

UML State Machines

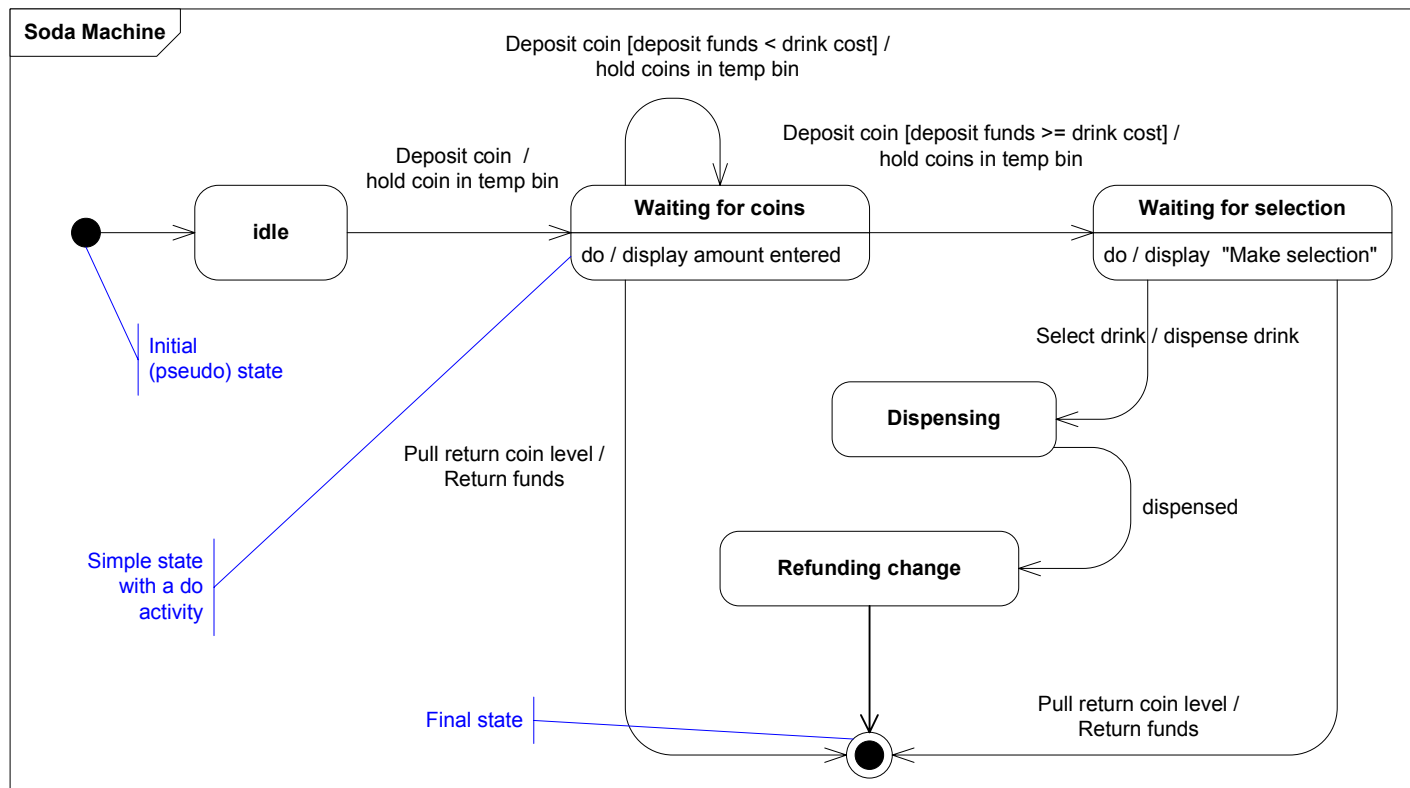
(UML Statecharts)

Introduction

- The StateMachine package defines a set of concepts that can be used for modeling discrete behavior through finite state-transition systems.
- Two kinds of state machines:
 - Behavioral state machines
 - used to specify behavior of various model elements (an object-based variant of Harel statecharts)
 - Protocol State machines
 - used to express usage protocols (they express the legal transitions that a classifier can trigger).

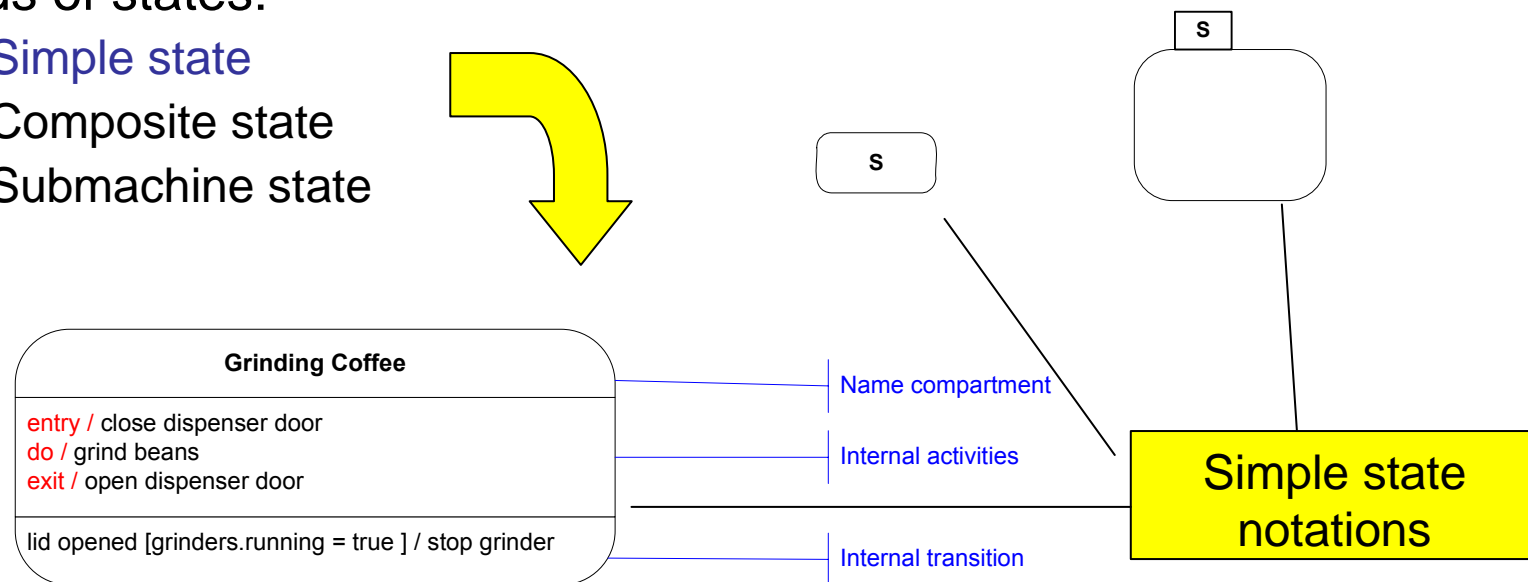
Behavioral State Machines

- Represents the behavior of a piece of a system using graph notation.



States

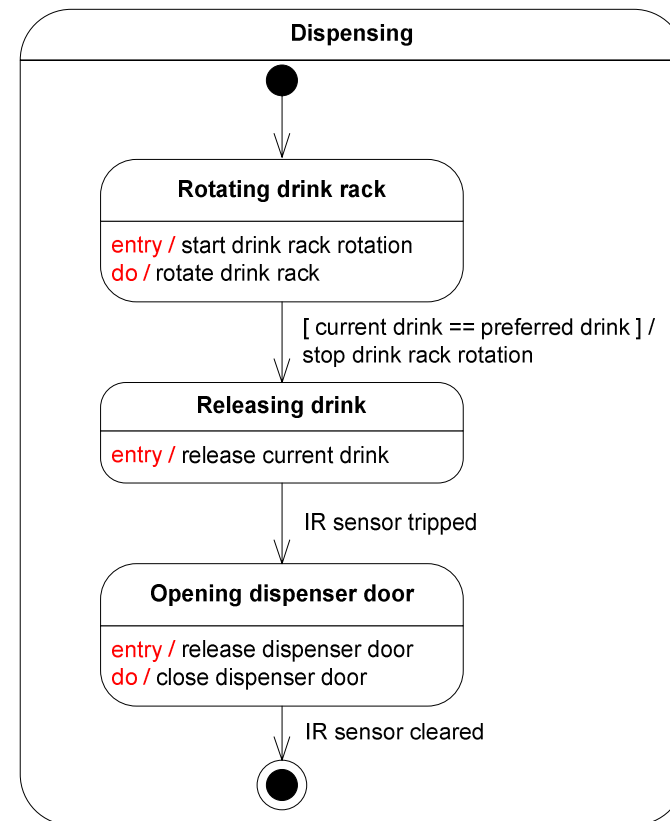
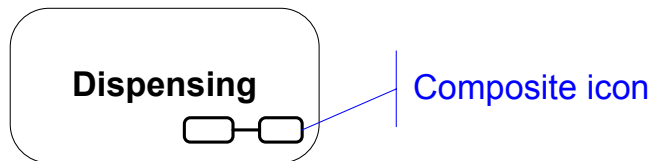
- A state models a situation during which some (usually implicit) **invariant condition** holds.
- The invariant may represent a static situation such as an object **waiting for some external event to occur**.
- However, it can also model dynamic conditions such as **the process of performing some behavior**.
- Kinds of states:
 - Simple state
 - Composite state
 - Submachine state



A state with compartments

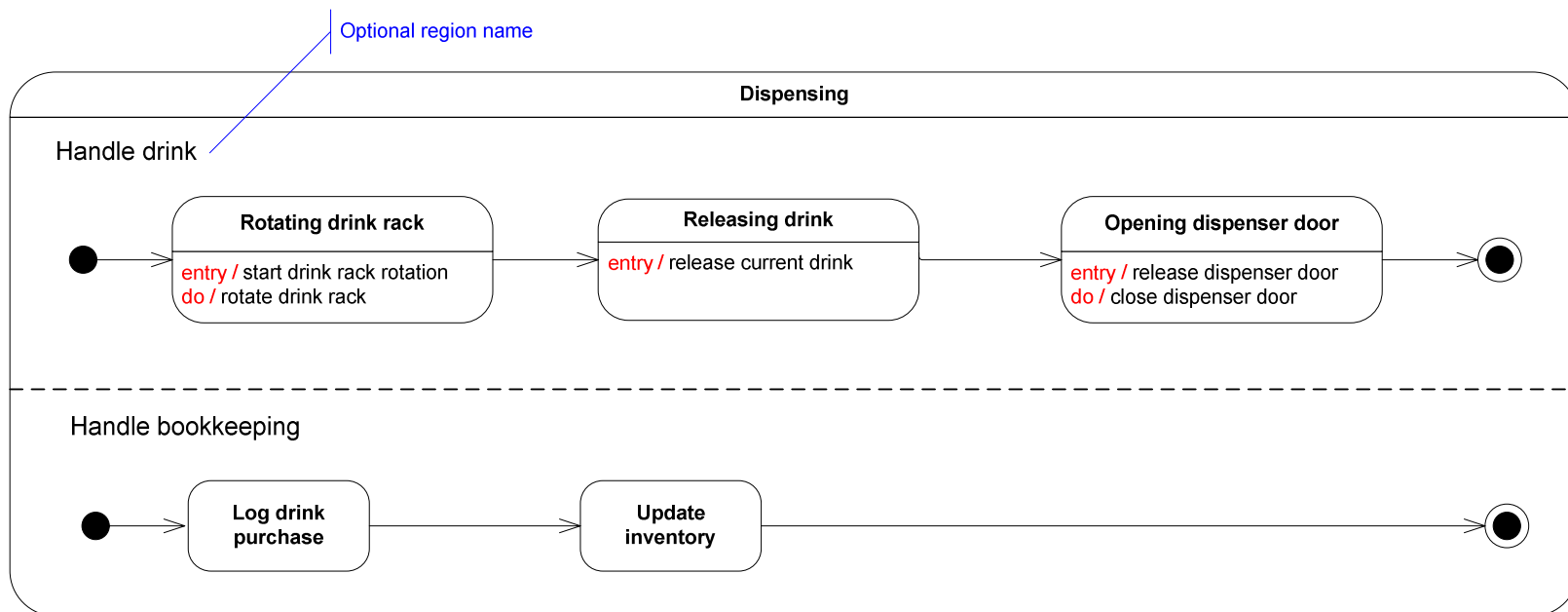
Composite states

- A **composite state** is a state with one or more regions.
- A **region** is simply a container for substates.
- A composite state with two or more regions is called **orthogonal**.
- Decomposition compartment.



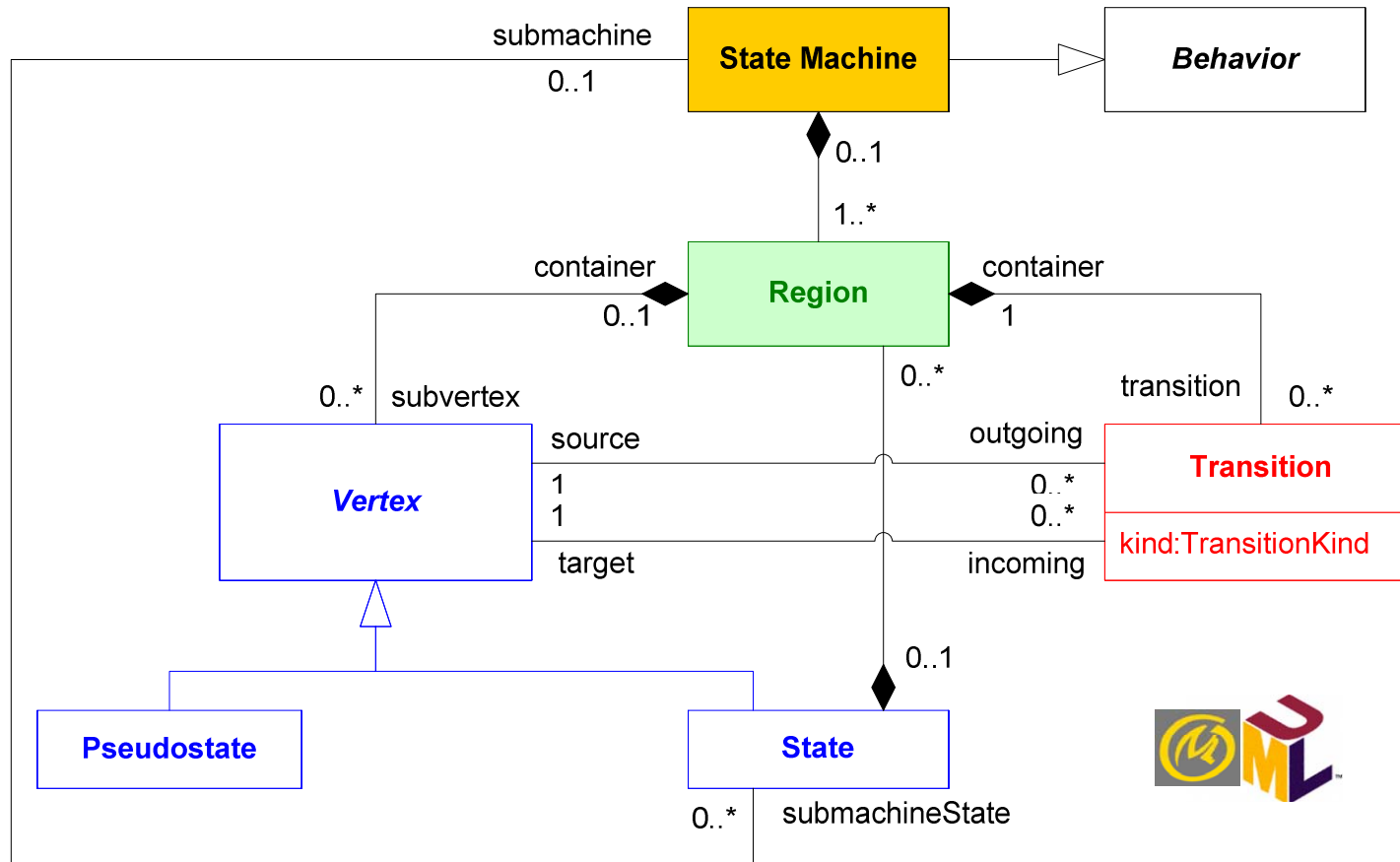
Composite states / regions

- Each region within a composite state **executes in parallel**.
- A transition to the final state of a region indicates **completing the activity for that region**.
- Once all the regions have completed, the composite state triggers a **completion event** and a **completion transition** (if one exists) triggers.



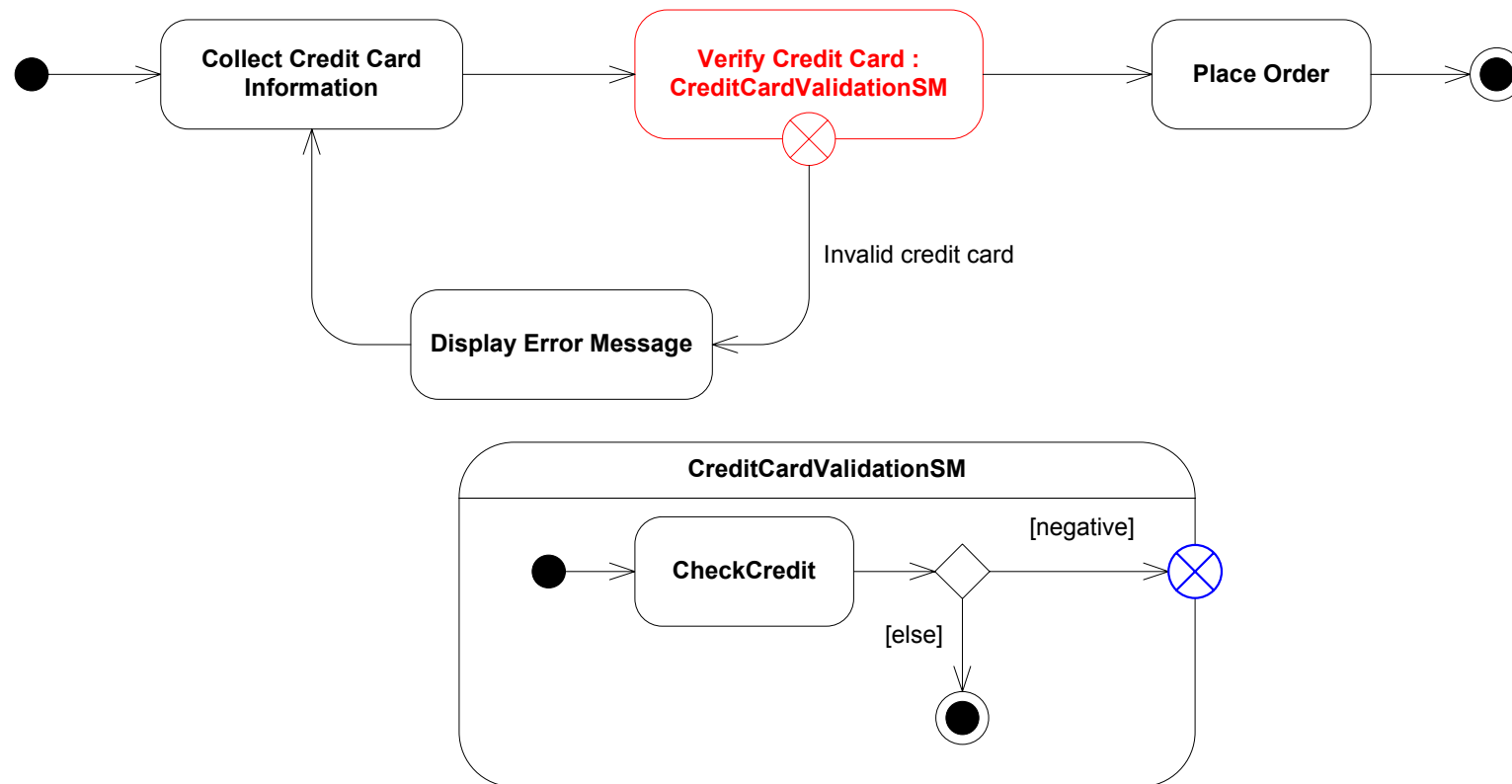
Composite state with two regions

Simplified State Machine meta-model



Submachine states

- Submachine states are semantically equivalent to composite states in that they are made up of internal substates and transitions.
- A way to **encapsulate** states and transitions so that they can be **reused**.

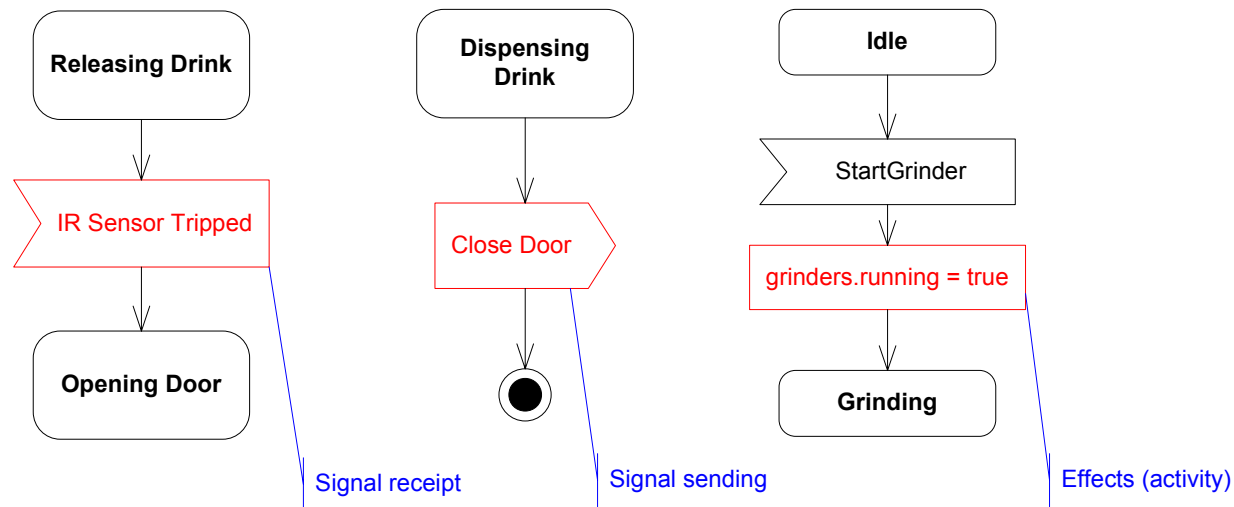


Transitions

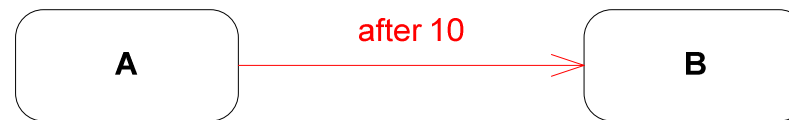
- A transition shows the relationship, or path, between two **states** or **pseudo states**.
- Label on the transition:
`trigger [guard] /effect`
- Transition types:
 - **Compound transition**
Represents change from one complete SM configuration to another.
Set of transitions, choices, forks, and joins leading to a set of target states.
 - **High-level transition**
A transition from a composite state.
 - **Internal transition**
A transition between states within the same composite state.
 - **Completion transition**
A transition from a state that has no explicit trigger. When a state finishes its do activities, a completion event is generated.

Transitions and signals

- Use of explicit icons to show **signal sending**, **signal receipt**, and **effect activities**.



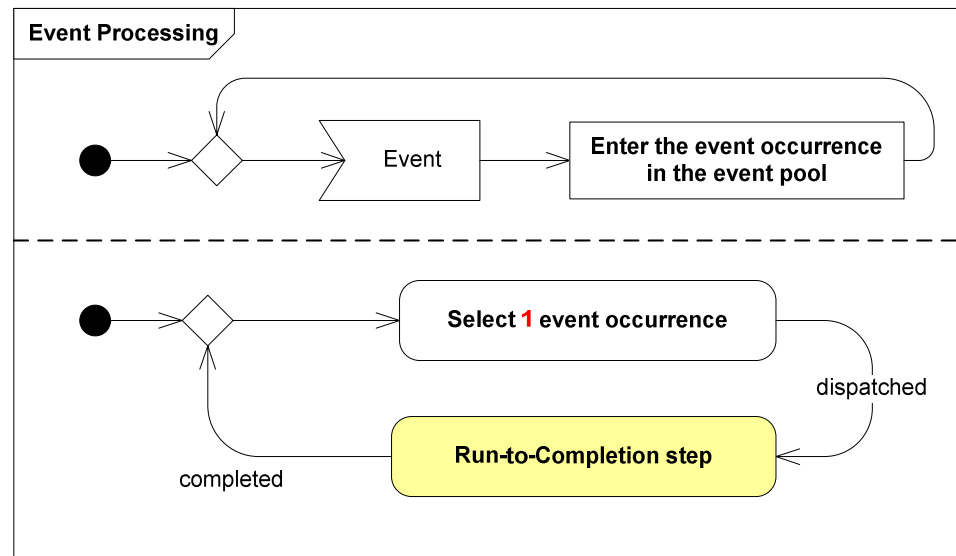
- Time Event:** occurrences linked to time
 - **after 10** relative to entering the current state
 - **at 5** refers to absolute time



Exit state A 10 units of time after entering it

Semantics

- The **event pool** for the state machine is the event pool of the instance according to the behaved context classifier, or the classifier owning the behavioral feature for which the state machine is a method.
- **Event processing - run-to-completion step**
 - Event occurrences are **detected**, **dispatched**, and then **processed** by the state machine, **one at a time**. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.
 - The semantics of event occurrence processing is based on the **run-to-completion** assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.



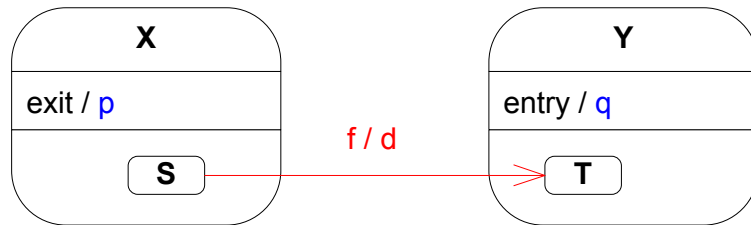
Run-to-completion step

- The **processing of a single event occurrence** by a state machine is known as a *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a **stable state configuration** with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion step* is the **passage between two state configurations** of the state machine.
- When an event occurrence is detected and dispatched, it may result in one or more **transitions** being **enabled for firing**. If **no transition is enabled** and the event (type) is not in the deferred event list of the current state configuration, the event occurrence is discarded and the run-to-completion step is completed.
- **In the presence of orthogonal regions** it is possible to fire multiple transitions as a result of the same event occurrence — **as many as one transition in each region** in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. Which of the enabled transitions actually fire is determined by the transition selection algorithm described below.
- During a transition, a number of **actions may be executed**. If such an action is a synchronous operation invocation on an object executing a state machine, then the transition step is not completed until the invoked object completes its run-to-completion step.
- **Conflicting transitions**
 - It was already noted that it is possible for **more than one transition to be enabled** within a state machine. If that happens, then such transitions may be in **conflict with each other**. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, **only one of them will fire in a single run-to-completion step**.
 - Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed.

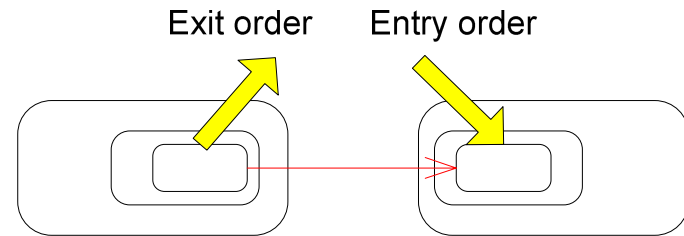
Run-to-completion (cnt'd)

- **Firing priority**
 - In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an *implicit priority*. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their *relative position in the state hierarchy*. **By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states** (Note: It is the converse in SyncCharts in case of strong abortion)
 - The priority of a transition is defined based on its source state. The priority of joined transitions is based on the priority of the transition with the most transitively nested source state.
- **Transition selection algorithm**
 - The set of transitions that will fire is a *maximal set of transitions* that satisfies the following conditions:
 - All transitions in the set are *enabled*.
 - There are *no conflicting transitions* within the set.
 - There is *no transition outside the set that has higher priority* than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out).

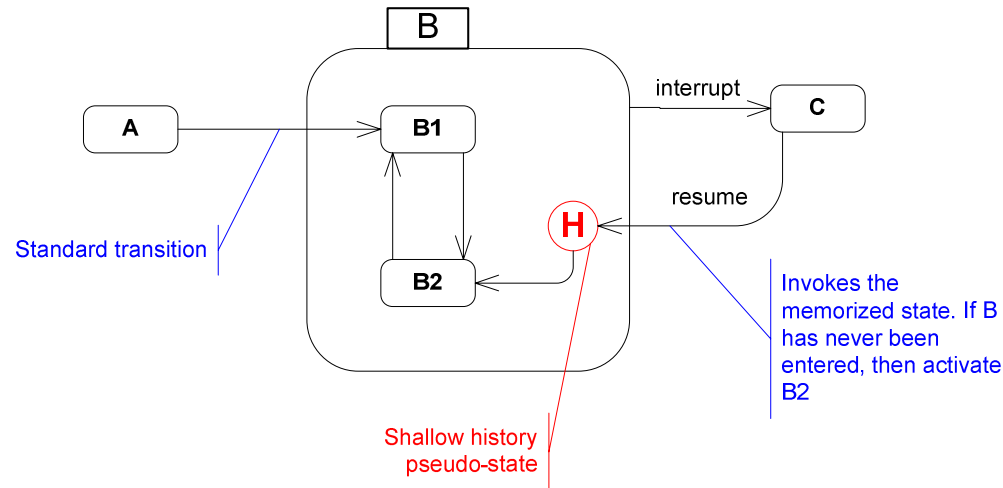
Other cases



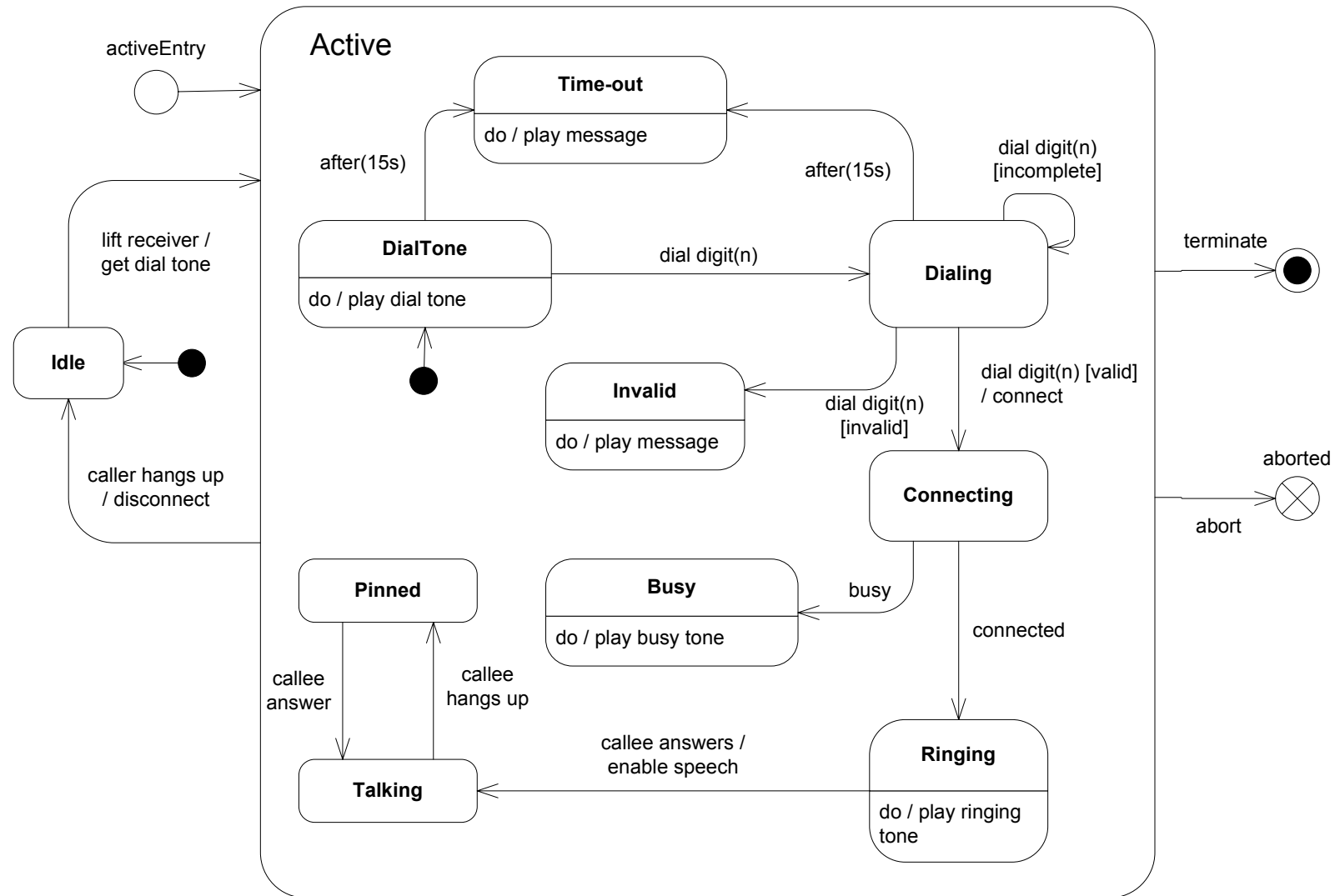
Execution ordering: p; d; q



General case

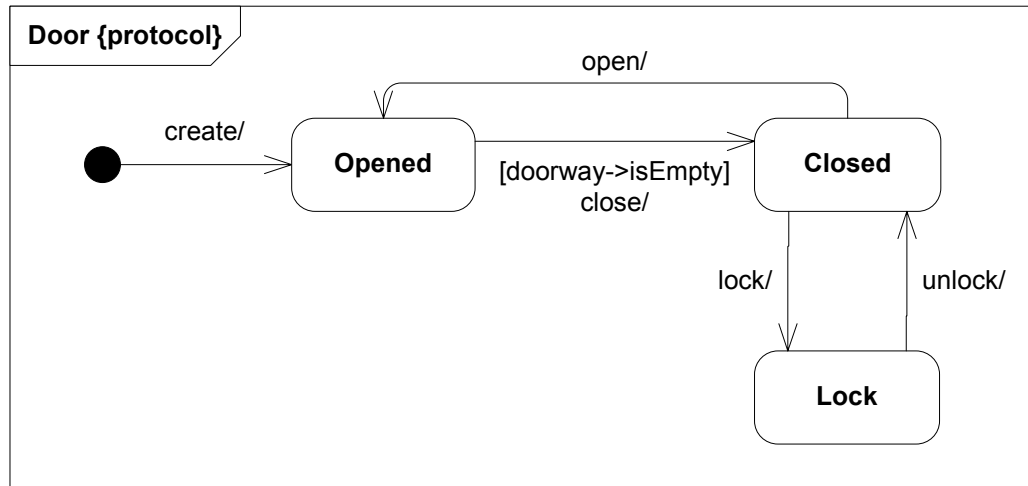


Example of behavioral SM



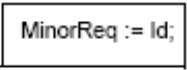
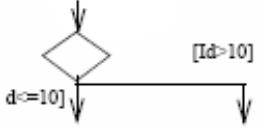
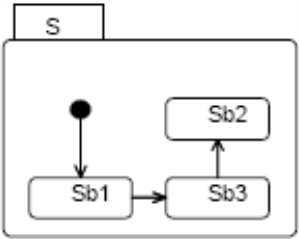




Protocol State Machines




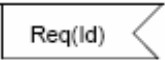
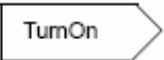
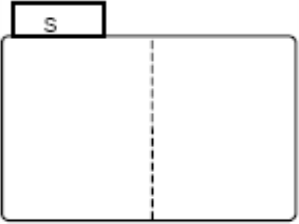
- A *protocol state machine* is always defined in the context of a classifier.
- It specifies which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifier's operations.
- Differences with behavioral SMs:
 - No entry, exit, and do activities
 - States can have invariants (in square brackets under the state name)
 - The keyword {protocol} placed after the SM name
 - Transitions in PSM have a precondition, the trigger and a postcondition:
 - [precondition] name [postcondition]
 - [precondition] name(param-list) [postcondition]
 - Each transition is associated with 0 or 1 operation on the owning classifier
 - No effect activity



Tells only **what state** the protocol implementation will be in, **not what it does** to get there.

Notations Summary

Node Type	Notation
Action	
Choice pseudostate	
Composite state	
Entry point	again 
Exit point	aborted 
Final state	
History, Deep pseudostate	

Node Type	Notation
History, Shallow pseudostate	
Initial pseudostate	
Junction pseudostate	
Receive signal action	
Send signal action	
Region	
Simple state	