Chapter 4
Modeling Real-Time Systems

What are you about to learn?

Knowledge Objectives

- Understand the modeling process.
- Understand the main elements of an UML state chart diagram.
- Understand the main elements of an UML activity diagram.
- Understand how to map an UML state diagram to C code.

Skill Objectives

- Ability to develop an UML state diagram from given requirements.
- Ability to develop an UML activity diagram from given requirements. Model concurrency (together with material covered in a later chapter).
- Ability to manually create C code from a given UML state chart.

4.1 Modeling Overview

A model is a reduced representation of the world.

In this chapter, the purpose of our modeling effort is to:

- Describe requirements
- Describe solution concepts
- Describe solutions
- Analyze solution concepts and solutions

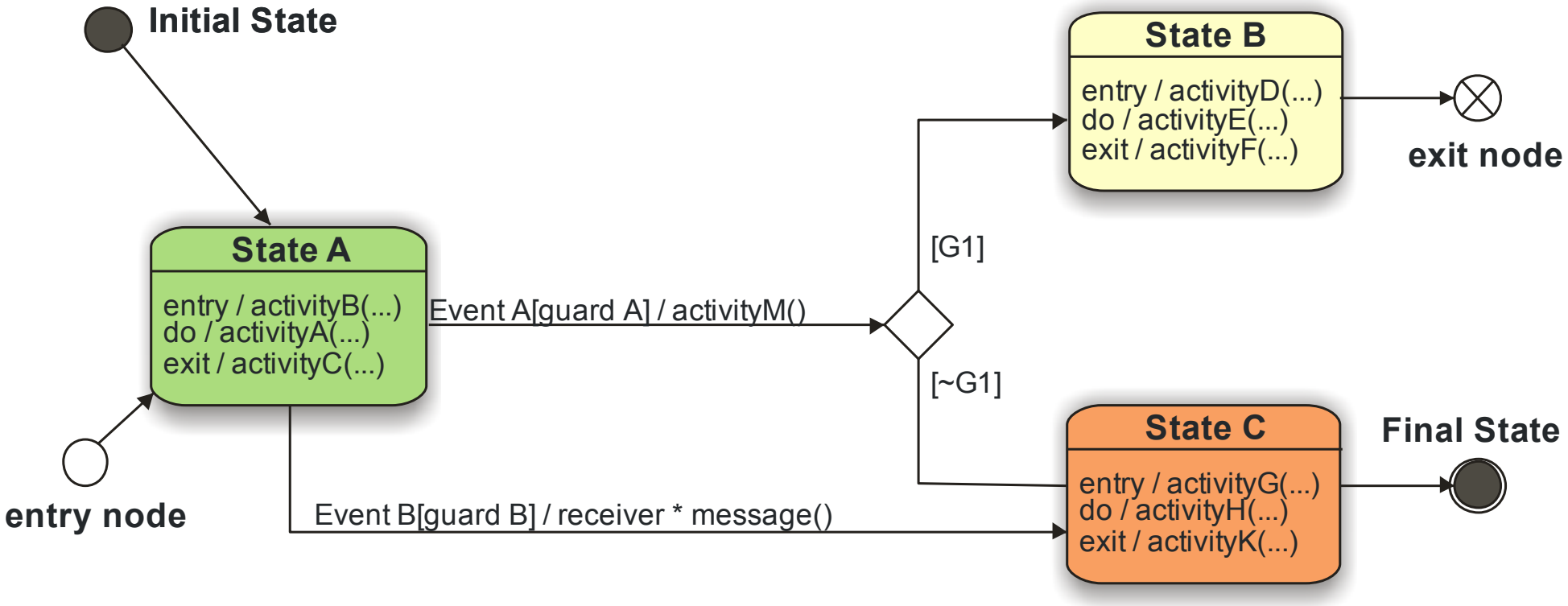
The following methods are commonly used for modeling behavior of real-time systems:

- Petri nets (More academically. Concept has been transferred to UML activity diagrams)
- State diagrams (Harel state charts, UML state charts)
- UML activity diagrams for the description of control and data flow and concurrency
- UML sequence diagrams for the description of execution scenarios
- Timing diagrams (In various notations)

There are additional methods for the development and analysis of control aspects in real-time systems. They will not be treated in this lecture.

4.2 UML State Machines

Many real-time systems can be modelled partly as a finite state machine.



A real-time system is comprised of a number of objects that can relate to each other.

The behavior of many of these objects can be modeled by a state machine.

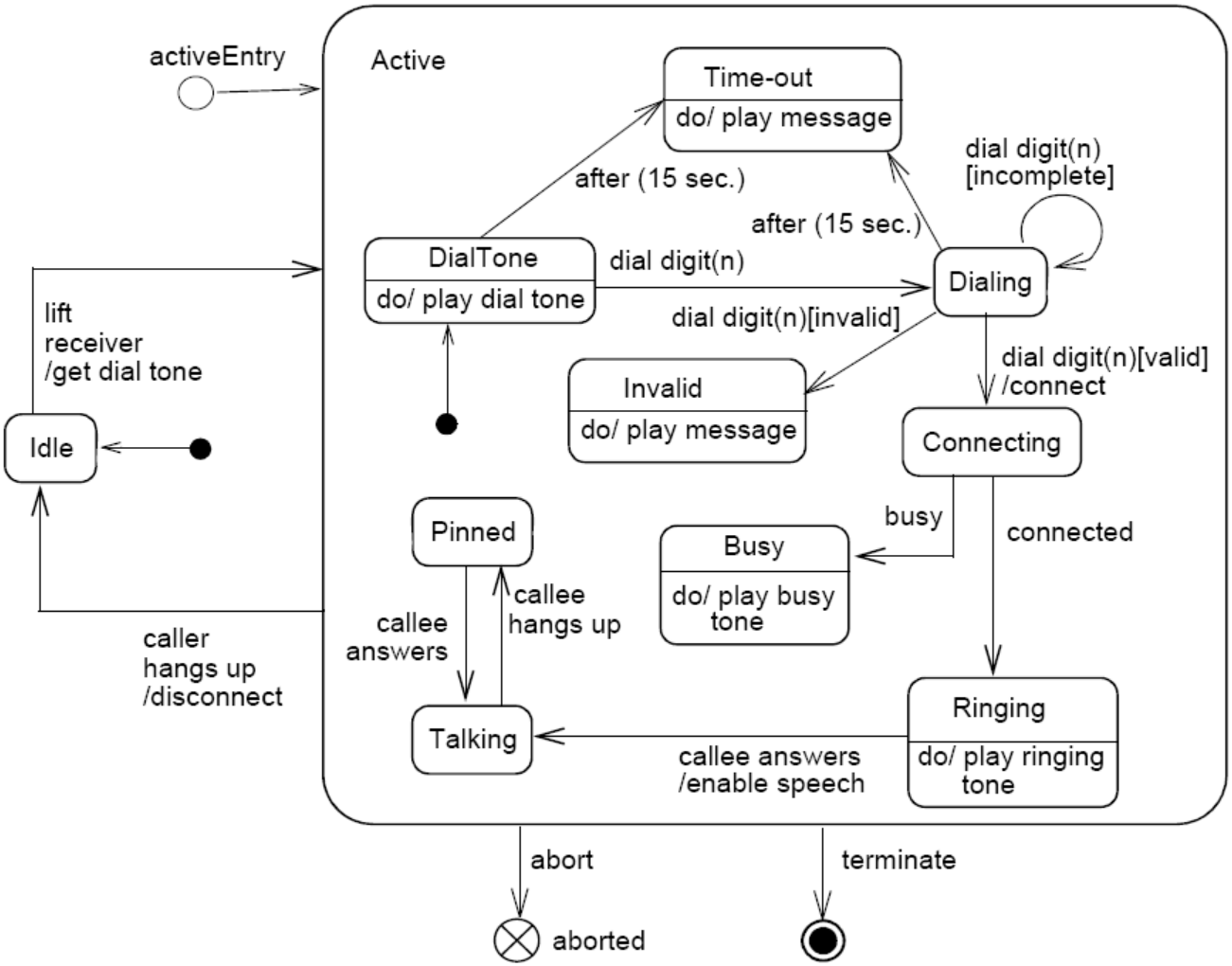
UML State Machine Diagram

Example: see diagram right

Objects stay in a certain state until an event triggers a transition.

The most important elements of a UML state chart are

- States
- Transitions
- Events



States

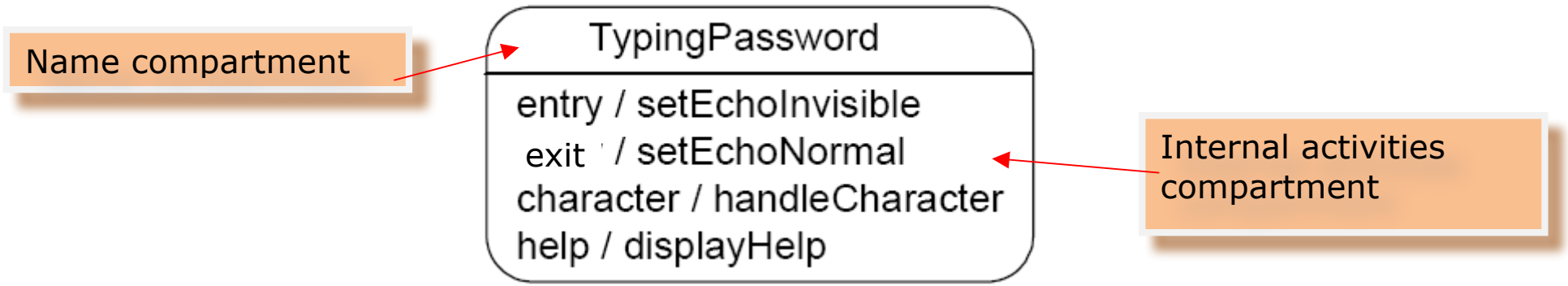
A state models a situation during which some (usually implicit) invariant condition holds.

The following kinds of states are distinguished:

- Simple state
- Composite state
- Submachine state

A composite state is either a simple composite state (with just one region) or an orthogonal state (with more than one region).

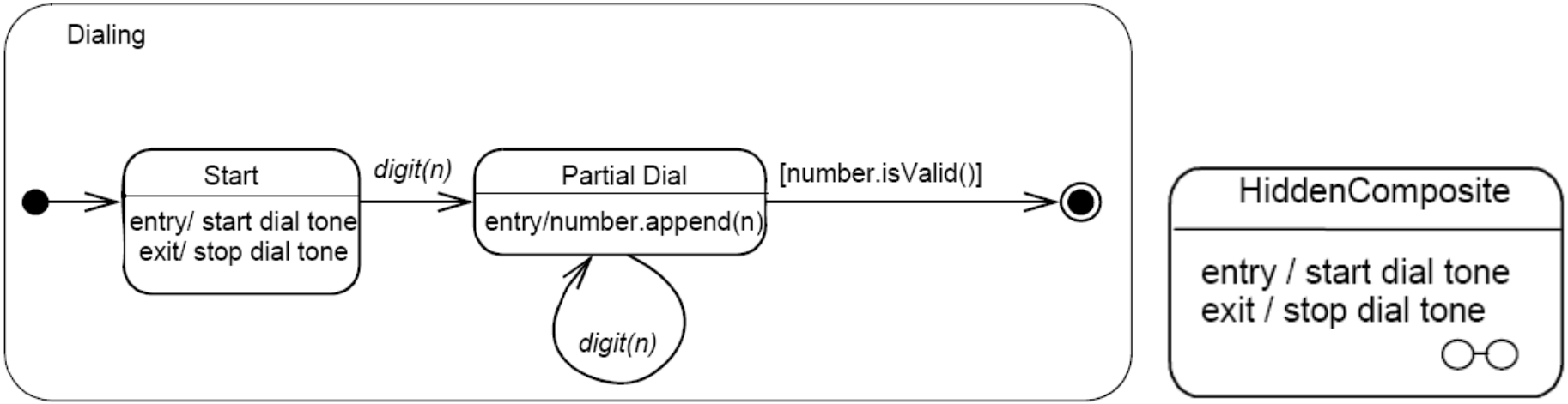
Simple state: A simple state is a state that does not have any substates.



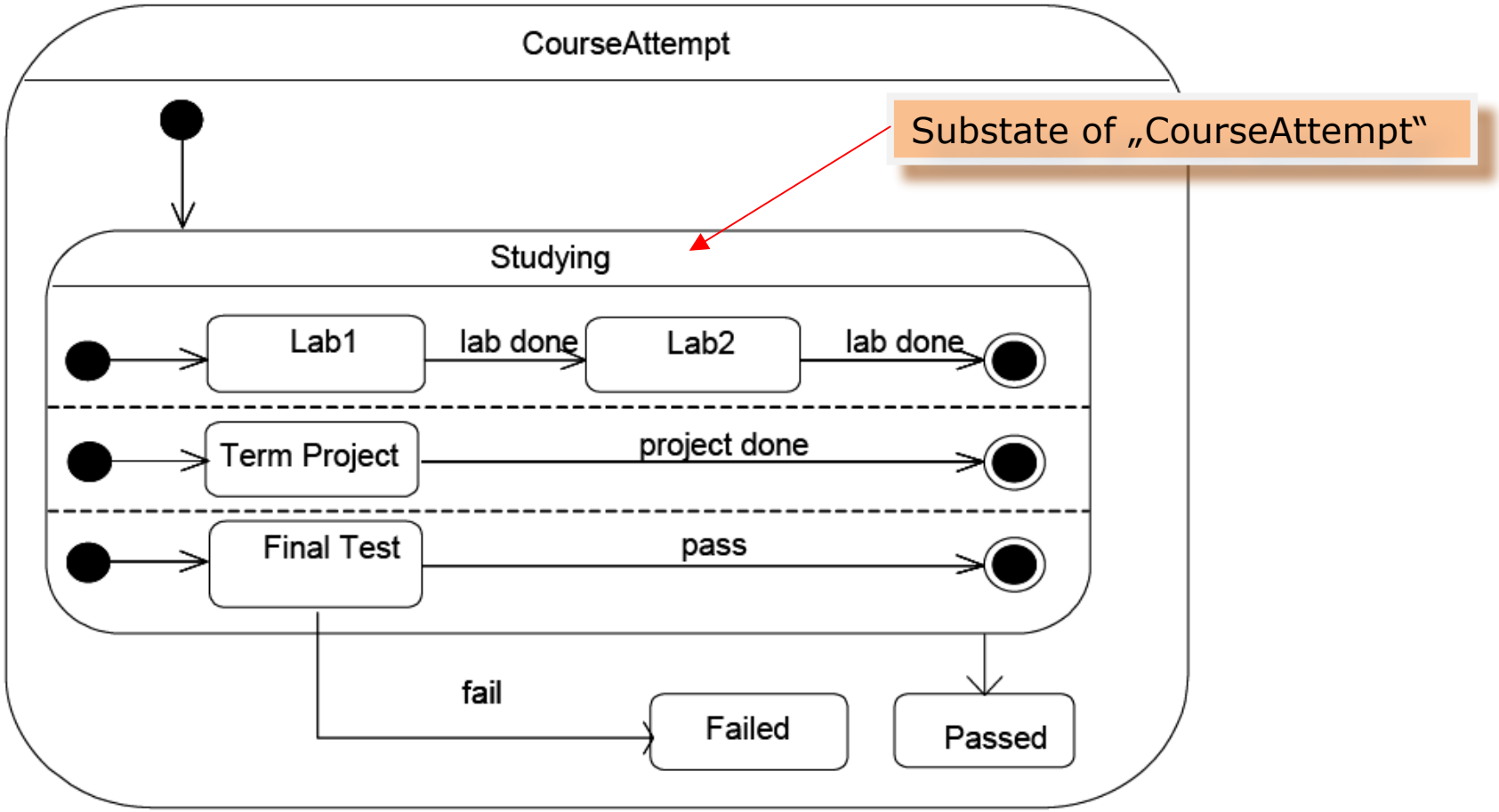
For the internal activities compartment, a number of labels are reserved for various special purposes and, therefore, cannot be used as event names:

- entry — This label identifies a behavior, specified by the corresponding expression, which is performed upon entry to the state (entry behavior).
- exit — This label identifies a behavior, specified by the corresponding expression, that is performed upon exit from the state (exit behavior).
- do — This label identifies an ongoing behavior ("do activity") that is performed as long as the modeled element is in the state or until the computation specified by the expression is completed (the latter may result in a completion event being generated).

Composite state: A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions. Example:



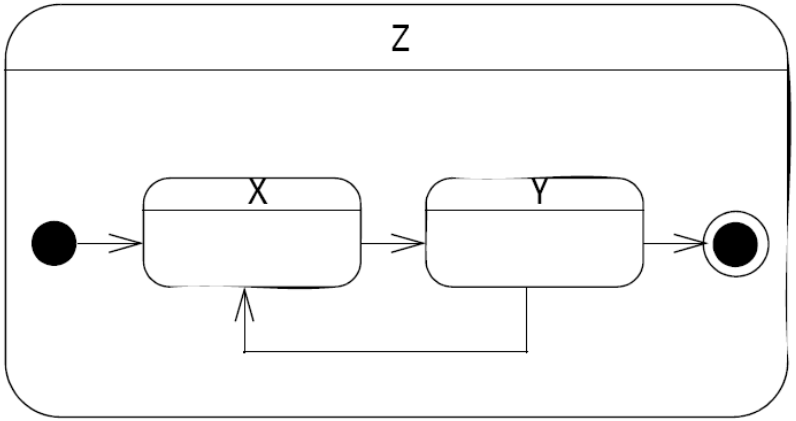
Another example:



“CourseAttempt” is an example of a composite state with a single region.
“Studying” is a composite state that contains three regions.

Any state enclosed within a region of a composite state is called a substate of that composite state.

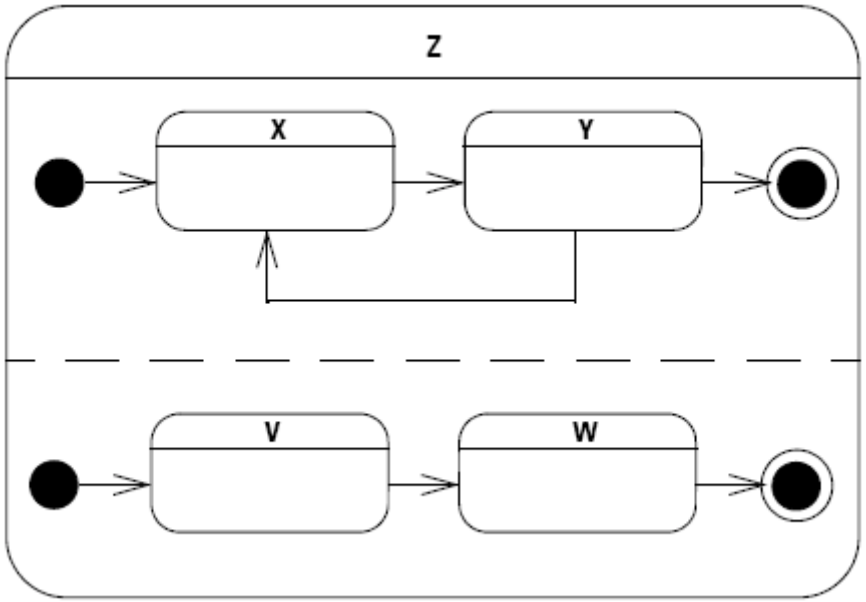
Another example for a composite state with a single region and two substates:



Object is in exactly one substate of state "Z" at any time, here either "X" or "Y".
When Object is in substate "X" or "Y", it is also in state "Z", and all other states "Z" may be a substate of.

Such a composition is also called an "OR composition".

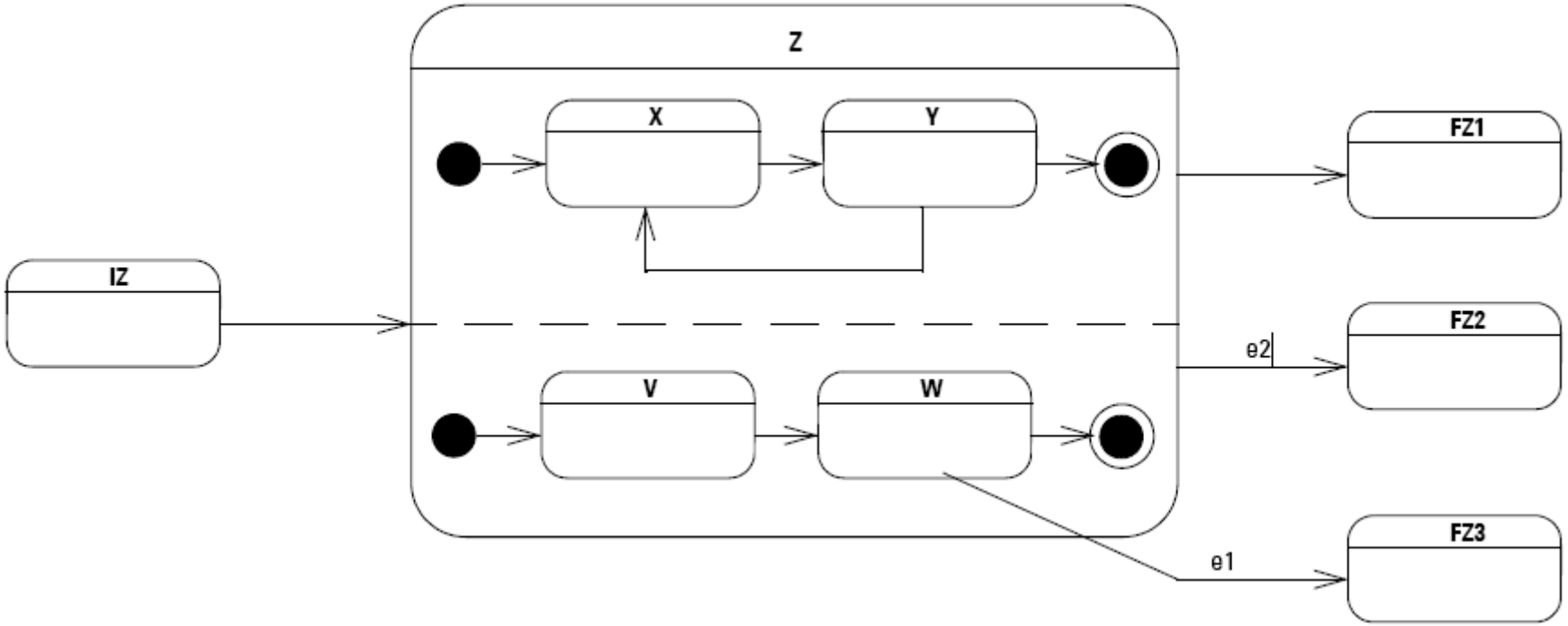
Another example for a composite state with orthogonal regions:



Object can be at the same time in “X” or “Y” AND “V” or “W”.

- “Z” will be left via an unmarked transition when all substates have reached their final pseudo state (implicit synchronization).
- “Z” will be also left if there is an outgoing transition from a substate out of the super-state, regardless of any other substates
- „Z“ will be left if there is a transition away from the superstate that is marked with an event, and that event triggers.

Example for exit behavior with orthogonal regions:



State „Z" will be left if

- "Y" and "W" have been left (subsequent state "FZ1")
- or in state „W" event „e1" triggers (subsequent state „FZ3")
- or in any substate event "e2" triggers (subsequent state „FZ2")

Excerpt from UML specification: “Each region of a composite state may have an initial pseudostate and a final state. A transition to the enclosing state represents a transition to the initial pseudostate in each region.
A newly-created object takes its topmost default transitions, originating from the topmost initial pseudostates of each region.
A transition to a final state represents the completion of behavior in the enclosing region. Completion of behavior in all orthogonal regions represents completion of behavior by the enclosing state and triggers a completion event on the enclosing state. Completion of the topmost regions of an object corresponds to its termination.
An entry pseudostate is used to join an external transition terminating on that entry point to an internal transition emanating from that entry point.
An exit pseudostate is used to join an internal transition terminating on that exit point to an external transition emanating from that exit point.
The main purpose of such entry and exit points is to execute the state entry and exit actions respectively in between the actions that are associated with the joined transitions.”

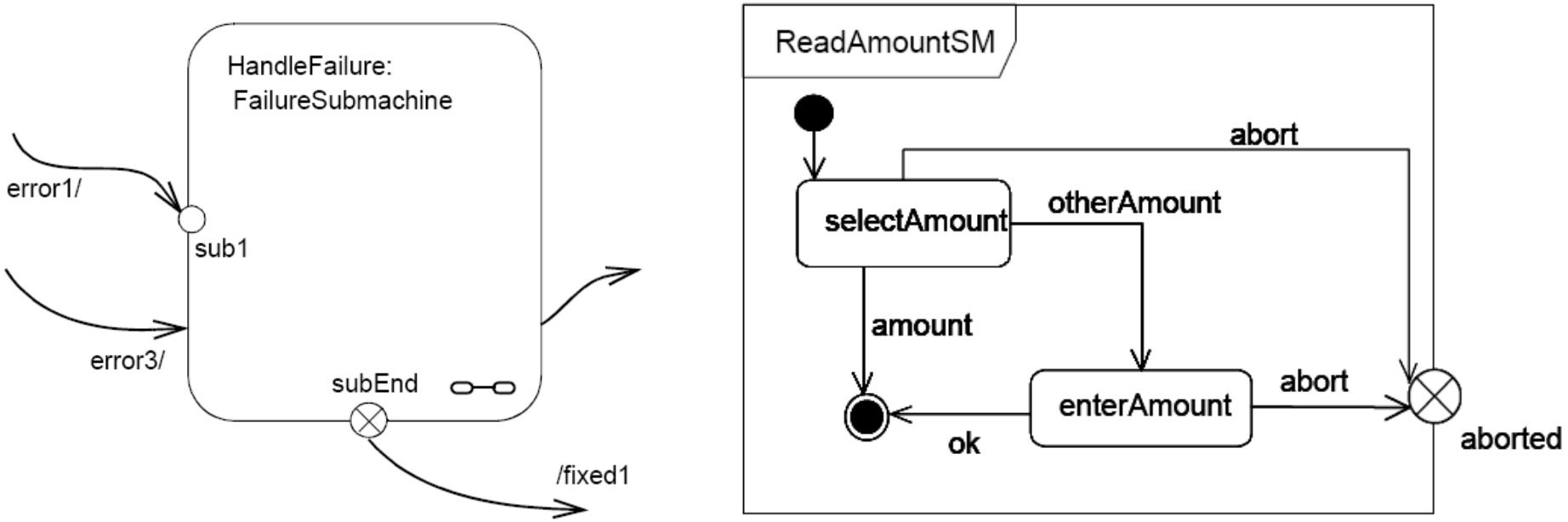
Submachine state: A submachine state specifies the insertion of the specification of a submachine state machine. Submachine state is a decomposition mechanism that allows factoring of common behaviors and their reuse.

The state machine that contains the submachine state is called the containing state machine.

The same state machine may be a submachine more than once in the context of a single containing state machine. A submachine state is semantically equivalent to a composite state.

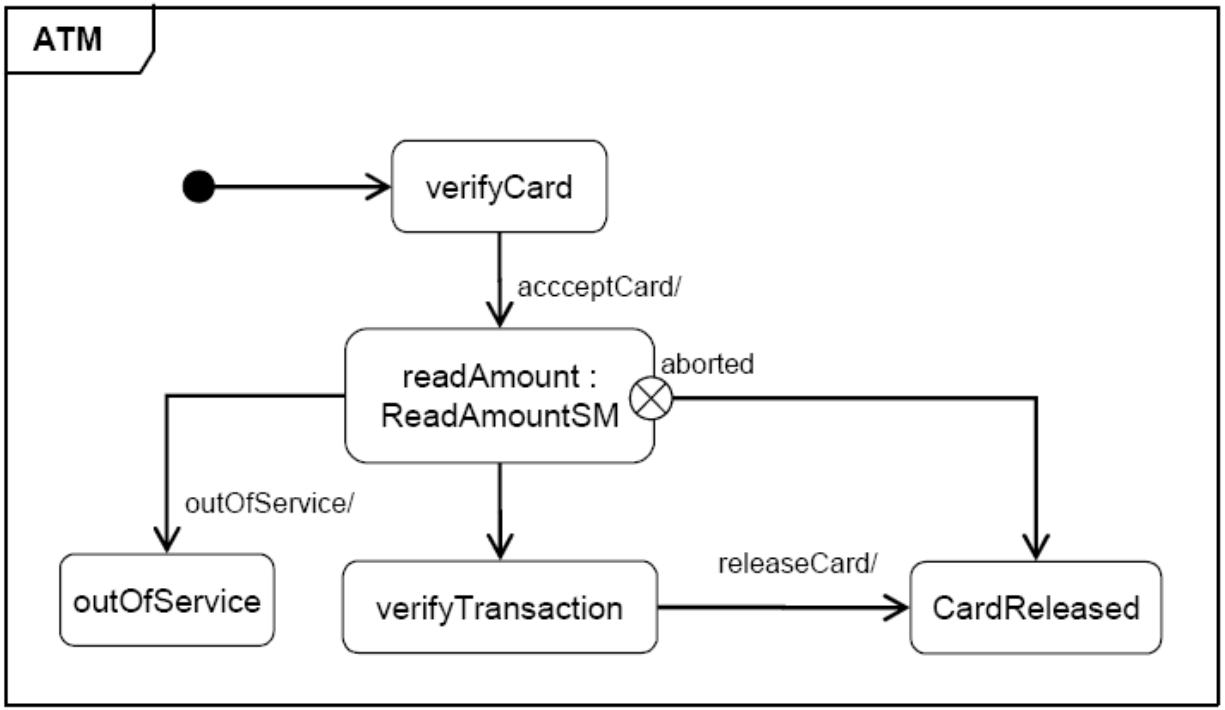
Entering and leaving this composite state is, in contrast to an ordinary composite state, via entry and exit points. The regions of the submachine state machine are the regions of the composite state. The entry, exit, and behavior actions and internal transitions are defined as part of the state.

Transitions in the containing state machine can have entry/exit points of the inserted state machine as targets/sources.



"ReadAmountSM" is a substate machine definition, with an explicit exit point ("aborted") and a standard entry point (initial pseudo state).

Usage of this substate machine see next slide:



Pseudostates

There are a number of special states or pseudostates.

The initial state

- Is represented by a small black circle
- Must not have any incoming transitions and exactly one outgoing transition
- The outgoing transition is immediately taken when system starts (power, reset)
- The outgoing transition must not have any guard or event associated with it

The final state

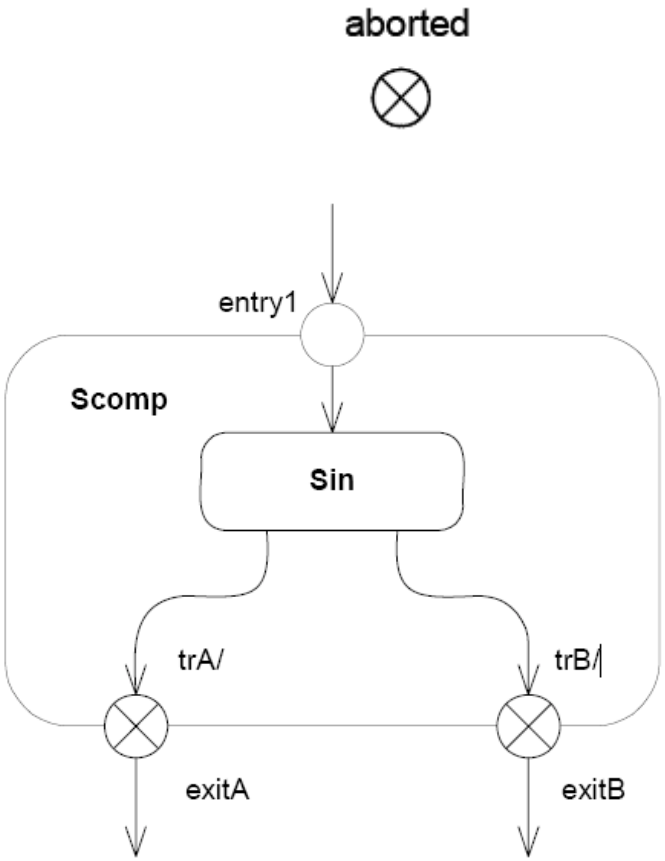
- Is represented by a small black circle surrounded by a ring
- Must not have any outgoing transitions
- If the state diagram is part of a superstate, the object will stay in the final state until an event takes the object away from that superstate

The entry point

- Used as entry point of a state machine or composite state
- In each region of the state machine or composite state it has a single transition to a vertex within the same region
- A state machine or composite state may have more than one entry point

The exit point

- Used as exit point of a state machine or composite state.
- Entering an exit point within a state machine or composite state implies exiting from this particular machine or state via the the transition that connects to this point in the composite state or state machine of the enclosing state.



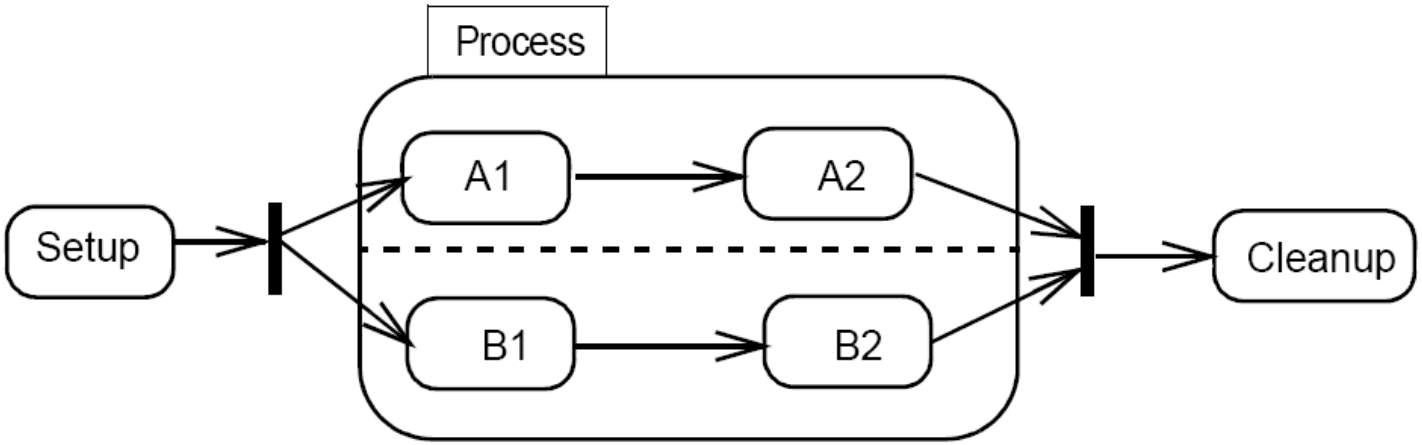
The join vertex

- Serves to merge several transitions form different orthogonal regions.
- Transitions entering join vertex cannot have guards or triggers.

The fork vertex

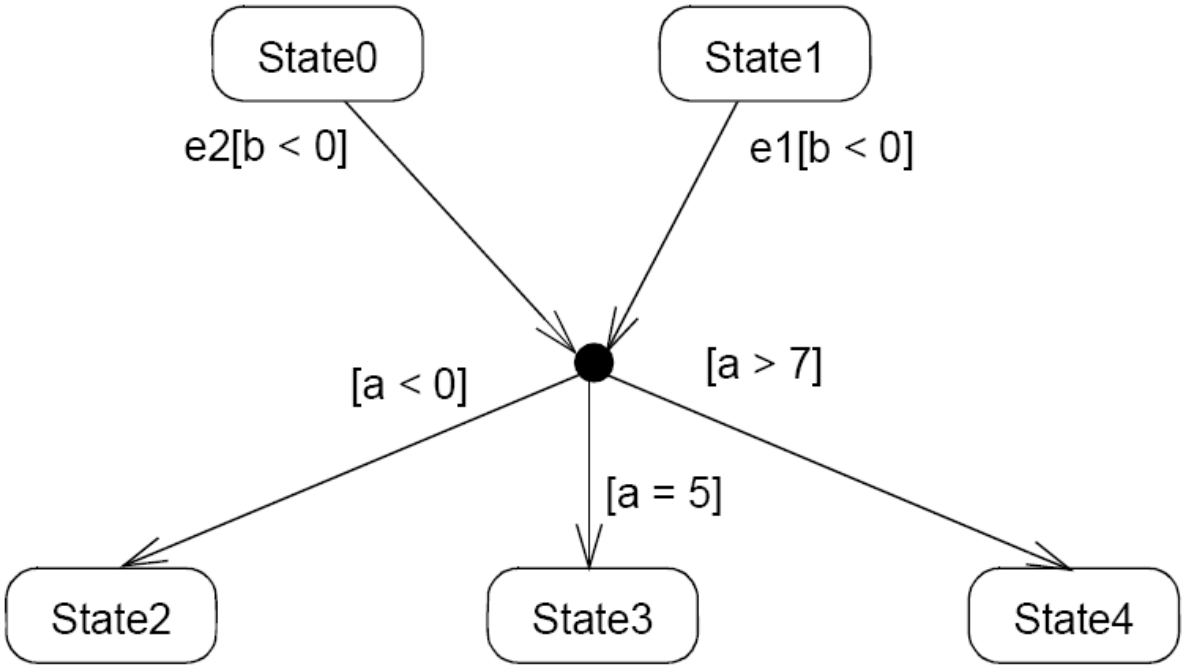
- Serves to split an incoming transition into two or more transitions terminating in different orthogonal regions of a composite state.
- The segments outgoing from a fork vertex must not have any guards or triggers.

Example for fork and join vertex:



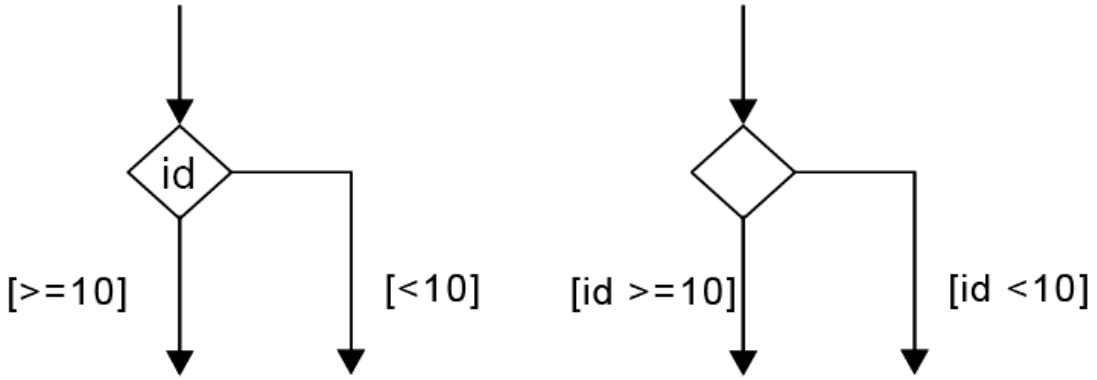
The junction vertex

- Are used to chain together multiple transitions.
- Used to construct compound transition paths between states.
- Can be used to converge multiple incoming transitions into a single outgoing transition (merge).
- Can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions.
- A predefined guard "else" may be defined for at most one outgoing transition.



The choice pseudostate

- When reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions.
- There should be one outgoing transition with the predefined "else" guard for every choice vertex.
- If more than one guard evaluates to true, an arbitrary one is selected.



The terminate node

- Entering a terminate pseudostate terminates execution of this state machine.
- The state machine does not exit any states nor does it perform any exit actions.
- Is equivalent to destroying the related object.

Transitions

A transition is a directed relationship between a source vertex and a target vertex.

The format of a transition is:

[<trigger>[‘;’ <trigger>]*[‘[‘<guard-constraint>’]’][‘/’ <behavior-expression>]]

Examples:

ButtonPressed() [CarIsMoving] / BrakeAction()

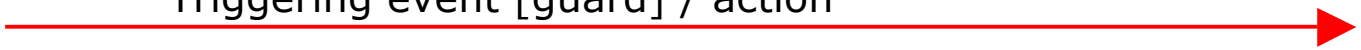
Right-mouse-down(location)[location in window]/object := pick-object(location); object.highlight()

A transition can be denoted by an arrow labeled with the trigger, guard, and action expressions.
The following rules pertain:

- A transition fires when the associated trigger occurs and the guard condition evaluates to true.
- A transition terminates any active actions in the source state.
- In case a transition does not have a trigger associated with it, the termination of the source state activities constitutes the trigger for this transition.

Format:

Triggering event [guard] / action

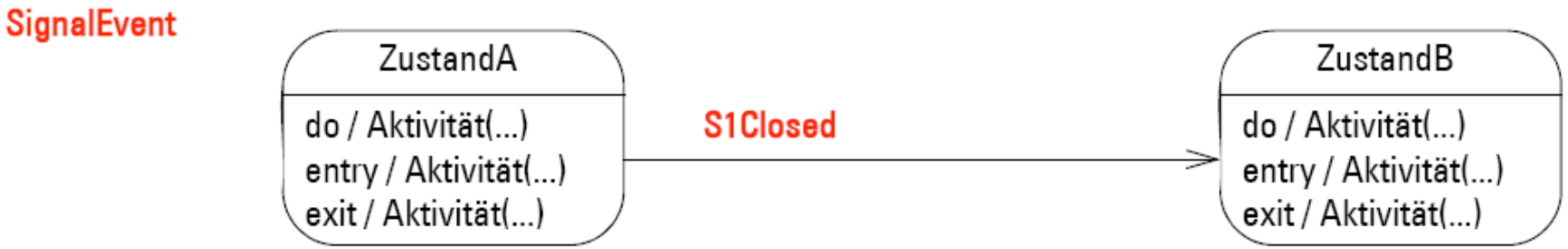


Triggering Events

In many cases, events trigger transitions. The triggering event can have arguments coming with it. There are a number of event types, some of which we shall mention here:

- SignalEvent
- CallEvent
- ChangeEvent
- TimeEvent

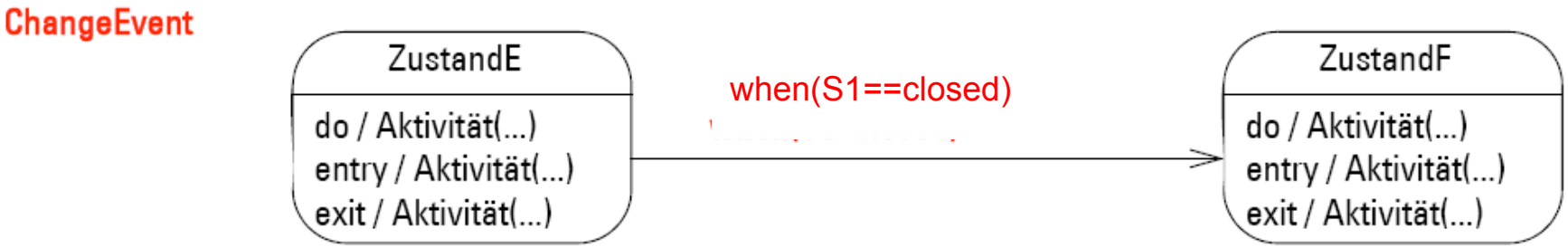
SignalEvent: a signal is received, i.e., generated by some system-external process. The event name identifies the signal. Parameters are possible, but unusual. A SignalEvent is typically realized by hardware or software interrupts.



CallEvent: a message is received, e.g., by means of a method or operation call. Event name designates the operation. Example: collidesWith(Date).

ChangeEvent: a certain condition is met.

- Instead of event name keyword "when" followed by condition.
- Event triggers when condition switches from false to true.
- If a transition does not fire because of a guard condition being false, the event condition has to first switch back to false before it can trigger again.
- ChangeEvent is permanently evaluated, the guard condition only after event has triggered.



Time Event: permits to define time related events.

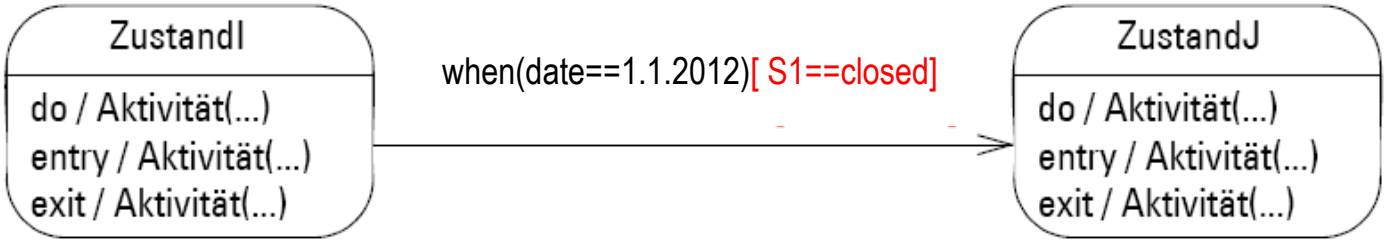
- Can be defined relative with keyword "after".
- Can be defined absolute with keyword "when" (not to confuse with ChangeEvent)

Example relative:

- after(3 sec since exit from state X)
- after (2 sec) which means two seconds after entry into the source state

Guards

In case an event is triggered, and the guard condition is false, the event is discarded (consumed).



Time Event: permits to define time related events.

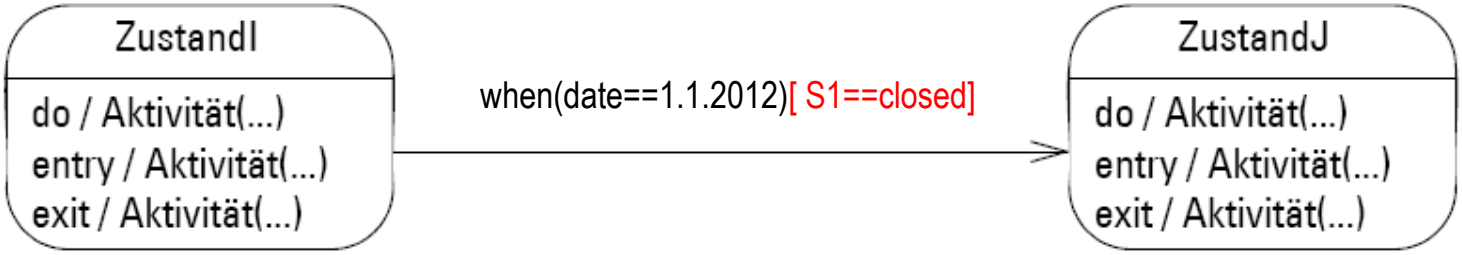
- Can be defined relative with keyword "after".
- Can be defined absolute with keyword "when" (not to confuse with ChangeEvent)

Example relative:

- after(3 sec since exit from state X)
- acter (2 sec) which means two seconds after entry into the source state

Guards

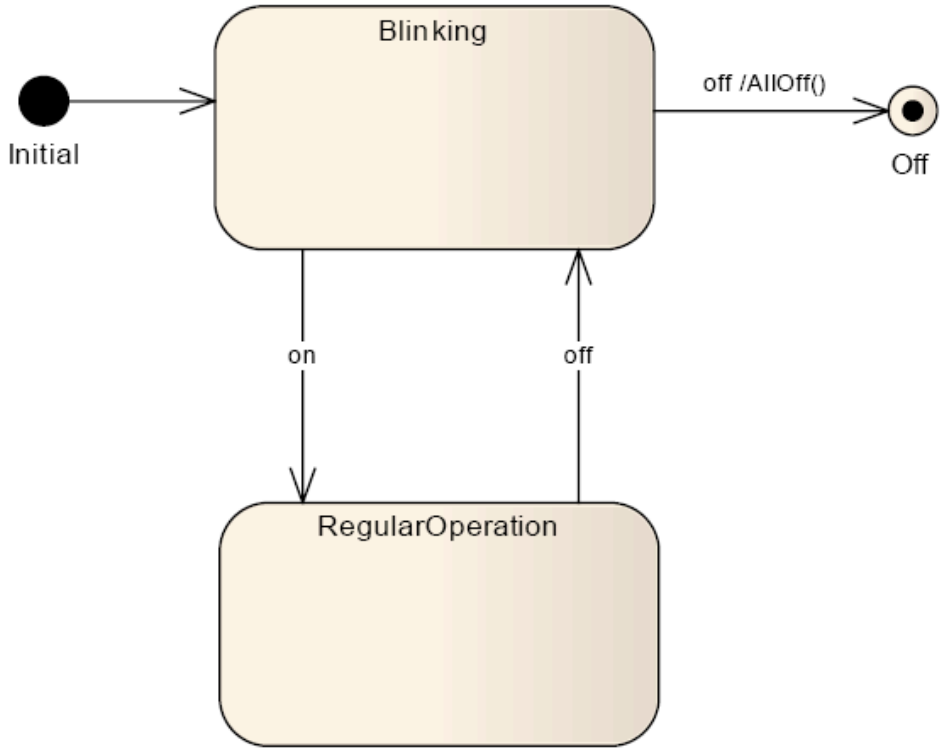
In case an event is triggered, and the guard condition is false, the event is discarded (consumed).



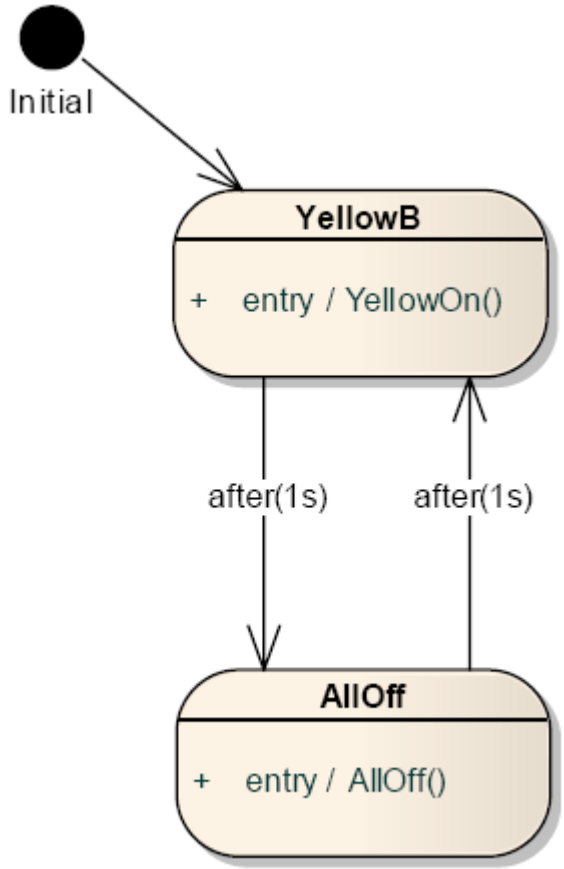
Example for a Simple State Machine: Traffic Light

Two modes of operation: blinking mode (Blinking), regular mode (RegularOperation), (Off).
Two user interface elements: button “on”, button “off”.

We model each mode of operation as a complex state.



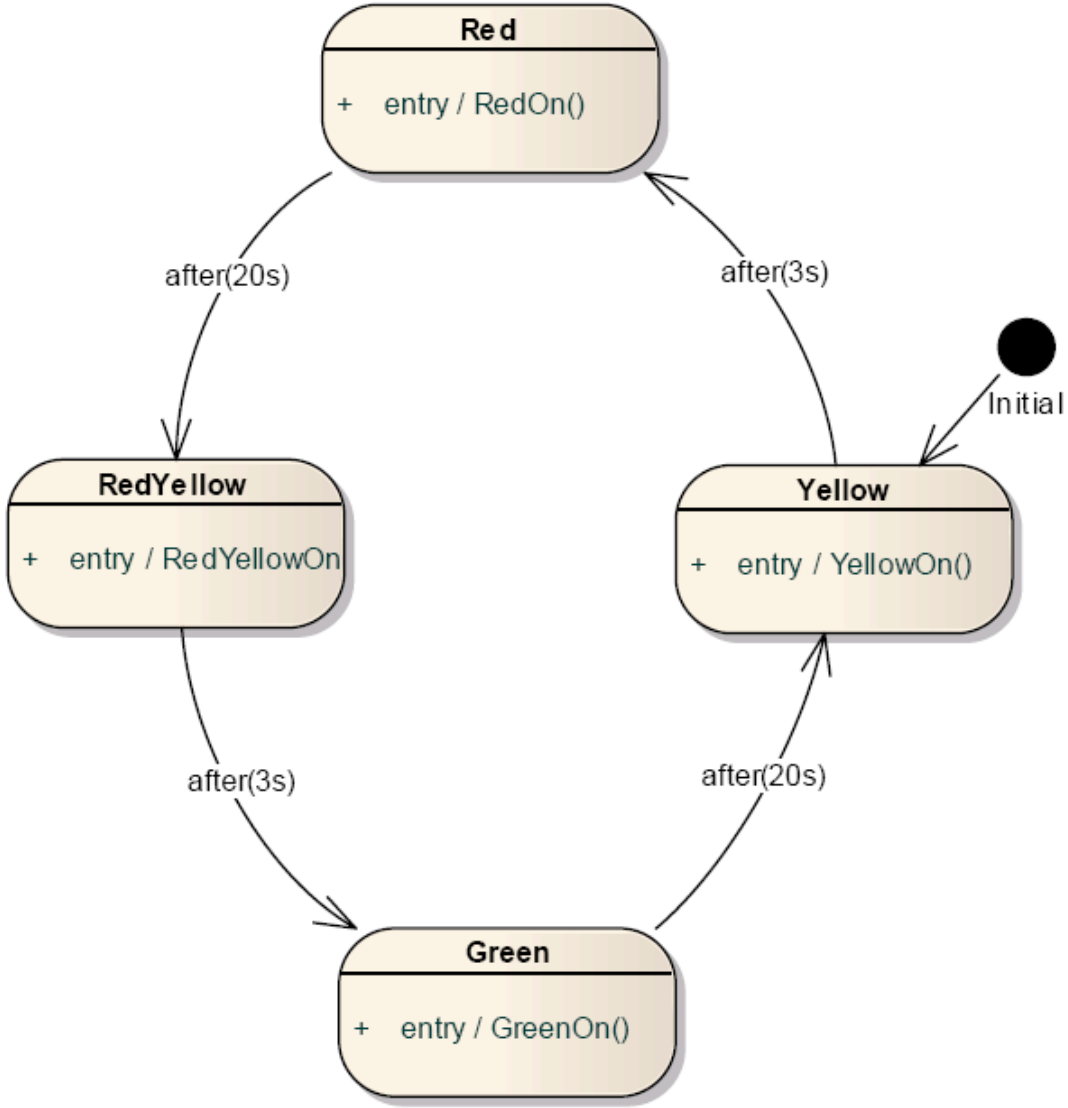
We model the blinking mode in more detail:



We have two simple states.

The exit condition is determined by the enclosing state.
There is an automatic switch between the two states every second.

Finally we model the regular mode:



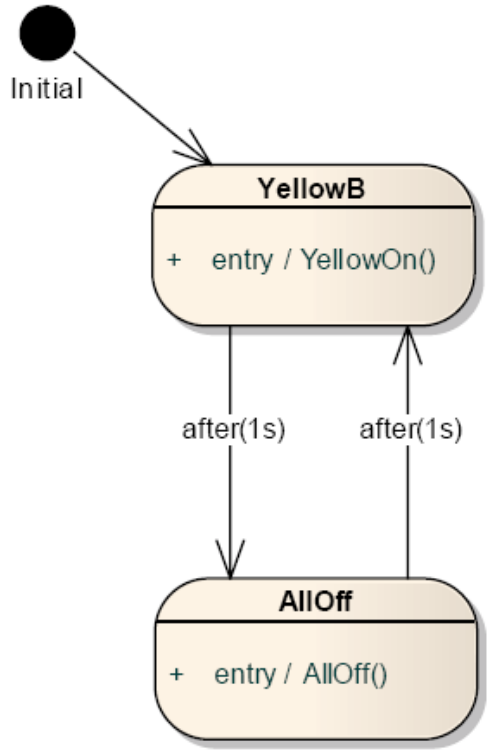
There are no exit points and no final state. This is because the superstate containing this substate defines entry and exit conditions.

Possible implementation: Switch-case approach see other lectures. The following is our approach, which is easily extendable and well to maintain:

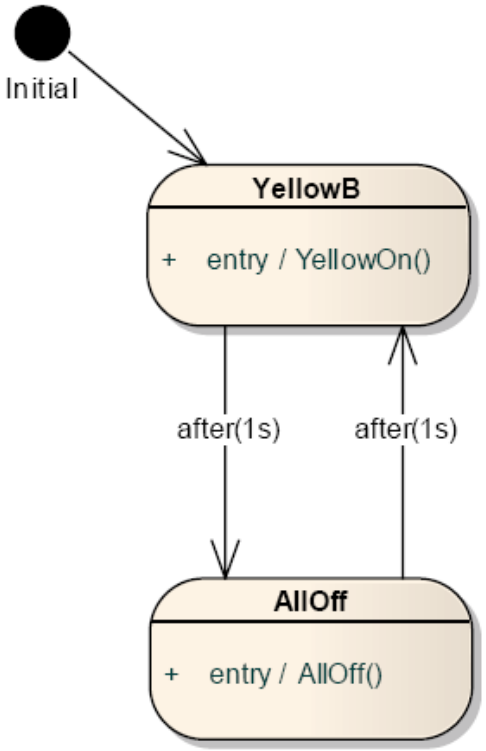
```
/* Real-Time Systems
(C) 2010 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, September 2010
*/
/* The states, once as enum, and in same order as function pointers */
typedef enum {RED=0, REDYELLOW, YELLOW, YELLOWB, GREEN, ALLOFF, OFF};
void (* stateTable[])()={ Red, RedYellow, Yellow, YellowB, Green, AllOff, Off };

unsigned char currentState;
unsigned char lastState;

void main( void ) {
   initHardware();
   initStateMachine();

   /* This is the heart of the state machine */
   for {
      stateTable[ currentState ]();
   }
}
```

Example for state function for state “YellowB” in blinking mode:

```
/* State YELLOWB */
void YellowB() {
/* three events are possible here:
- eventOn
- eventOff
- after(1s)
*/
   if (currentState != lastState) {
      resetTimer();
      lastState = currentState;
      transitionActivity = NONE;
      YellowOn(); /* onEntry activity */
   }
   if (eventOn()) { /* reaction on event "on" */
   /* here we could place a guard condition */
   /* here we could prepare for a transition activity */
   /* transitionActivity = YELLOWBTOYELLOW; */
      currentState=YELLOW; /* next state */
   }
   else if (eventOff()) {
   /* reaction on event "off" */
   /* transitionActivity = YELLOWBTOYELLOW; */
      currentState = OFF; /* next state */
   }
   else if (getTimerValue() >= 1) {/* reaction of event "after(1s)" */
```



```
      currentState=ALLOFF; /* next state */
   }
   if (currentState != lastState) {
   /* Here we could place the onExit activity */
   /* Here we execute the transition activities, if any */
      switch (transitionActivity) {
         case NONE: break;
         case YELLOWB2YELLOW: break;
         default: break;
      }
   }
}
void AllOff() {
   if (currentState != lastState) {
      lastState = currentState; /* eat up state transition */
      setAllOff(); /* onEntry activity */
      resetTimer();
   }
   if(getTimerValue() >= 1) {
      currentState=YELLOWB; /* set next state */
   }
   else if (eventOn()) {
      currentState=YELLOW; /* set next state */
   }
   else if (eventOff()) {
      currentState=OFF; /* set next state */
   }
}
```



4.2 UML State Machines

4.3 UML Activity Diagrams

Activity modeling emphasizes the sequence and conditions for coordinating low-level behavior.

Activities model the control flow and data flow between actions of an activity.

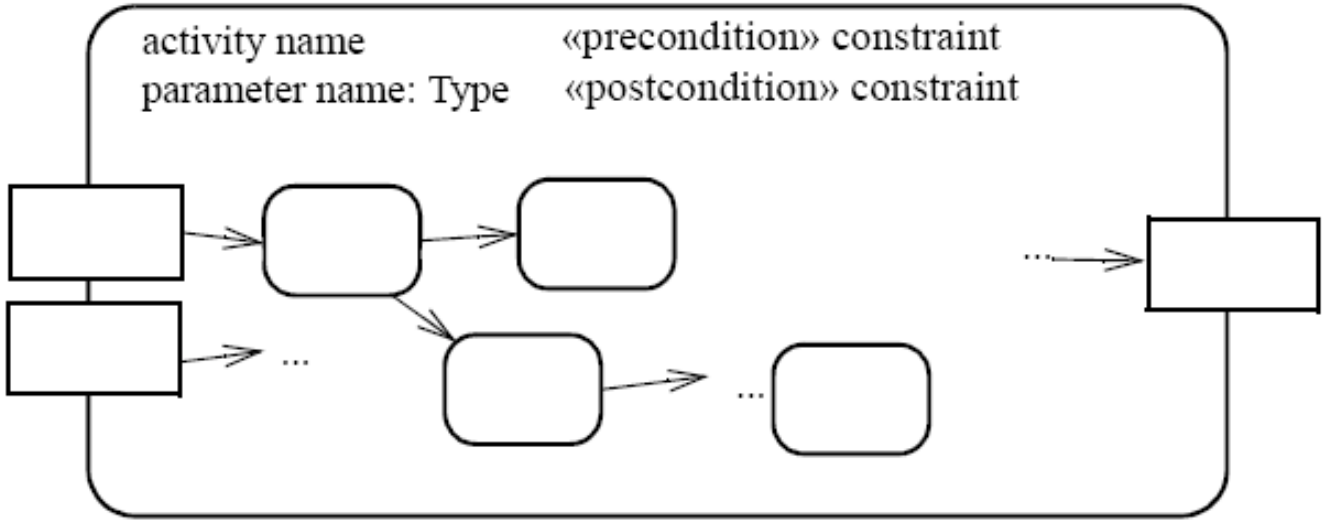
An action represents a single step within an activity.

Activity diagrams take the token concepts from Petri nets.

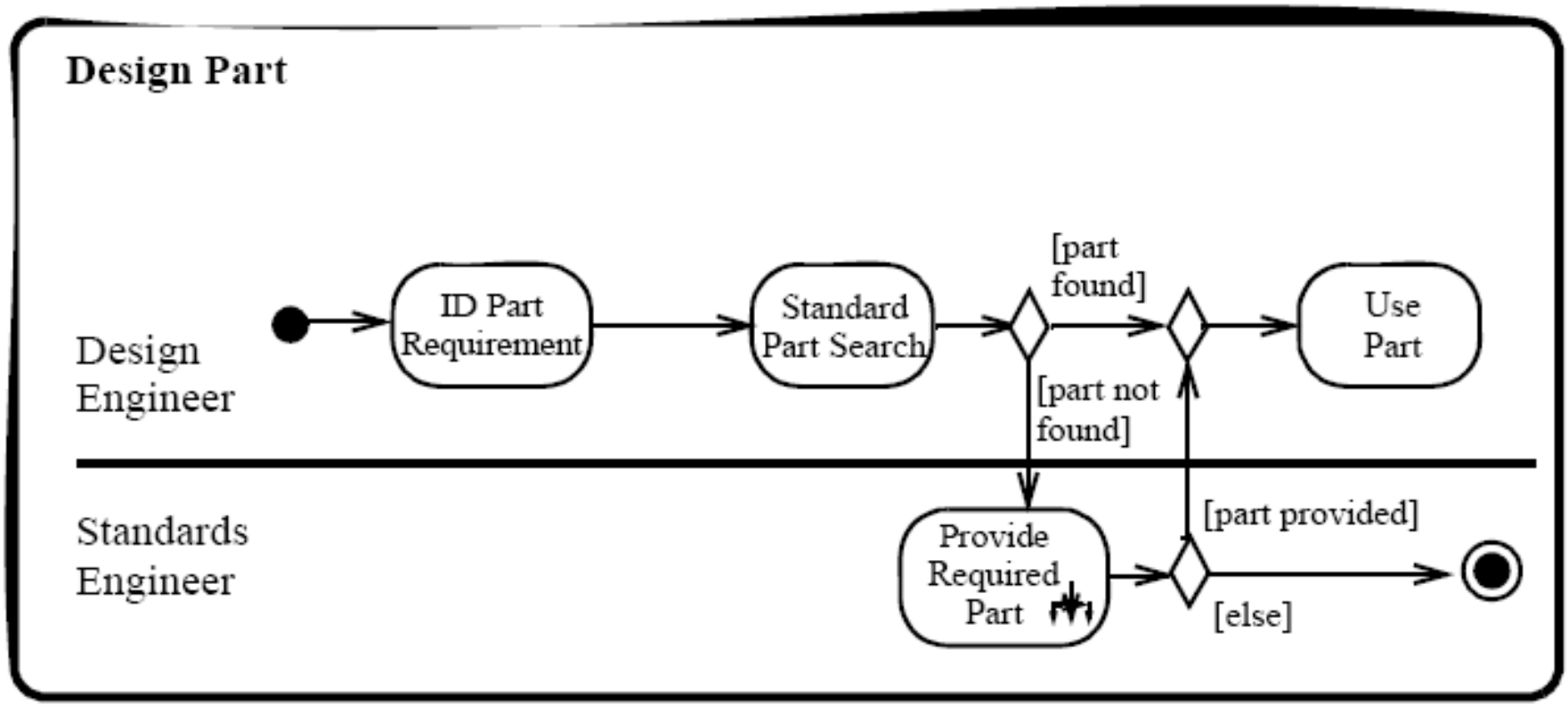
An activity is a directed graph, consisting of nodes and edges.

Nodes represent actions, control constructs, or data stores.

Edges define the transfer of control or data from a predecessor node to a successor node.



Example:

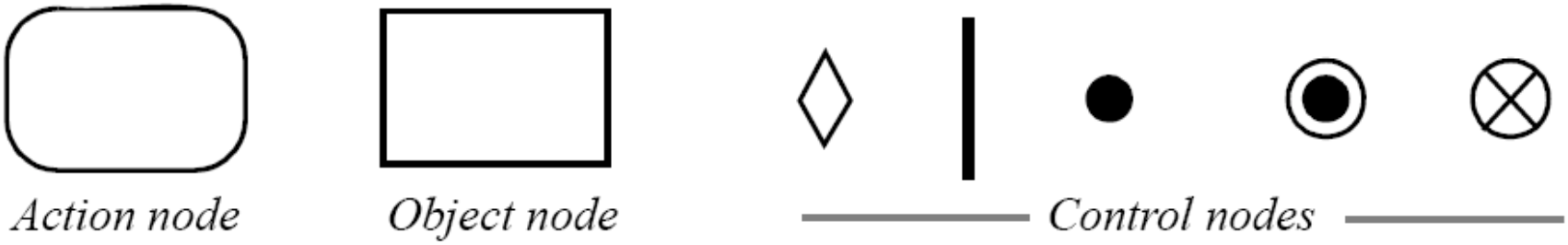

Design Part
Design Engineer
ID Part Requirement
Standard Part Search
[part found]
Use Part
[part not found]
Standards Engineer
Provide Required Part
[part provided]
[else]

Activity Nodes Overview

A node is just a point in an activity.

There are three types of nodes:

- Executable or action nodes (e.g., actions)
- Object nodes (e.g., a data store)
- Control nodes (e.g., a fork)



Action Nodes (Executable Nodes)

An action is the fundamental unit of behaviour specification.

An action takes a set of inputs and converts them into a set of outputs.

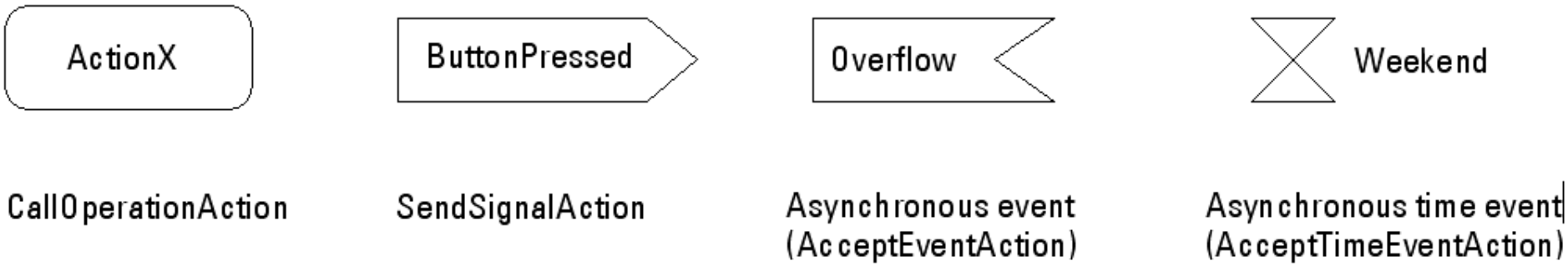
Actions themselves can contain any behavioural descriptions, such as state machines or activity models.

Actions are considered atomic, but can be interrupted.

There are a number of predefined specialized UML actions (56 as of UML 2.1).

Examples:

- CallOperationAction
- SendSignalAction
- SendObjectAction
- AcceptEventAction
- AcceptTimeEventAction

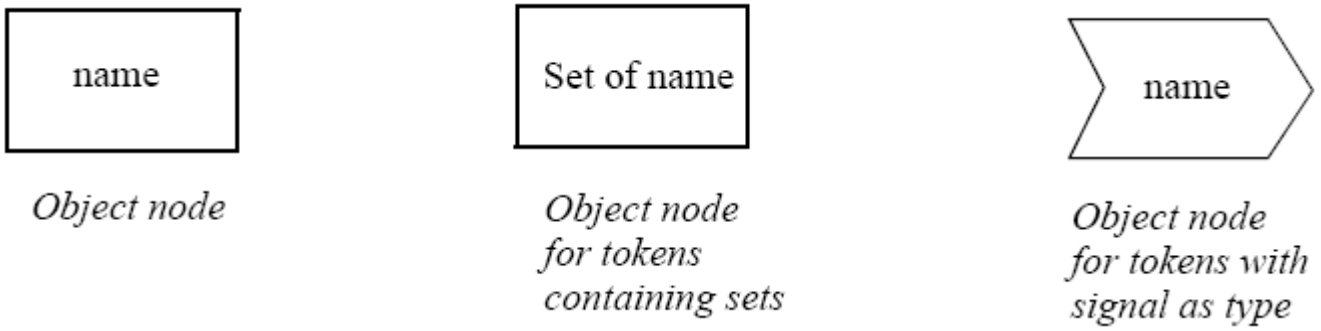


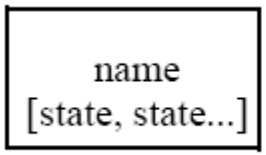
Object Nodes

Object nodes contain instances of a classifier, e.g. an object of a specific type.

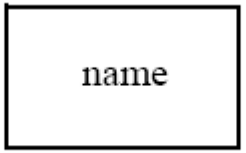
Values stored in object nodes are the result of an action and can be input for other actions.

Notation:



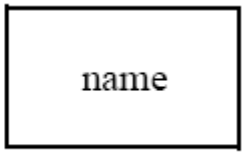


Object node for tokens containing objects in specific states



{upperBound = 2}

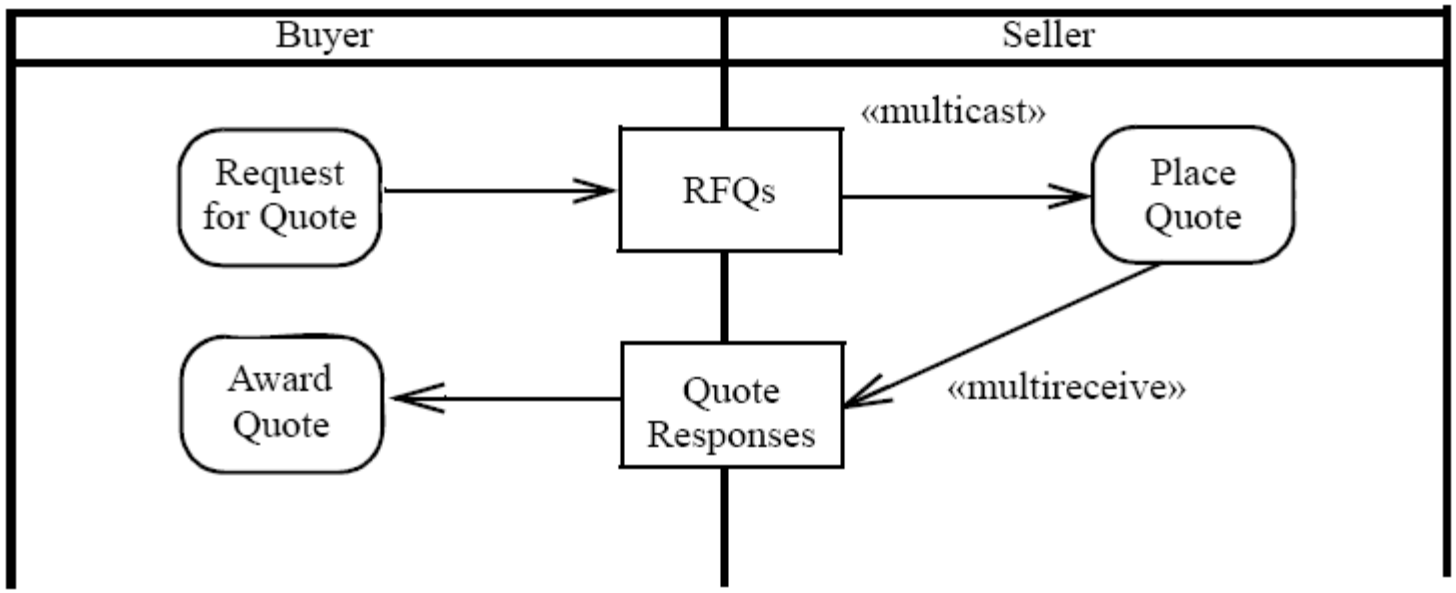
Object node with a limited upper bound



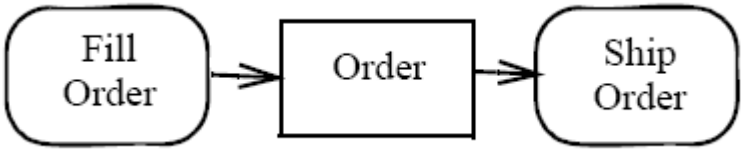
{ordering = LIFO}

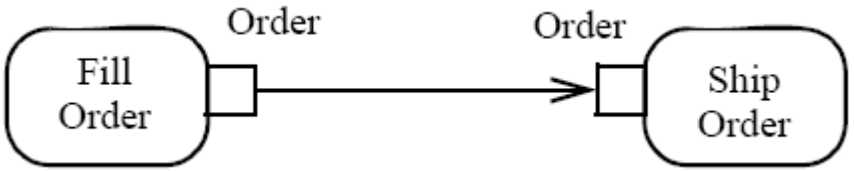
Object node with ordering other than FIFO

Example 1:



Example 2:

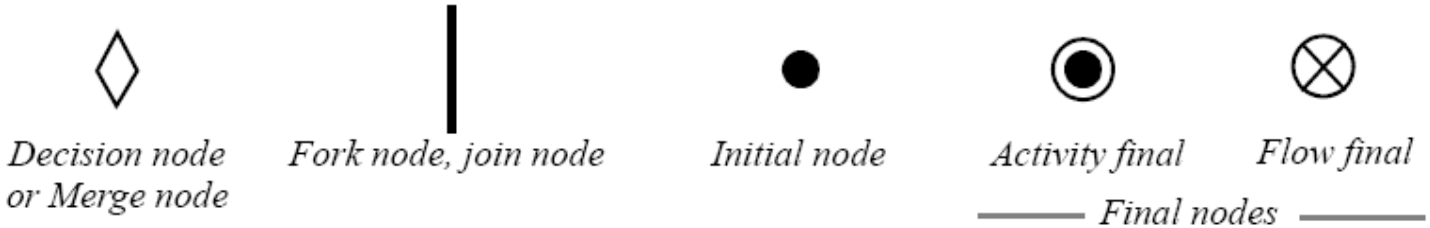

Fill Order
Order
Ship Order


Order
Order
Fill Order
Ship Order

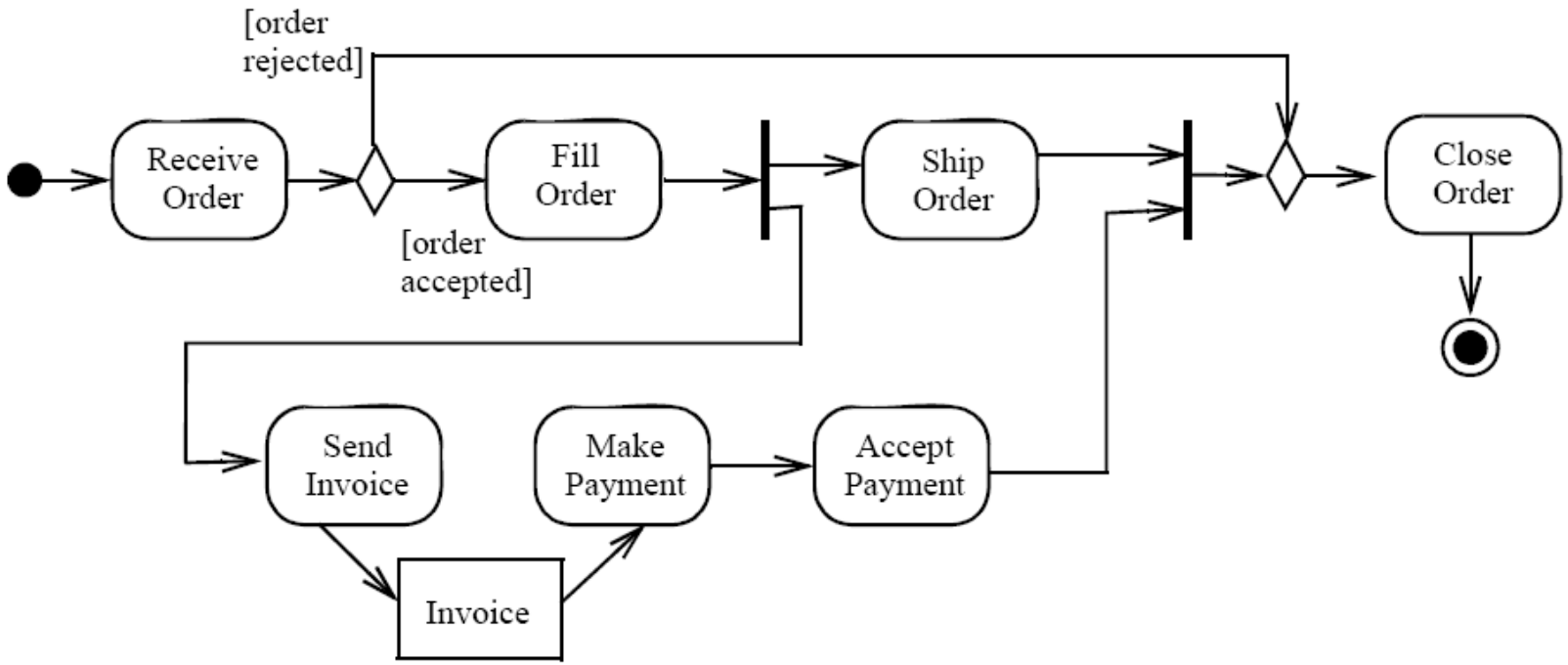
Control Nodes

Control nodes coordinate flow in an activity.

Notation:



Example:



Edges

Activity edges connect activity nodes.

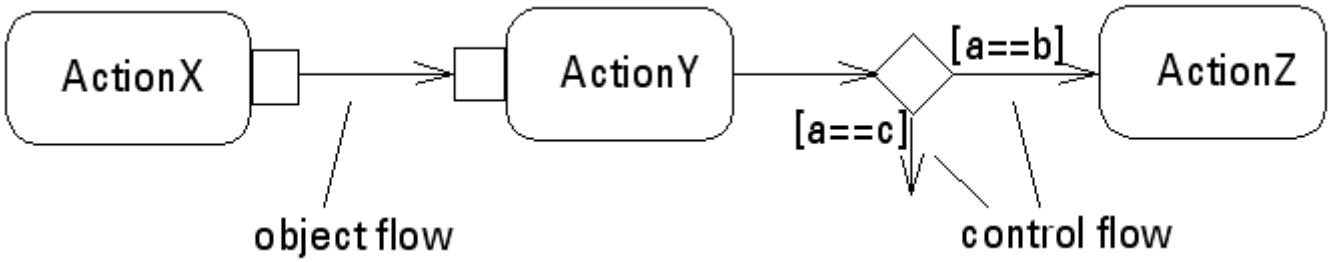
Activity edges define the data and control flow of an activity.

There are object flow edges (connecting object nodes) and .

Data is transported via object flow edges.

Edges can be named and can have guard conditions.

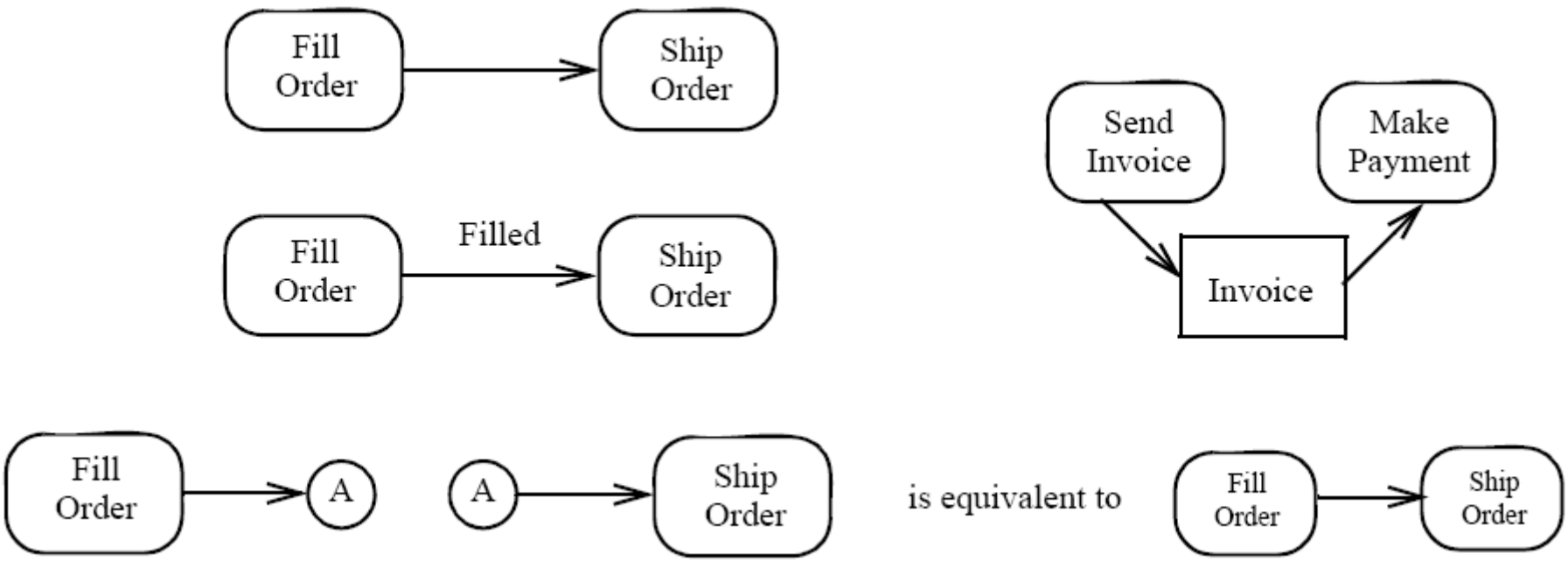
Example for notation:



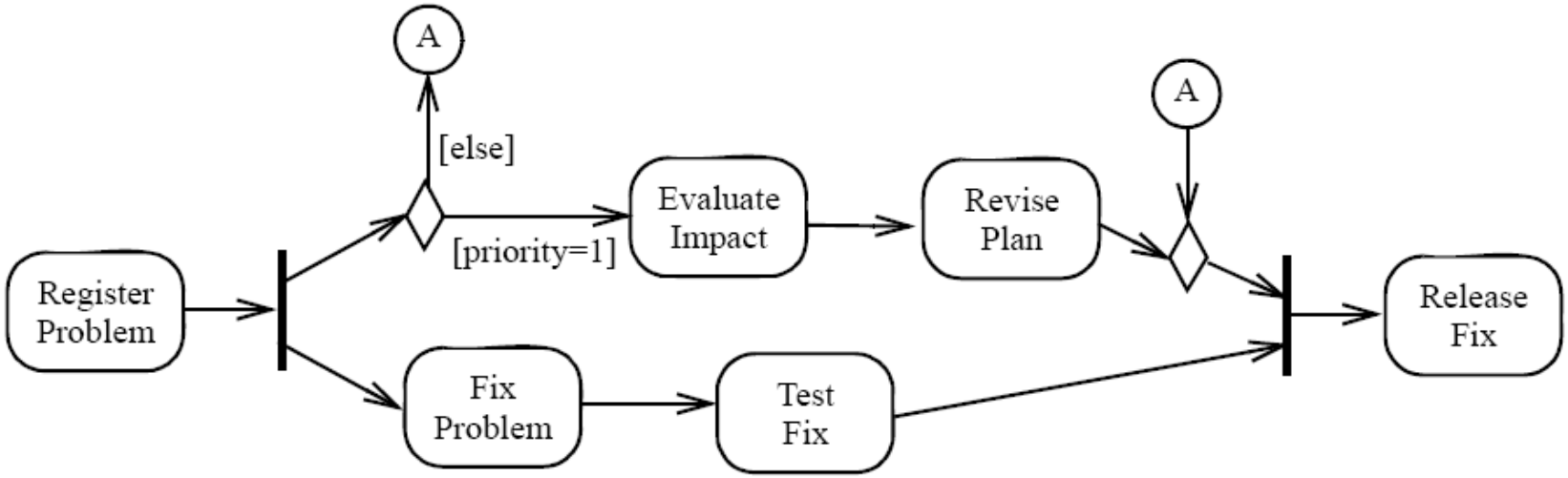
There are two guards at the control flow.

It is best to have one guard labeled "else" to avoid dead stops.

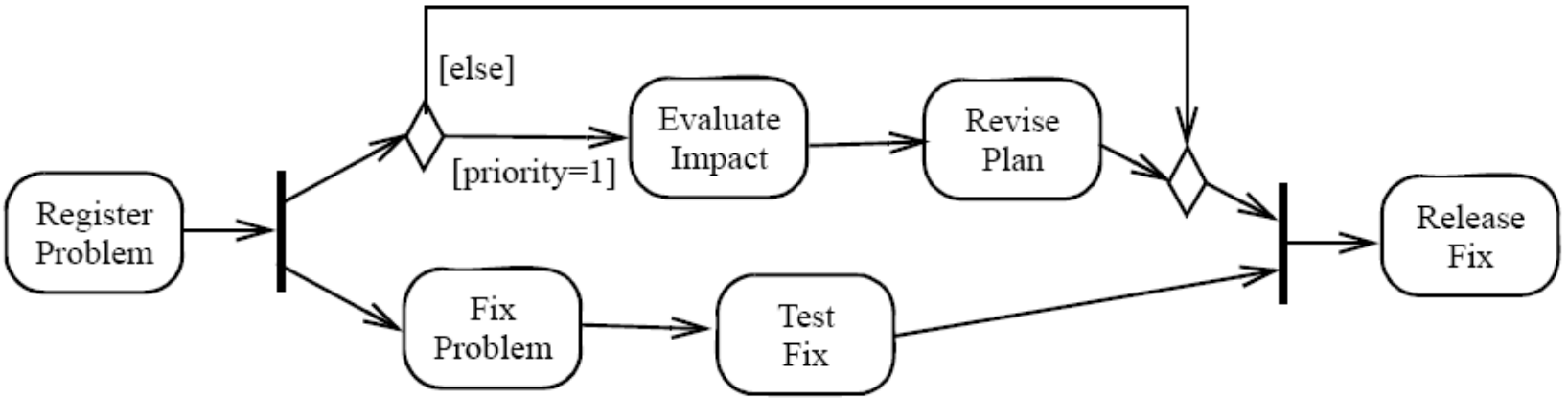
More examples with named and unnamed edges and connectors:



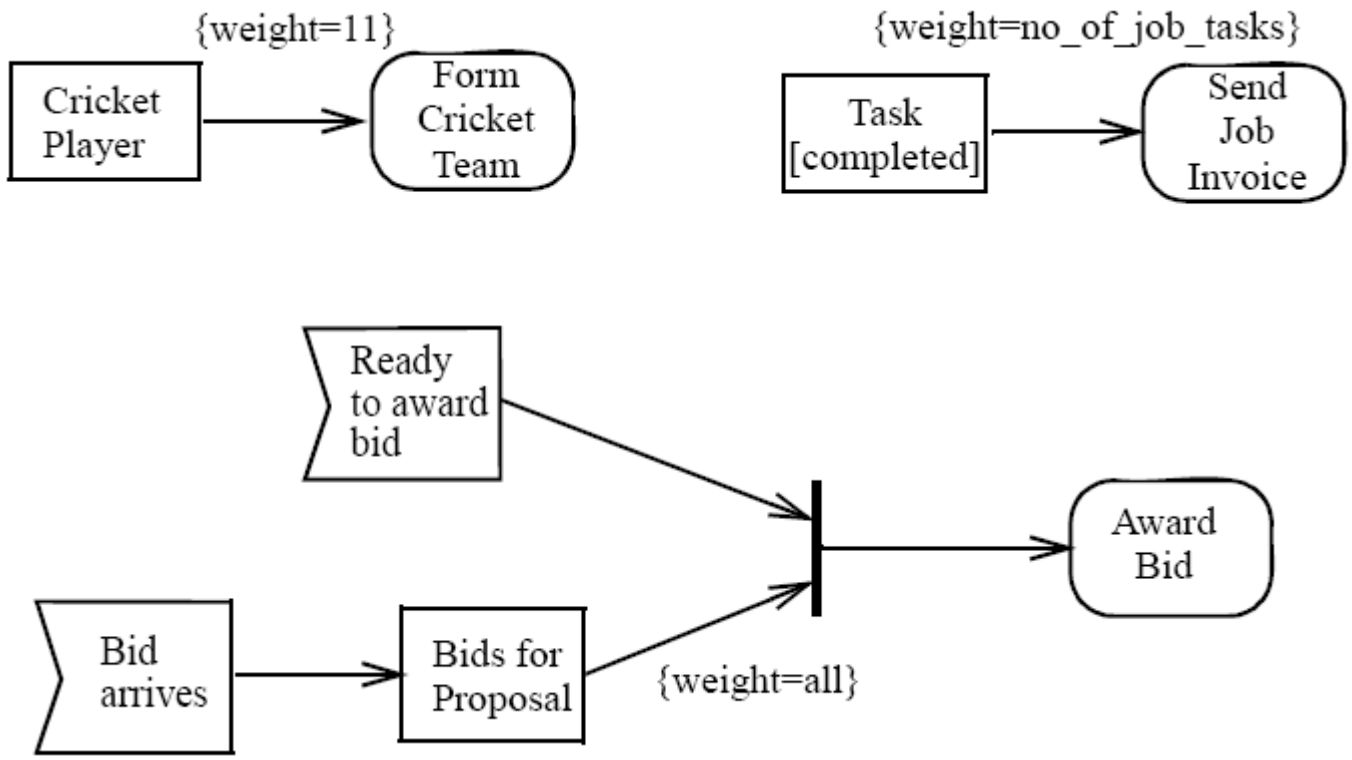
A connector example:



is equivalent to:



An example with weights:



A cricket team can only be formed if there are 11 players.

The bid can only be awarded if all bids have arrived and the signal "Ready to award bid" has arrived.

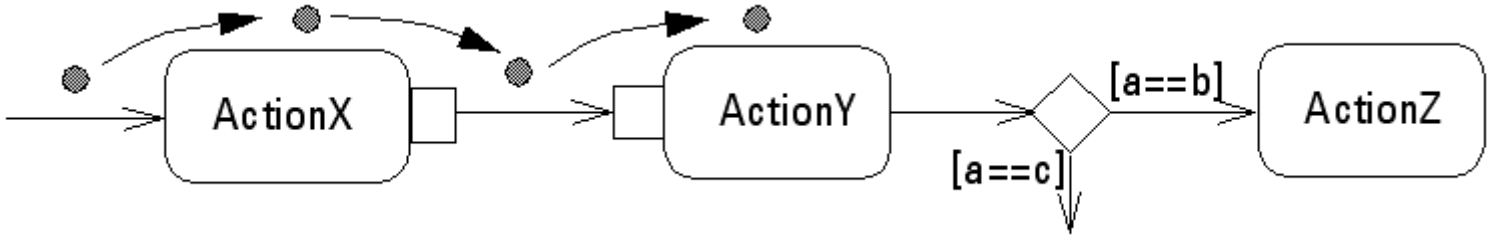
The job invoice can only be sent when all tasks have been completed.

The Token Concept

Data and control flow is conceptually transported via tokens.

A data or control token flows along an edge from a predecessor node to a successor node.

In case of concurrent activities, more than one token can exist in parallel.



The steps of executing an action with control and data flow are as follows:

1. An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join).
2. An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution.
3. An action continues executing until it has completed. Most actions operate only on their inputs. Some give access to a wider context, such as variables in the containing structured activity node.
4. When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork), and it terminates.

Partitions

Partitions group actions that have some properties in common.

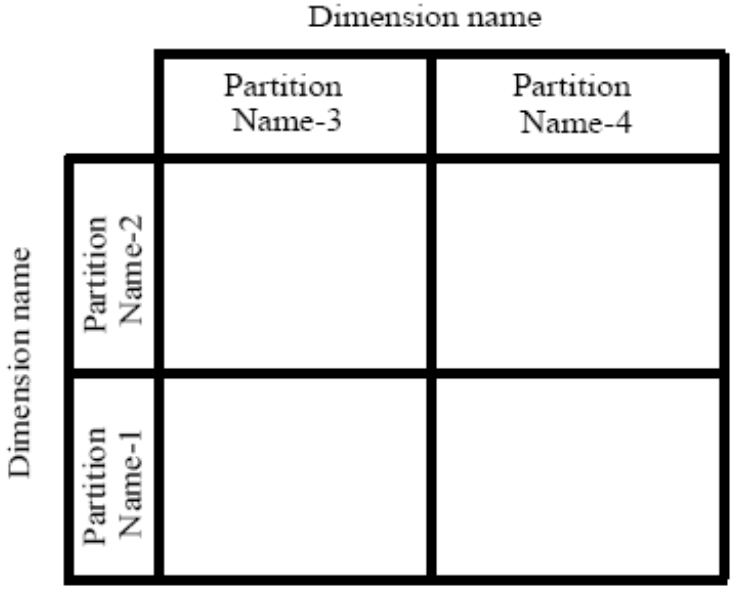
Partitions have no effect on the flow within an activity; they just represent a logical view.

Partitions are illustrated by "swim lanes".

Notations:

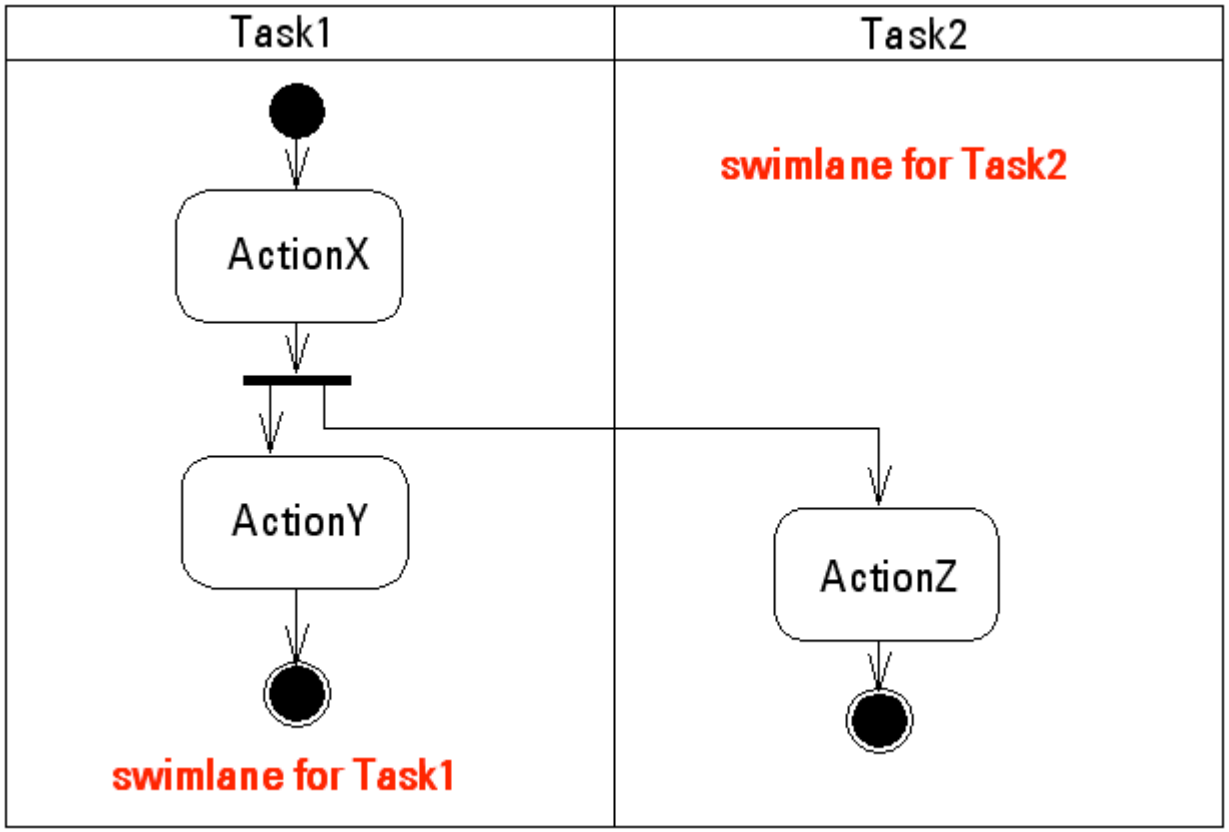
a) Partition using a swimlane notation

b) Partition using a hierarchical swimlane notation

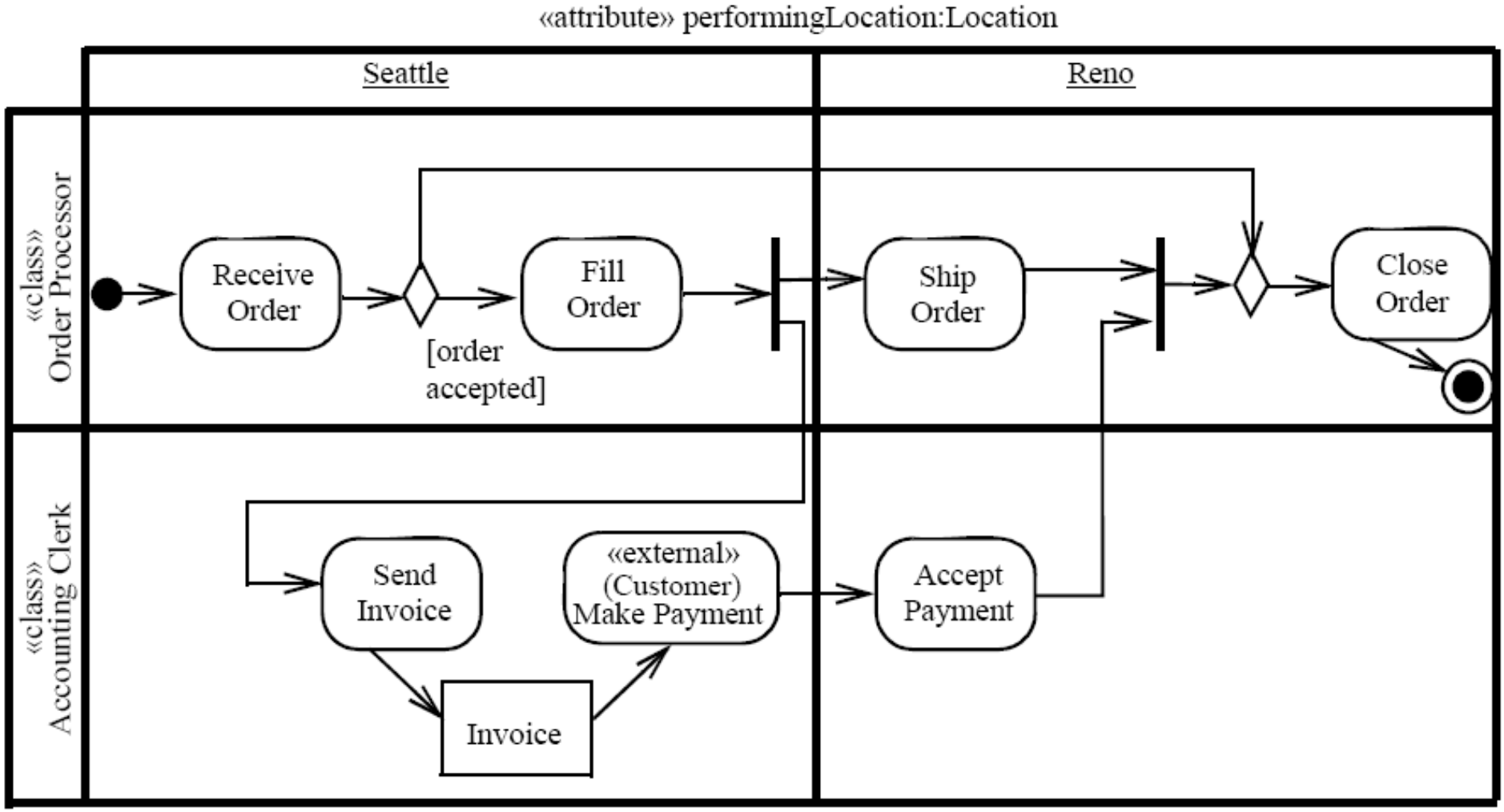


c) Partition using a multidimensional hierarchical swimlane notation

Simple example:



More elaborate example with multidimensional swimlanes:



Structured Activity Nodes

A structured activity node is an executable activity node that may have an expansion into subordinate nodes. The subordinate nodes must belong to only one structured activity node, although they may be nested.

A structured activity node can be thought of similar to a block in a programming language.

A structured activity node defines its own name space.

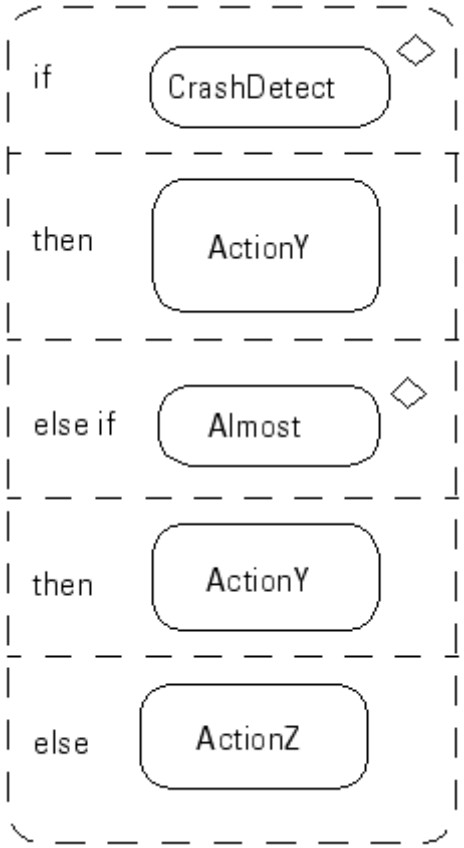
A structured activity node acts like a simple node regarding data and control token flow.

The notation of structured nodes borrows from the Nassy-Shneiderman notation (DIN 66261).

We look at two types of structured nodes: conditional nodes and loop nodes.

Conditional Node:

A conditional node represents an if-then-else like structure.
The condition is marked by a small diamond shape.
Conditional nodes can be nested.
The different regions are separated by dashed lines.



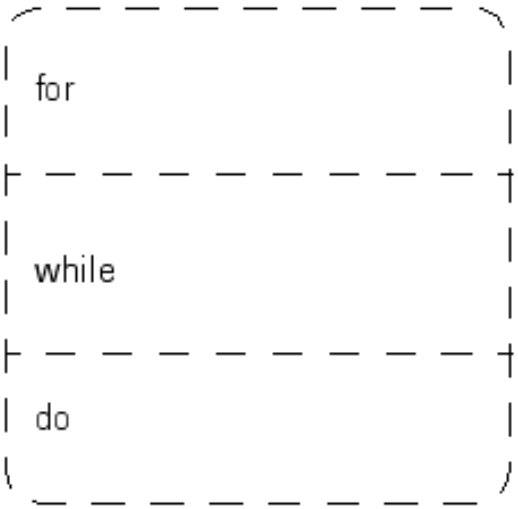
Loop Node:

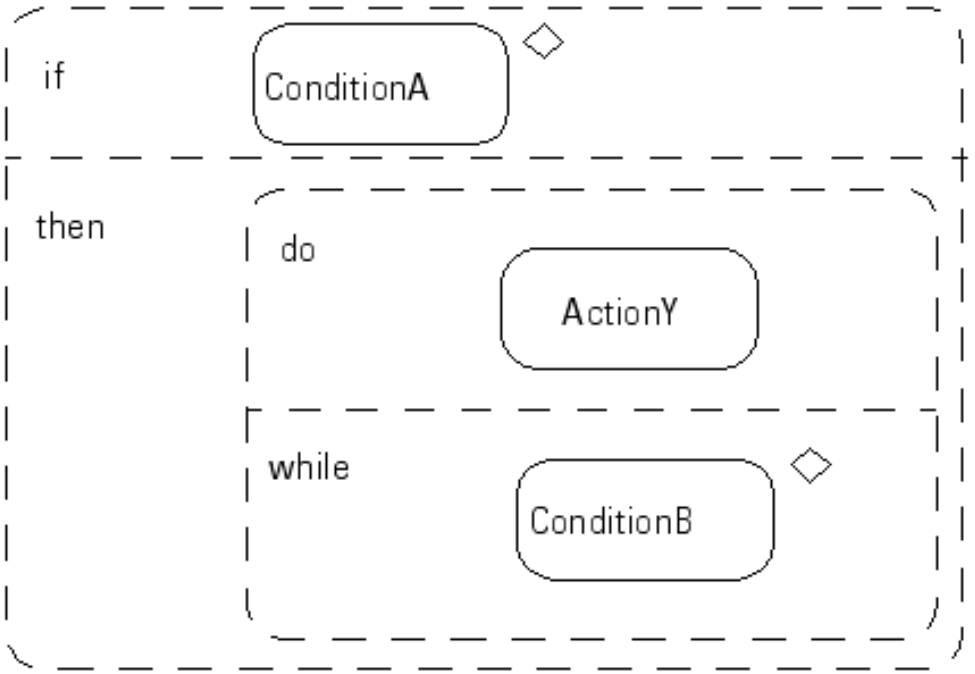
A loop node represents a loop structure.

It consists of:

- Initialization region
- Test region
- Execution region

It can be used to model while-do and do-while loops.





Exception Handling

An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node.

Exceptions can be raised by external events.

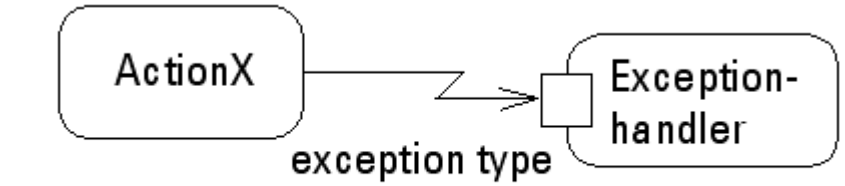
Exception can be raised by the system itself (e.g., software interrupts, RaiseExceptionAction).

Exceptions can interrupts executing actions.

Exception terminates regular execution, no return.

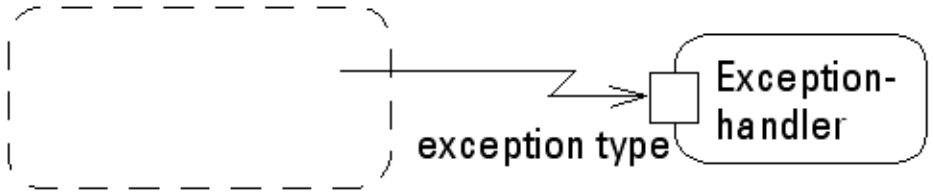
An executable node can be protected by an exception handler node.

A protected node may resume execution after return from the exception handler.



protected node

It is possible to protect an entire region by an exception handler.



Interruptable activity region with interrupt edge

Points to Remember