Specifying Objects' Behaviour

Interaction diagrams show message-passing behaviour between a collection of objects. Each diagram illustrates a single transaction, often covering a short period of time.

Over its whole lifetime, an object may participate in many interactions. Depending on its state at the time, it may respond to messages in these interactions in different ways at different times.

We need a way of giving a complete and concise specification of an object's behaviour. This can be done by defining a *state machine* for the object.

State machines are shown on *statechart diagrams*.

Object Protocol

Objects interact with their environment by receiving and sending messages.

Some objects can be sent any message at any time, and will respond sensibly to each request.

Other objects have a specific *protocol* that specifies the sequences of messages that the objects ‘understand’. Sending a message that is not in accordance with the protocol will result in undefined behaviour.

One important use of state machines is to specify objects’ protocol.

State-dependent behaviour

Objects can receive the same message at different times. Depending on a number of factors, the object may respond differently to the same message at times t_1 and t_2.

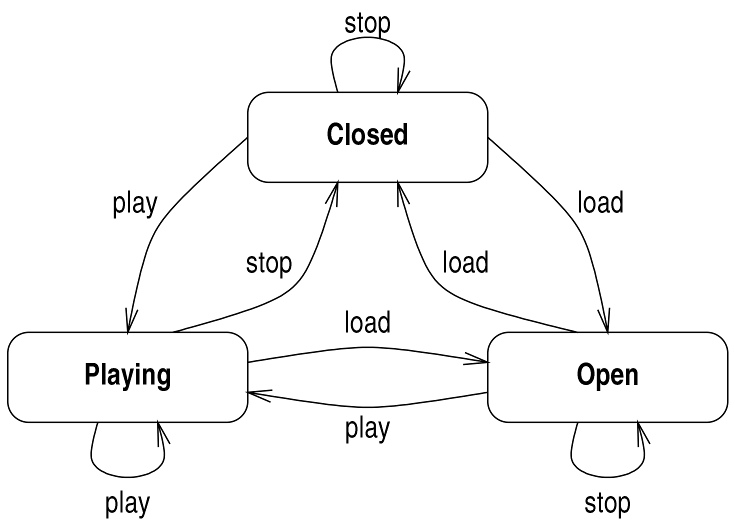
If this is the case, we say that the object is in a different state at those two times. State machines specify the different states that an object can be in and the way in which it responds to the messages that it receives while in each state.

The connection between these two aspects of an object's behaviour is that the state of an object at any time is a function of the messages it has received up to that time. It therefore makes sense to show both aspects of an object's behaviour on a single diagram.

State machine concepts

Imagine a CD player whose drawer could be open or closed, or which could be playing a CD. The user controls the player by means of three buttons to play, stop playing or load a CD.

A statechart could be drawn to describe this CD player.



The basic components of a statechart are:

States At different times an object will be in different states, in the sense of reacting differently to the same stimulus.

Events Correspond to the messages the object receives (presses of buttons in this case).

Transitions Arrows showing how an object moves from state to state in response to events.

State machine semantics

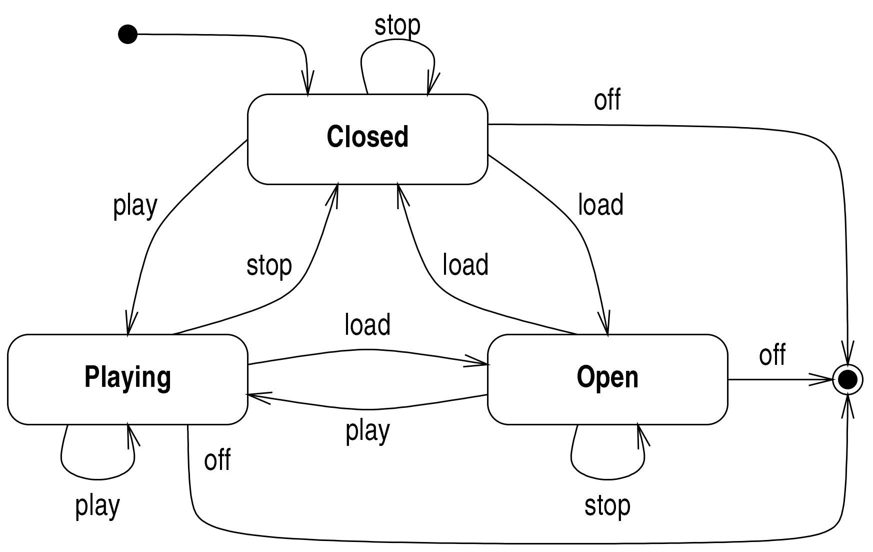
A simple state machine such as the one above can be thought of as ‘executing’ in the following way.

- At any given moment a object is in one of the states shown on the diagram. This is known as the *active* state.
- Each of the outgoing transitions from the active state is a candidate for *firing*.
- When an event is detected, a candidate transition labelled with the detected event will fire. The active state becomes inactive, and the state at the other end of the transition (which may be the same state) becomes active.
- If no outgoing transition from the active state is labelled with the event detected, that event will simply be ignored.

Events very often correspond to calls to an object’s operations: these are sometimes called *call events*.

Initial and Final States

Special states model the creation and destruction of an object.

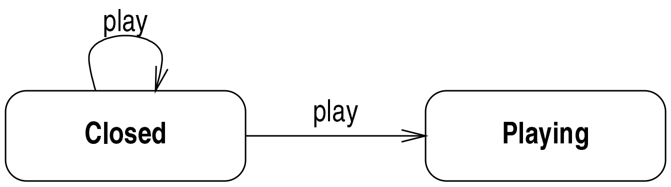


This does *not* mean that the CD player is physically destroyed. What is being modelled is the state of the software controlling the CD player, and this is only active when power is being supplied to the machine.

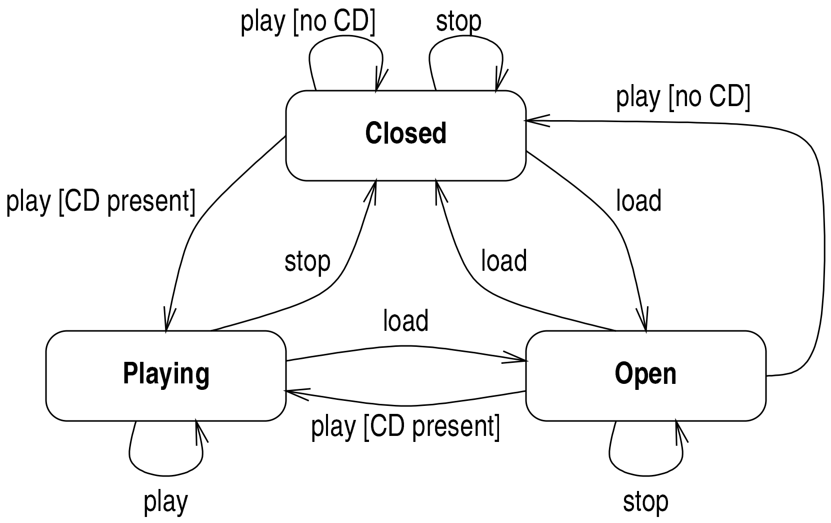
An outgoing transition from an initial state corresponds to an object’s construction. When an object arrives at a final state, it will be destroyed.

Guard Conditions

A statechart is *nondeterministic* if a state has two or more outgoing transitions labelled with the same event.



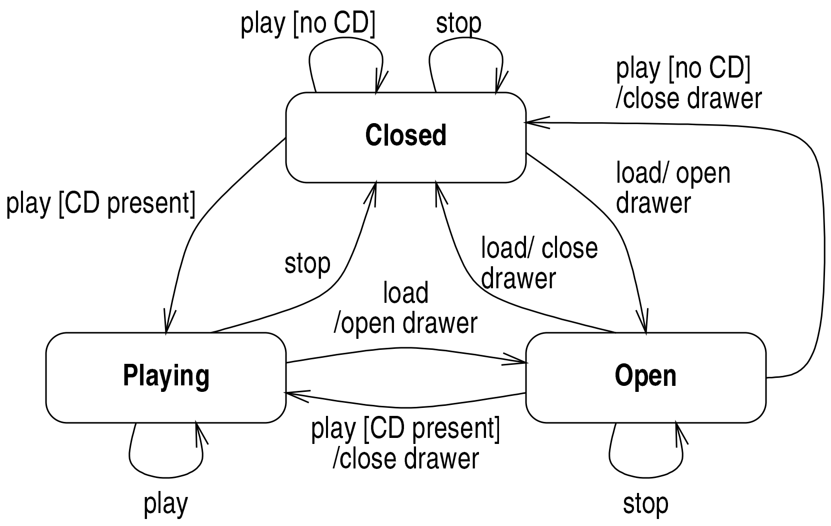
This is not wrong, but it is often desirable to remove the ambiguity, particularly if it doesn’t exist in the real system. This can be done by adding *guard conditions*:



A transition with a guard condition will only fire if the condition is satisfied when the event is detected. In this case, exactly one of the conditions will be satisfied at all times. The nondeterminism is therefore removed.

Actions

Actions can be added to transitions to show what an object does in response to detecting a particular event.



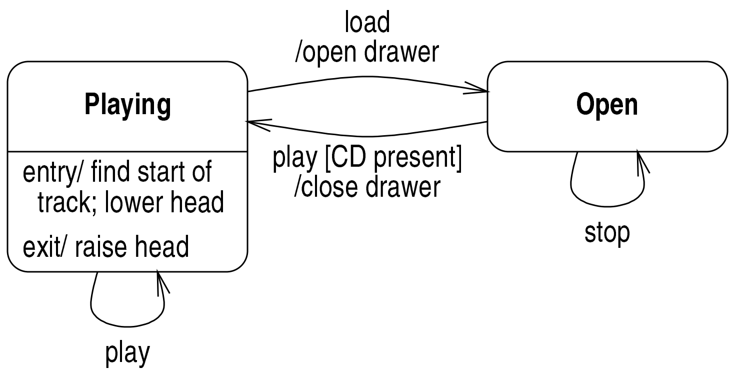
An action is often thought of as taking an insignificant amount of time to complete. More precisely, an action cannot be interrupted by another event but must always run to completion.

Actions can be specified in the following ways:

- Informally, using natural language.
- Using the target programming language.
- Using a default language specified by UML.

Entry and exit actions

Actions can also be specified as *entry* and *exit* actions on a state.

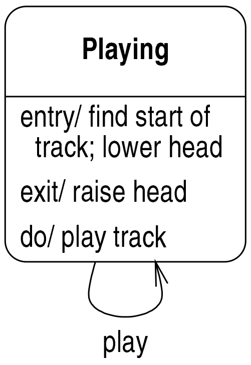


Entry actions are performed every time a state becomes active, and exit actions are performed when a state ceases to be active.

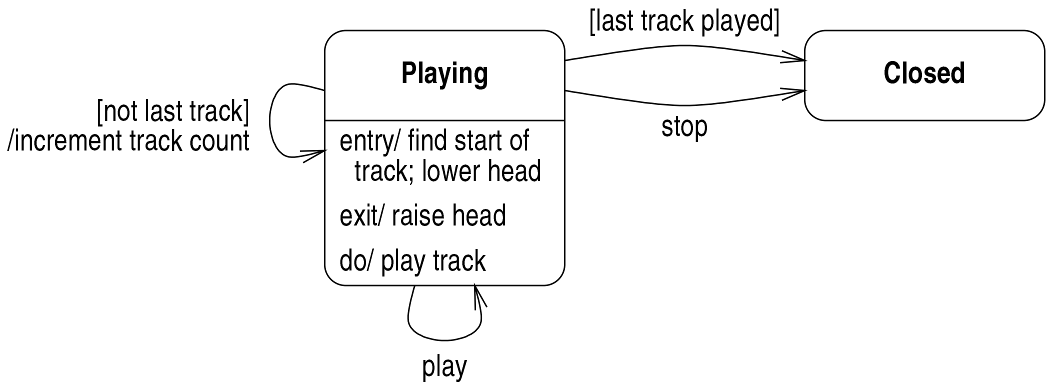
This applies even when a transition starts and finishes at the same state. According to the diagram above, if ‘play’ is pressed when a CD is playing, the playing head will be raised and lowered before playing continues.

Activities

States can specify ongoing activities that carry on while the object is in the state.



Unlike actions, activities can be interrupted by other events. If an activity runs to termination without being interrupted, a *completion transition* fires. Completion transitions have no event, but can have guard conditions to distinguish different possible outcomes.

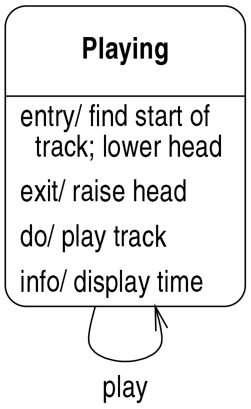


Self-transitions and internal transitions

A *self-transition* is a transition which begins and ends at the same state. It counts as a change of state, so

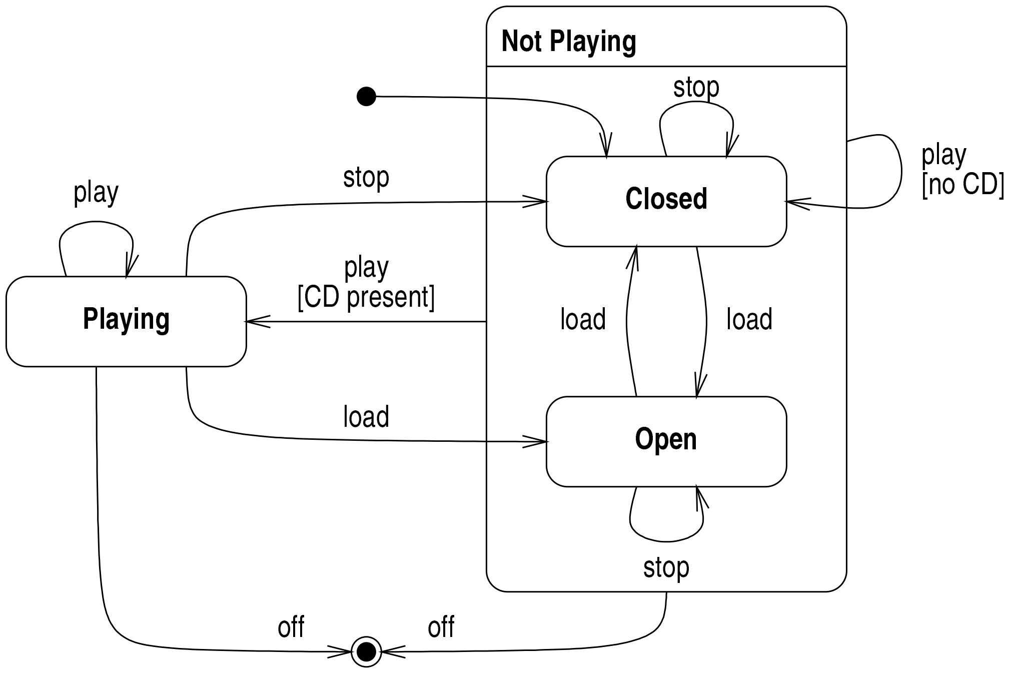
- the state's activity is terminated
- the state's exit action is performed
- any action on the self-transition is performed
- the state's entry action is performed
- the state's activity is restarted

Sometimes an object responds to an event without leaving its current state, and without doing the above. For example, perhaps pressing 'info' causes the CD player to display the amount of time left on the current track. This can be modelled as an *internal transition*.



Composite states

Statecharts can sometimes be simplified by introducing a *composite state* containing a number of substates:

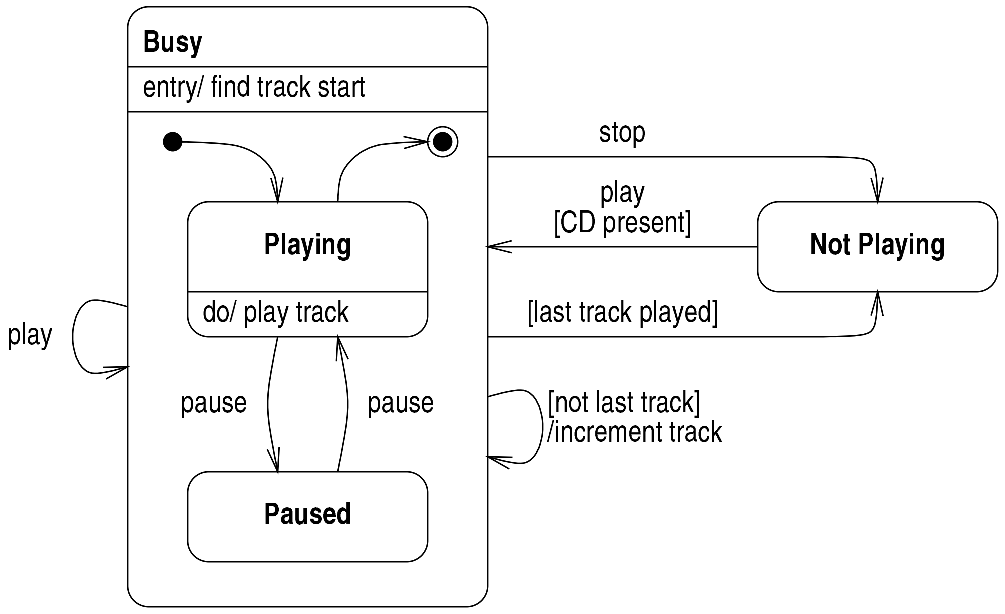


- When a composite state is active, exactly one of its substates is also active.
- An event can cause a transition from either the superstate or the active substate to fire.

Properties of composite states

- An initial state in a composite state indicates the substate that becomes active following a transition which terminates at the boundary of the composite state.
- A final state in a composite state indicates that the state can be left. Arrival at it enables completion transitions from the boundary of the composite state.
- Composite states can have their own entry and exit actions.

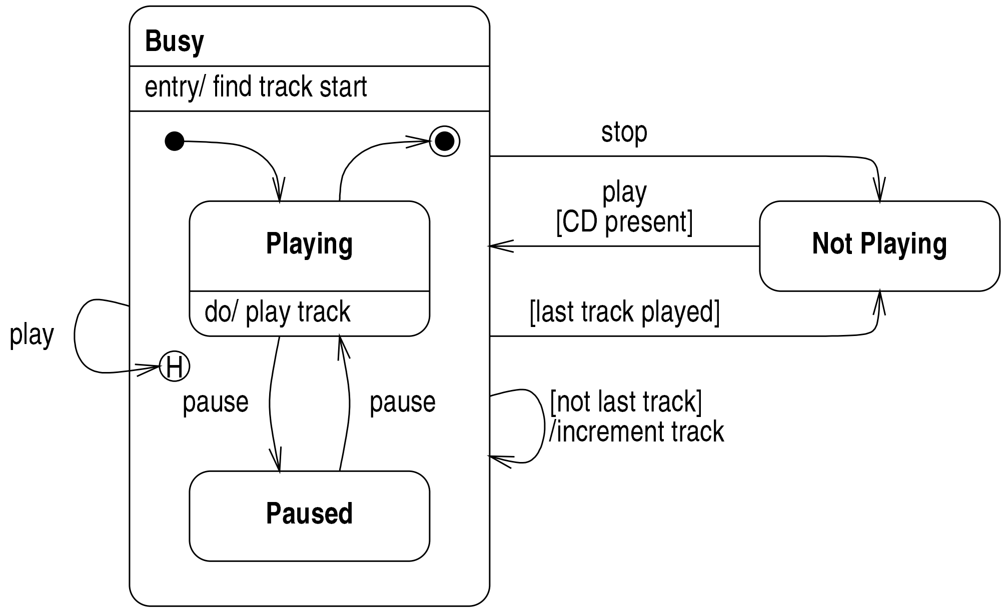
For example, assume that the CD player can be paused and restarted at the same place when it is playing a track by pressing a ‘pause’ button.



History states

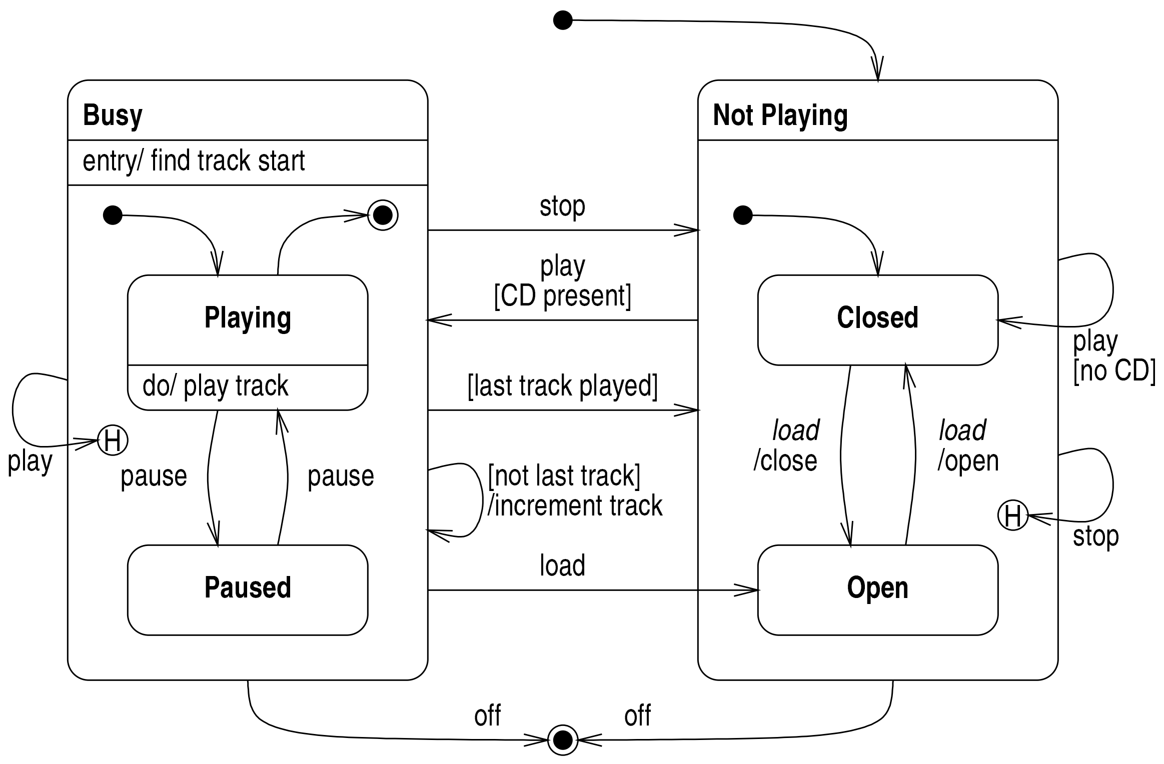
Pressing ‘play’ in the playing state causes the current track to restart. Assume that pressing ‘play’ in the paused state causes the CD player to go back to the start of the track, but remain in the paused state.

This could be shown by two self-transitions on the playing and paused states. As the same event and action is involved in both cases, however, it would be nicer to use a single self-transition on the active state. To do this correctly requires a *history state*.



Complete CD player statechart

In this version of the complete statechart for the CD player, some simplifications have been made in the ‘not playing’ state.

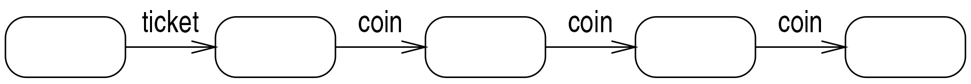


Developing statecharts

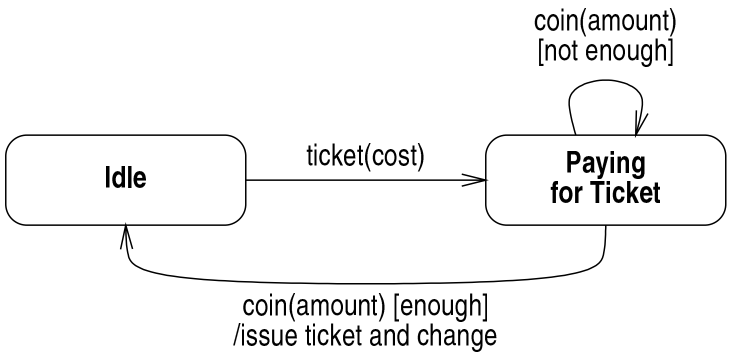
Statecharts can be developed by considering a series of *scenarios*. A scenario describes a particular sequence of messages that might be received by an object in the course of one transaction. Scenarios can often be derived from object interaction diagrams.

For example, develop a statechart for a typical ticket machine where money can be entered and different ticket types selected, in any order.

A first scenario selects the ticket type and then enters the money. A basic statechart for this is:

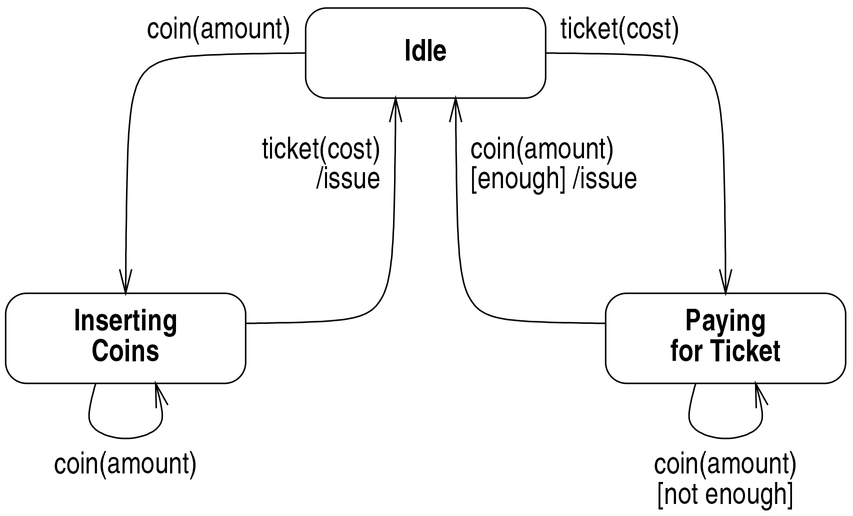


This can be refined by dealing with iterable events, and naming the states:

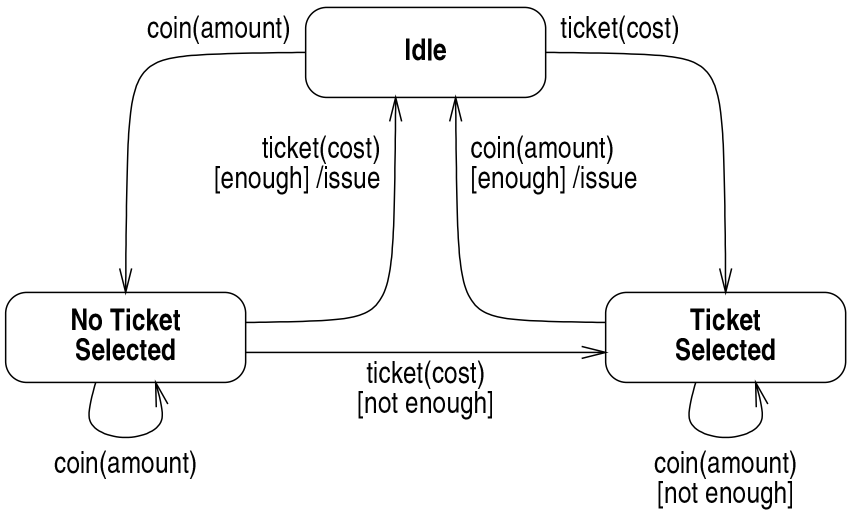


A second scenario

A second scenario allows money to be entered before the ticket type is selected. This requires additional transitions and a new state:



We can refine this by allowing the ticket type to be selected at any time:



coin(amount)
Idle
ticket(cost)
ticket(cost) [enough] /issue
coin(amount) [enough] /issue
No Ticket Selected
Ticket Selected
ticket(cost) [not enough]
coin(amount)
coin(amount) [not enough]

Time events

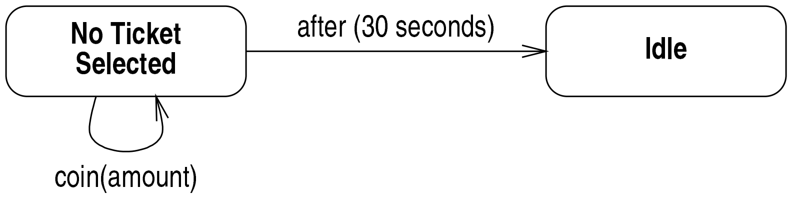
One requirement on a realistic vending machine is that it should provide a *timeout* capability: if 30 seconds, say, pass without any external stimulus being detected, the current transaction should be automatically aborted.

A timeout should be modelled as a transition, but a special kind of event is needed: the whole point, after all, is that these transitions should fire when *no* event is received.

UML defines two special *time events* which can be used in these cases.

- **after (***time***)**: fires a specified time after the last state was entered.
- **when (***time***)**: fires at the specifed time.

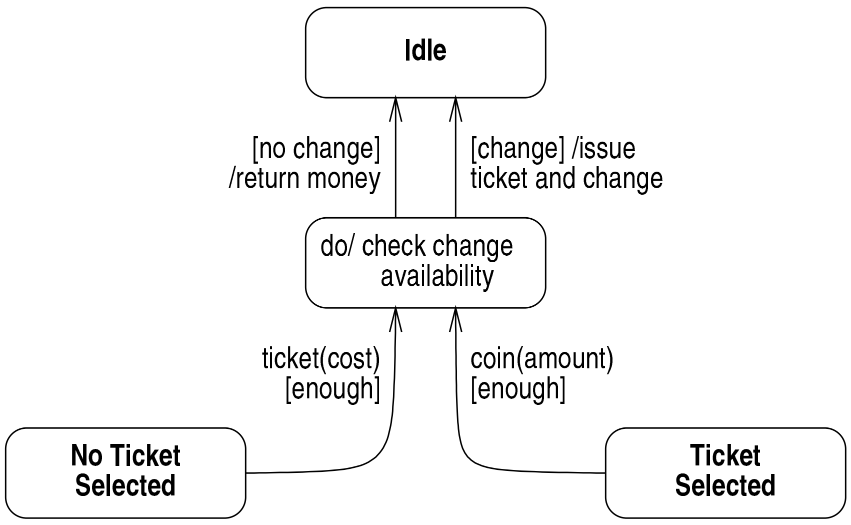
In the ticket machine, the first of these could be used as follows:



Activity states

Two transitions lead back to the idle state after successful completion of a transaction. In each case, we need to check whether any change required is available; if it is, the ticket should be issued, and if not the money entered should be returned.

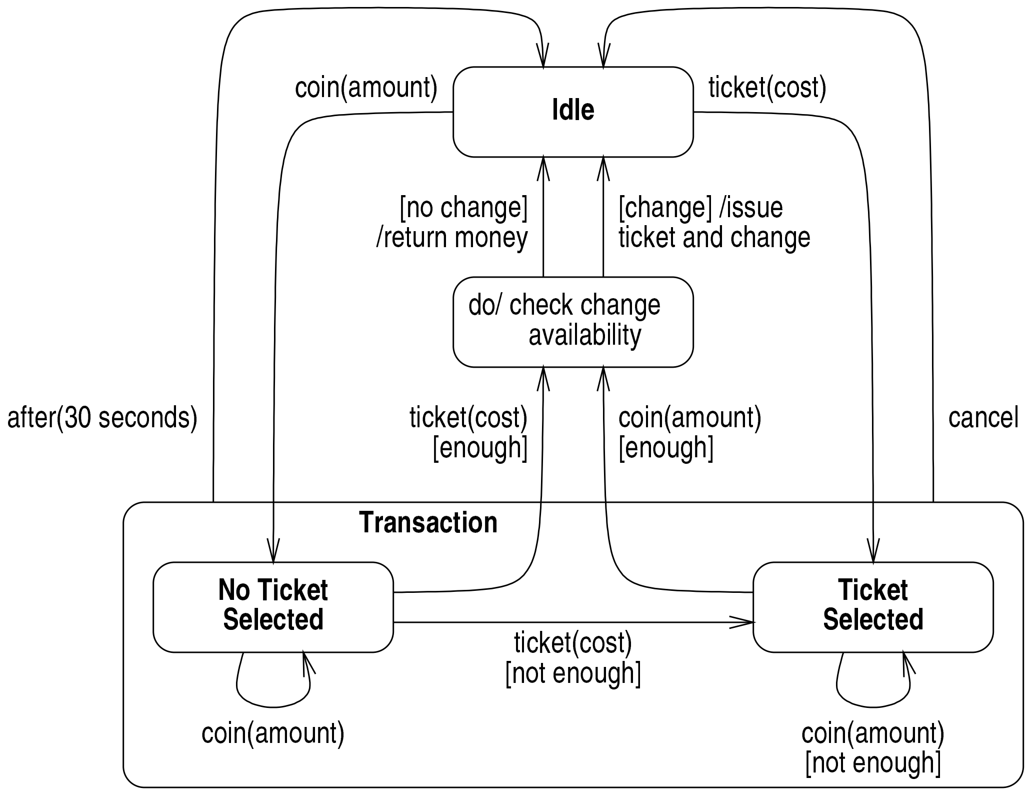
To avoid repeating this test and its outcomes, an *activity state* can be used to simplify the structure of the statechart.



An activity state represents an internal process of the object being modelled. Only completion transitions lead from it, meaning that external objects cannot interrupt the process.

Ticket Machine Statechart

The complete statechart adds a composite ‘transaction’ state to minimize the number of transitions necessary to show timeout and cancellation events.


Idle
coin(amount)
ticket(cost)
[no change] /return money
[change] /issue ticket and change
do/ check change availability
after(30 seconds)
ticket(cost) [enough]
coin(amount) [enough]
cancel
Transaction
No Ticket Selected
Ticket Selected
ticket(cost) [not enough]
coin(amount)
coin(amount) [not enough]