State machines is one of the most common ways to model behavior in UML. It is in particular useful for describing the behavior of objects which transition between different states during their lifetime. Often these transitions are triggered by events of various kinds. A UML state machine can be anything from an informal early-stage analysis model, to a fully formal implementation-style model. In this article we will describe how Rational Software Architect can be used for simulating UML state machines of different kinds.

We will cover four different aspects:

- How to create a state machine based UML model to give an informal description of the behavior of an object in the model.
- How to run a simulation session for state machines, in order to check their consistency and correctness.
- How to make an informal state machine more formal by defining signals and by using the UML Action Language.
- How the different UML state machine concepts will affect a simulation session.

Before reading this article it is a good idea to have a general understanding of Rational Software Architect and UML modeling. It is also useful to have read the “Model Simulation in Rational Software Architect – Getting Started” article since some of the items we will encounter are introduced in this article.

Note that most of the features we will go through refer to news in version 8 and are not available in previous versions of Rational Software Architect.
Creating and Simulating a UML State Machine – an Example

Let's start by creating a UML model which describes an online video-on-demand service. From experience we know that the usage of such a service typically involves several states, so we have good reasons to believe that a state machine is the appropriate UML notation for modeling the behavior of this service.

Note that state machine modeling is typically not the first phase of modeling. In a real-world situation you may start with some use-case and class modeling to identify what kinds of objects the model should contain. You may also use activity modeling for describing some of the scenarios of interaction between these objects. However, when you have identified an object which seem to have a "state-full" behavior, you may start modeling this behavior using a state machine.

Since the purpose of this article is to study state machine based models we will assume that we already have created a UML model with the following contents:

As you can see we have only identified one component (called "VoDService") so far. It is very likely that this component later on will be broken down into smaller components, each with a more dedicated responsibility. However, at this early stage we don't bother about this and we start by describing the behavior of the VoDService component using a state machine.

To create the state machine select the VoDService in the Project Explorer and perform the context menu command Add Diagram->State Machine Diagram.

The main purpose of our Video-on-demand service is to let customers buy movies which will be streamed to their computers or home entertainment systems. We therefore give the name "Buy Movie" to both the state machine and the state machine diagram.
Now let us edit the diagram and describe the states which we think our service component will enter when a user buys a movie. Basically we expect that the user starts by logging in to the service if he already has a user account. If not, he will start by creating an account. Then he will browse through the titles we have to offer, select one movie to watch, pay for it, and finally we will start streaming the selected movie to him. The state machine diagram will look something like this:

![State Machine Diagram](image)

This state machine is an initial high-level description of our service, without much details. However, already now we can get some feeling for the behavior of this state machine by running a quick simulation, to step through what we have modelled to far.

We start the simulation directly from the state machine diagram using the context menu command *Execute As - Model*.

When the simulation starts we are asked if we want to switch to the Model Execution perspective.
Since this is a good idea we accept this suggestion.

The model execution perspective by default shows the Debug View in its top-left corner. From this view we can perform the Step Event command in order to step through the state machine step by step.

Each time we press the Step Event button the “next to execute” marker is moved to the next symbol that is about to execute. In a state machine diagram this marker may appear on transitions and on pseudo-states. The initial node is an example of a pseudo-state, and it is the first element that will execute when we simulate our state machine.

Note that the “next to execute” marker does not appear on states. Instead we will see a green border around a state when it is active.

As you may have noticed, all transitions in our state machine are informal. That is, we have only given them an informal name, and have not specified any signals which may trigger them. Therefore we will be asked which of the outgoing transitions to execute when WaitForLogin becomes the active state:
We select the "log in with existing account" transition to simulate what will happen in case the user already has an account. Then we continue to step through the state machine by means of the Step Event command until we reach the final state.

To see the history of executed elements we can turn on a preference. Click on the view menu in the Debug View and perform Model Execution – Animation Preferences...

In the Preference Editor set the Mark executed elements preference to Colorize executed elements. Now we can see which parts of the state machine we have executed so far:
We can now restart the simulation (using the command *Restart* in the context menu of the simulation session node in the Debug View) and this time select the "create new account" transition to also cover the CreateNewAccount state.

When we have completed our initial simulation of this, still very informal, state machine, we may decide to make it slightly more formal. For example, we may define the signals which may trigger the transitions from the WaitForLogin state. We create a class diagram in our package and add two signals:

Then we add a trigger for these signals on the "create new account" and "log in with existing account" transitions, using the Properties Editor.
In the final implementation these signals should be sent when the user presses some buttons on our Video-on-demand website. However, since we have not created this website yet we will run a simulation where we send these signals manually.

Start a new simulation session using the *Execute As - Model* command in the same way as before. Then step until we reach the state WaitForLogin. This time we don't get prompted to select one of the outgoing transitions. Instead we get the following information when we perform *Step Event* in this state:

![No more events to dispatch!]

This means that we now have to inject an event, corresponding to one of the signals we defined previously, to proceed. One way to do this is to use the Events view. Start by selecting the VoDService instance in the Debug View and perform the command *Populate Event View* from its context menu.
This command computes all signals which can be received by the selected instance. In our case it will look in the state machine and realize that two signals can be received. It will then create formal events for these signals and add them to the Events view:

Note that the formal event for the LogIn signal by default gets two wild-card arguments, corresponding to the two parameters we defined for that signal (userId and password). Let us now inject the LogIn signal event to our simulated VoDService component. We can do this by dragging the LogIn formal event from the Events view and dropping it onto the VodService instance in the Debug View. When doing so we will get prompted to replace the wildcard arguments with real string values.

Type two strings separated by a comma, and press OK. In the Debug View you can see that an event now is waiting in the input queue of the VoDService instance. If you select it you can also see and edit the event arguments using the Variables view:
Now perform *Step Event* and see how the event is removed from the Debug view (the event is said to be consumed), and the "log in with existing account" transition gets marked as the next transition to execute.

![State machine diagram with event removed from Debug view](image)

The rest of the simulation session works just the same as before.

Another step in making the state machine more formal is to add some action code to it. Such action code should be UAL (UML Action Language) code, and is typed by selecting an element in the state machine diagram and then in the context menu perform the *Show Code View* command. The Code view will appear, and you can use it for entering the action code for the selected element.

For a fully formal state machine implementation the UAL code could actually be the production code that performs the logic of running the state machine. However, writing such code for all states and transitions in our Video-on-demand service is beyond the scope of this article. Instead we will just show that for the purpose of simulation any Java code can be typed in the Code view. This makes it possible to add some simple logging for the "log in with existing account" transition:

```
System.out.println("LogIn with userId = " + msg.userId + " and password = " + msg.password);
```
Note that the event that triggers the transition can be referred to from the action code by means of the 'msg' identifier. This variable is typed by the Login signal we have defined previously. It is important that the signal parameters are defined to be public. If they were private or protected we would not be allowed to access them from the transition code without calling an access method.

Now let's run the simulation once more. We inject the Login signal event just like before. When the "log in with existing account" transition has executed we can look in the Console view to see what was printed there:

![Console View](image)

This concludes the first part of this article. We have shown how to create a simple informal state machine model, and how to simulate it to find potential problems and to ensure that all relevant parts of it are covered. We have shown how to make the state machine more formal by replacing informal transitions with transitions that trigger on signals. We have also shown how UAL (or Java) code can be used in the state machine to perform actions when it executes.

The work with further refining the state machine of our Video-on-demand service is likely to gradually make it more detailed and formal. For example, we may expect several of the states to become decomposed into sub states, similar to what we have done with the CreateNewAccount state. We may also suspect that the final implementation of this service must introduce some parallelism so that
1. the service can handle multiple customers at the same time
2. the service is not hung up while a customer watches a movie

One way to handle these problems is to break down the VoDService component into multiple sub components, with their own state machines. We will do this exercise in a future article in this series. In the rest of this article we will go through the different constructs available in a state machine diagram, and how they will affect a simulation session.
State Machine Symbols and Simulation

In this section we will go through the different symbols that can be created in a state machine diagram and their implication when simulating the state machine.

If we look at the symbol palette for a state machine diagram we can see that there are three sections that are relevant for us. The most commonly used constructs are found in the State Machine section:

Then we have a number of different kinds of states in the State types section:

And we have a number of different pseudo-states in the Pseudostate types section:
All of the above constructs are supported during simulation. However, some of the constructs are much more commonly used than others. Also, only some of the constructs allow for UAL code to be entered in order to execute some actions during simulation. Now let's go through the constructs one by one, starting with the most common, and leaving more unusual constructs for last.

**State**

A state can represent either a static situation, where the object is waiting for something to happen, but it can also represent a dynamic situation, where the object is actually performing some actions while the state is active. A state belongs to a region; either a top-level region of the state machine itself, or a region of another (composite) state. Within one region at most one state can be active at the same time. During simulation the active state is marked by a green border. Here is an example where we have a composite state CreateNewAccount that is active. Within a region of this state a substate PickUserName is also active.
The set of states that are currently active in a state machine is called the active state configuration. In the above example the active state configuration consists of the states 'CreateNewAccount' and 'PickUserName'.

There are three different snippets of UAL code that can be associated with a state.

- **Entry**
  This code will run each time the state gets activated.

- **Exit**
  This code will run each time the state gets deactivated.

- **Do Activity**
  This code will start to run when the state gets activated. It is intended to be used for actions that should run as long as the state remains active. It will therefore run in its own thread of execution, in parallel to other behaviors, and will not affect the main state machine execution.

During simulation it is recommended to avoid using UAL code for Do Activity, as it is hard to write any meaningful code there due to its parallel execution. In particular it is not possible to guarantee that code for Do Activity will run to completion, since when the state gets deactivated the Do Activity behavior will stop. From a practical point of view it is therefore better to put such code in the Entry field.

**Initial State**
The initial state is a so called pseudo-state. A pseudo-state is a vertex in a state machine graph which may have outgoing transitions to states or other pseudo-states. The initial pseudo-state in a state machine is where execution begins when the state machine is simulated. If the state machine does not have an initial pseudo state nothing will happen when simulation starts.

It is also possible to have an initial state inside a region of a composite state. In this case it represents a default vertex which will execute when the composite state is entered explicitly. Note however that a composite state also can be entered implicitly by a transition to one of its sub states. In this case the initial state will not execute during simulation. Here is an example:

![State Machine Diagram](image)

If transition T1 executes, the composite state A will be entered explicitly, and the initial state I will in this case execute. However, if transition T2 executes, the composite state A will only be implicitly entered as a consequence of entering state B. In this case the initial state I will not execute.

**Final State**

A final state is a special kind of state which, when entered, will complete its containing region. If the completed region is the last top-level state machine region to be completed then the object that owns the state machine will be destroyed. During simulation this is shown by the removal of the instance from the Debug View.

It is not possible to associate any UAL code with a final state.

---

1 "Vertex" is in UML a common term for all nodes in a state machine graph, e.g. states and pseudo-states. Everything to which you can connect transitions is a vertex.
Region

By default a state machine or a composite state only has one region. However, by adding additional regions to a state machine or composite state it is possible to express the orthogonal execution of what is contained in those regions. Here is an example:

This state machine has two regions. The upper region contains a composite state C which in turn also has two regions.

In each region at most one state can be active at any point in time. For the above example it means that it is possible that both C, D, S1 and S2 are active at the same time (in fact all these states will be active when this state machine has run to completion).

As mentioned above the set of states that are currently active in a state machine is called the active state configuration. Besides from the green border animation the active state configuration is also shown in the Variables view during simulation, if the containing instance is selected. Active states within a composite state will be qualified by the name of the composite state, and different regions are separated by a '||' mark. Here is what may be shown for the example above:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Owner</td>
<td>null</td>
</tr>
<tr>
<td>$State</td>
<td>D</td>
</tr>
</tbody>
</table>
During simulation orthogonal regions will execute one by one to completion. The order in which orthogonal regions execute are not defined. However, if one region cannot continue its execution (for example because it is waiting for a signal event to arrive) then other orthogonal regions get the chance to execute.

**Transition**

A transition connects states and pseudo-states. A transition may contain action code that is the "effect" of the transition, i.e. the code that will execute when the transition executes. A transition may also have a guard condition which defines when it can execute. The guard condition is a boolean expression which must evaluate to true for the transition to execute. It is also allowed to use the keyword 'else' to denote a transition which can execute when no other transition from the source vertex can execute.

A transition may have a set of triggers which define which signals that may cause the transition to execute. However, it is also possible to have transitions without triggers. If a transition has neither triggers nor a guard, it is said to be informal. During simulation a dialog will appear if more than one informal transition can execute. You then have to make a decision which of these transitions that shall execute.

The rules for deciding which transition to execute in a state machine with many regions, composite states with orthogonal regions, and guard conditions, become somewhat complex for the general case. The full details about this can be found in the UML Superstructure standard document.

One small deviation from the specified run-time semantics of state machines concerns what the standard refers to as a completion event. This event is conceptually what causes transition guards to be reevaluated and trigger-less transitions to fire. During simulation we do not use a special completion event but instead let *Step Event* be the command to use in order to "drive execution forward". For example, if you change some condition which makes a guard evaluate to true, you need to perform *Step Event* for the guard to be reevaluated and the transition to execute.

As was already mentioned above it is possible to access the event that has triggered a transition from action code, both in the transition effect and in the transition guard. This is done by means of a 'msg' variable. If the transition has one single trigger, the type of 'msg' is the signal for that trigger. However, if the transition has more than one trigger, 'msg' will have a generic type and in that case you must use the 'instanceof' operator to test it against a particular signal.

**Choice Point**

A choice point is a pseudo-state which can be used to split an incoming transition into multiple outgoing transitions. The guards of the outgoing transitions are evaluated when the choice point
executes, and one of the transitions with an enabled guard condition will be the next one to execute. If the outgoing transitions are informal, a run-time prompting dialog will appear as usual during simulation.

**Terminate**

![Terminate](image)

A terminate is a pseudo-state which, when it executes, immediately stops execution of the state machine. The object that owns the state machine is also destroyed. Hence, the execution of a terminate pseudo-state has the same effect as executing a final state in the top-level region of a state machine.

**Junction Point**

![Junction Point](image)

A junction point is a pseudo-state which can merge multiple transitions into one, and/or to branch one transition into multiple transitions. When it is used for transition branching it is similar to a choice point. However, contrary to a choice point the guards of the outgoing transitions are not evaluated when the junction point executes. Instead these guards are already evaluated when leaving the previous state. A junction point is therefore used for static branching while a choice point is used for dynamic branching.

**Shallow and Deep History**

![History](image)

The history pseudo-states come in two versions; one for shallow and one for deep history. They may only appear inside a composite state; not directly in a state machine. When they execute a substate of the composite state will become active. For a shallow history this substate is the previously active substate, and if this substate in turn has substates they are not activated. A deep history substate works the same except that the entire tree of active substates get reactivated when it executes.

There may be an outgoing transition from the history pseudo-states which will be triggered in case no substate has been active previously.

Here is an example:
When this state machine is simulated 'State1' will first be entered through the transition 'T1' which arrives at the shallow history pseudo-state. Since this is the first time this composite state is entered the outgoing transition to the 'default' substate will be activated. Next, the transition 'T2' will execute. This transition takes us back to the same 'State1' but this time we enter it explicitly which means that the initial pseudo-state within 'State1' will execute next. This in turn activates the 'sub' and 'subsub' states. Finally, the transition to the deep history state of 'State1' executes. This time we will not transition to the 'default' state since we have a previously active state configuration for 'State1'. All these substates will now be activated again, meaning that the animated diagram will look like this:
**Fork and Join**

The fork pseudo-state is used for forking one incoming transition into multiple outgoing transitions which target nodes inside different orthogonal regions of a composite state. The join pseudo-state can then be used to join back such transitions.

Here is an example:

![State Machine Diagram Example](image)

When we simulate this state machine the fork pseudo-state to the left will activate both A and B within the composite state S. Then the join pseudo-state to the right will join the transitions back into one outgoing transition again.

**Composite State**

Creating a composite state from the state machine diagram palette is just a shortcut for creating a state with one initial region.

**Orthogonal State**

Creating an orthogonal state from the state machine diagram palette is just a shortcut for creating a state with two initial regions.

**Submachine State**

A submachine state is similar to a composite state, but instead of defining its own regions with substates etc. it references another statemachine. It is therefore useful in order to reuse a statemachine multiple times from other statemachines.
During simulation a submachine state behaves exactly like a composite state. When it is entered, its statemachine diagram will be opened and simulation proceeds by executing the referenced statemachine.

**Entry and Exit Point**

Entry point and exit point pseudo-states can be used as named points of entry or exit for a composite state. Here is an example:

![Entry and Exit Point Diagram](image)

However, it is perhaps more common to use them on a state machine, like in the example below:

![Entry and Exit Point Diagram](image)

When used on a state machine, the entry and exit points can be referenced from a submachine state using connection point references.

![Connection Point Reference Diagram](image)

A connection point reference appears as a separate pseudo-state in the state machine editor palette. However, they can only be used together with entry/exit points, and during simulation they have no effect other than to transfer control to the referenced entry/exit point.
More Information about Simulation

We can find more information about the simulation feature in the online help for Rational Software Architect available at http://publib.boulder.ibm.com/infocenter/rsahelp/v8/index.jsp and in other articles available on this wiki.

In particular the following wiki articles are relevant for simulation:

- “Model Simulation in Rational Software Architect: Getting Started” This is a very simple walk-through of how to design an activity and run it in a small simulation session.
- “Model Simulation in Rational Software Architect: Sequence Diagram Simulation”. A fairly detailed walk-through of how to design sequence diagram based models and use them for simulation.
- “Model Simulation in Rational Software Architect: Activity Simulation”. A fairly detailed walk-through of how to design activity based models and use them for simulation.
- ”Model Simulation in Rational Software Architect: State Machine Simulation”. This article!
- “Model Simulation in Rational System Architect: Simulating UML Models” A detailed walk-through of all the commands, views and features available in the simulation support in Rational Software Architect.