^target.signal(arglist)

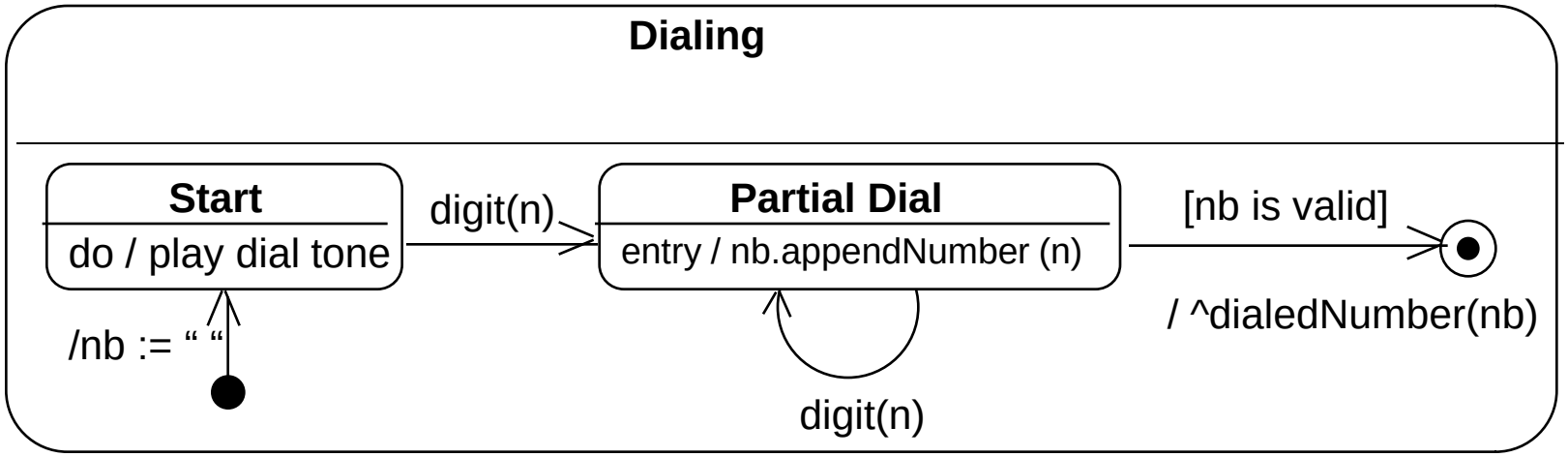
- Must have a name or identifier string that represents the name of a signal
- May have an argument list (actual parameters)
- May refer to parameters of the triggering event and to attributes and links of the object that owns the statechart

Sent Signals versus Actions

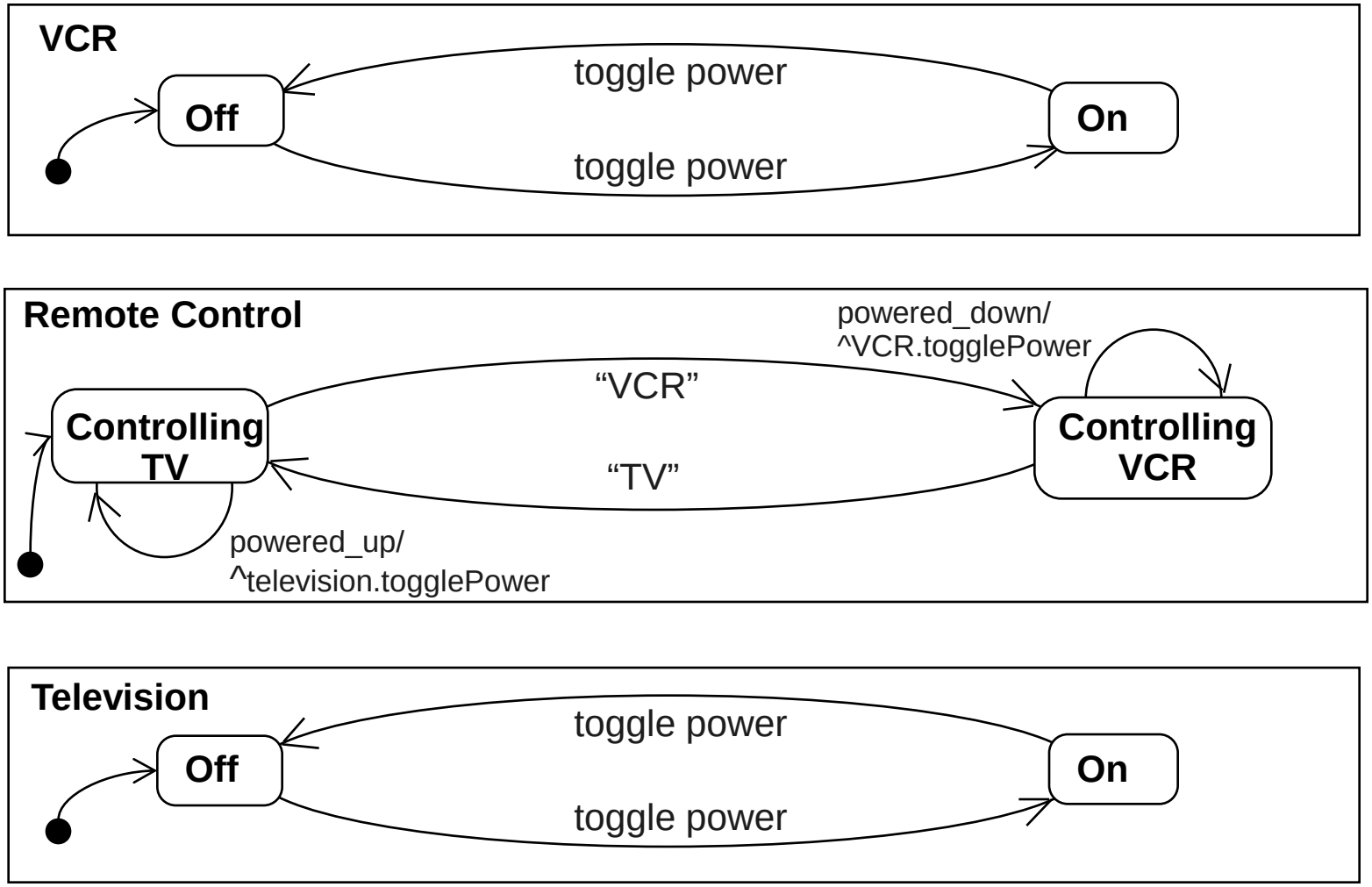
- Sent clauses result into the asynchronous sending of signals to other objects whereas actions are synchronous
- The “asynchronous” property is the main difference with actions on transitions. Actions on transitions, entry, and exit must be completed before entering the new state. In the send clauses, signals are sent and the resulting operations are executed in an asynchronous manner.

Substate Example





Sending Events Between state machines

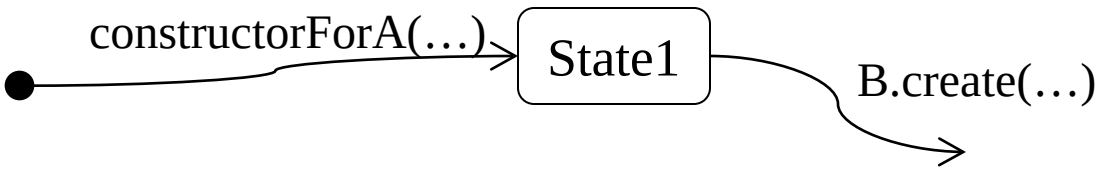


Creating and Destroying Objects

- Creating/Destroying an object:
  - regarded as sending an event to the class itself
- Comments:
  - the event arguments serve to initialize the object
  - new objects begin in an initial state from where they receive the creation event as their first event
  - a class could have multiple possible “birth” events
  - an object ceases to exist (is destroyed) when it reaches a top-level terminal state
  - as part of this transition it can send an event
  - the action and event expression can be attached directly to the terminal state

Creating and Destroying Objects

- Statechart for class A, who ‘knows’ class B



State-Based Modeling

- State-Based Behaviour?
- States
- Transitions
  - Triggering events and Guards
  - Execution semantics
  - Actions and activities
- Substates
- More on events on transitions
- Guidelines

Guidelines for Developing state machines

- A state must reflect an identifiable situation or an interval of time when something is happening in the system. A state name is often an adjective (Initial) or a phrase with an adjective (Elevator idle).
- Each state must have a unique name.
- It must be possible to exit from every state. It is often the case that state machines do not have a terminating state.
- Do not confuse events and actions. An event *cause* a state transition. The action is the *effect* of a state transition.
- Events indicate that something just happened whereas actions are commands.

Guidelines for Developing state machines II

- An action executes “instantaneously”. An activity executes throughout a given state.
- You may have more than one action associated with a transition.
- All these actions conceptually execute simultaneously; hence no assumptions can be made about the order in which the actions are executed. Consequently, no interdependencies should exist among the actions or you need to introduce an intermediate state.
- If a state transition is labeled event[condition], a state transition takes place only if, *at the moment the event happens*, the condition is true.
- Actions, activities, and conditions are optional. Use them only when necessary.