Model Simulation in Rational Software Architect: Simulating UML Models

Mattias Mohlin
IBM

Rational Software Architect 8.0 provides the possibility to simulate UML models. Simulation is a powerful tool which can be used in many phases of modeling and for various purposes.

- During early analysis, where a model still has an informal nature, simulation may be used as a means to obtain a better understanding of the dynamic model behavior, to detect modeling mistakes early and to validate that specified scenarios work as intended.

- During deployment planning, simulation can be used in order to study the impact of running different scenarios against one or many deployment topologies, which either already exist or are planned to be implemented. This can help in finding communication bottlenecks and to choose the optimal deployment topology. By producing graphical visualizations of the communication scenarios in a topology, it also becomes easier to communicate with stakeholders that are not familiar with UML modeling.

- During development, a model may be more formally defined using expressions and actions written in the UML Action Language (UAL). This increases the expressive power significantly but also the risk of introducing bugs in the model. Simulation of such a model is similar to debugging of traditional source code and is the most efficient way to find and fix problems.

In this article we describe how to use the simulation capabilities in Rational Software Architect 8.0. All tool features are described with screen shots where possible.
1. UML Scope
All kinds of UML behaviors can be simulated:

- Activities (defined using activity or interaction overview diagrams)
- Interactions (defined using sequence diagrams)
- State machines (defined using state chart diagrams)
- Opaque behaviors (defined using UAL)

During simulation the relevant diagrams are animated to show information about the simulation. For example, the element that is the next one to execute is marked by a decorator ( возле диаграммы ).
2. Starting a Simulation Session
The easiest way to start a simulation session is to do it from the context menu that appears when right-clicking a UML diagram. Select Execute As – Model to start simulation.

The behavior that will be simulated is obtained from the context of the selected diagram. If the diagram is not related to at least one executable behavior the Execute As – Model command will not be available in the context menu. If, on the other hand, the diagram is related to more than one executable behavior a dialog will appear that lets you choose which of these behaviors to execute.

For example, a class may contain many behaviors, so if the Execute As – Model command is invoked on a composite structure diagram in that class, the above dialog will appear and will list all behaviors defined in the class.

The Execute As – Model command is also available in the context menu of certain elements shown in the Project Explorer, such as classes, behaviors and diagrams.

When a simulation session has been started by launching a behavior, the diagram that defines the behavior will be opened, and the "next to execute" decorator is placed on the first element that will execute in the behavior. For an activity this is an initial node, for an interaction it is one of its top-most occurrence specifications, for a state machine it is an initial pseudo state and for an opaque behavior it is the first line of action code. For an activity the activity diagram may look like this when a simulation of the activity has just been started:
At this point in time the initial node has not yet executed, but the decorator shows that it is the
next element that will execute when the simulation proceeds.

Another thing that happens when starting a simulation session is that Rational Software Architect
suggests that you switch to the Model Execution perspective.

![Confirm Perspective Switch](image)

The Model Execution perspective contains the views that are useful during simulation. In
particular it contains the Debug view which is described below.
3. Debug View

The Debug view is the view from which many simulation commands are performed. The Debug view also shows a run-time view of the simulated model in the form of a tree. Most of the simulation commands operate in the context of an element that is selected in this run-time model. The picture below shows the Debug View and all kinds of elements that may appear in the run-time model. The most common simulation commands are also highlighted.

The following elements may appear in the run-time model:

1. **Launch.** This is the top-level element in the run-time model and it acts as a container for the other elements. Everytime you launch a new behavior (for example using the *Execute As – Model* command) a new launch element will appear in the Debug view. Rational Software Architect allows you to have more than one launch at the same time. Usually only one launch is active at any point in time, and the others are terminated because their sessions have run to completion and have been terminated. However, it is also possible to launch many behaviors and execute them in parallel, and in this case the Debug view will show one active launch element for each. It should be noted that multiple active launches can be confusing and is only recommended to be used by the experienced user. When a launch is terminated the text "TERMINATED!" appears after its name:

```
sm_Port[UML Model Execution] - TERMINATED!
```

You can remove a terminated launch by selecting it and then press the Delete key. You can also use the "Remove All Terminated" command in order to remove all terminated launches from the Debug View.

2. **Session.** A session represents the simulation of a behavior. Usually there is exactly one session within each launch element. However, as we will see later it is possible to run multiple simulation sessions automatically as part of one single launch operation. In that case there may be multiple sessions under a single launch element.

Each session shown in the Debug view has a unique number that identifies it and is shown in its label.
3. **Active instance.** This element represents an instance of a UML classifier that has a behavior. UML calls these classifiers "behaviored classifiers". A simple example of a behaviored classifier is a class that has an activity behavior.

![Recruitment](image1.png)

![Hiring process](image2.png)

An active instance has a name and a type. The type is the UML classifier and the name is usually just a number that is unique within the containing session. The first active instance gets number 1, the second 2 and so on.

4. **Part.** A session always has exactly one top-level active instance which is typed by the classifier that contains the simulated behavior. However, that classifier may contain composite properties typed by other behaviored classifiers. An active instance for the containing classifier will in this case contain other active instances corresponding to those behaviored classifiers. Such nested active instances are known as parts.

A part is a regular active instance and hence shares the same icon in the Debug View. However, the name of a part is the name of the composite property where it is located, followed by the active instance number in parenthesis.

5. **Port.** A port may appear as a child element of an active instance. It represents the instance of a port that is defined for the classifier that is the type of that container active instance.

6. **Event.** An event is the instance of a UML signal. Events can be sent to active instances, either directly, or indirectly through a port owned by the active instance. Conceptually every active instance has an event queue where events that have been sent to that active instance wait until they are dispatched. The Debug view shows this event queue by listing available events under each active instance. The first event in the queue appears as the first child element. When an event is dispatched to an active instance it may trigger some action within its executing behavior. For example, an AcceptEventNode in an activity can wait for a certain event to arrive. When the event is present at the front of the event queue it may be dispatched, and then trigger the execution of the AcceptEventNode.

A simulation session may either be running or it may be suspended, for example because a breakpoint has been hit, or the user has manually suspended it by means of the Suspend command ( ). The run-time model of the Debug View is updated only when simulation is suspended. If the simulation is resumed by means of the Resume command ( ) the Debug View will only show the Launch and the Session elements until the next time the simulation is suspended.

### 3.1. Launch Configuration

When you invoke the *Execute As – Model* command in order to start a simulation session, a launch configuration is created which contains all information that is required in order to start the simulation session. Usually you don't need to bother about this launch configuration. It will be created automatically, and if an existing launch configuration for the selected element already exists (because it has been launched previously) then that launch configuration is reused. However, if you want you may look at the launch configuration in the Launch Configuration...
dialog. To do this, invoke the *Execute As – Execute Configurations...* command. For example, this dialog allows you to give a descriptive name to the launch configuration. It also allows you to change the element that will be simulated when the launch configuration is launched.
4. Stepping Through a Simulation Session

A common way of stepping through a simulation session step by step is to use the *Step Event* command ( ). If the run-time model contains an active instance that has a non-empty event queue then the first event in that event queue may be dispatched when Step Event is performed. However, even if there are no events visible in the run-time model Step Event may still do something when performed. This is because a model may contain internal events which are not shown in the run-time model because they do not belong to a particular active instance. There are many kinds of internal events, each of which represent "something that happens". For example, just after a simulation session has been started an internal event is injected to the model which represents the ability for an element to execute. This event is always injected as soon as there is at least one element in the model that is ready to execute. Hence, by performing the Step Event command it is possible to step through the execution of a behavior one element at a time.

If all event queues are empty and there is no element that is ready to execute (i.e. the "next to execute" decorator ( ) is not present anywhere in the diagram) then Step Event will not do anything, and an information message appears:

```
No more events to dispatch!
The event queue is empty!
```

This message usually indicates one of two things:

1. The simulated behavior has run to completion.
2. Nothing more can execute in the simulated behavior until a user-defined event is sent to an active instance.

4.1. Locating the Next Element to Execute

If you have many diagrams open it may be hard to find the diagram that shows the next element to execute. Then you can right-click on the Session element in the Debug view and invoke the *Show Next To Execute* command. The diagram that shows the next element to execute is then opened (if not already open) and is brought to front, and the "next-to-execute" element is highlighted so that you can see what will happen next in the simulation.

The *Show Next To Execute* command is also useful if you happen to close the simulated diagram, or if you have multiple active simulation sessions at the same time.
4.2. Controlling the Next Element to Execute

Sometimes it can happen that more than one element is ready to execute. For example, consider the following activity:

The green circles represent tokens. A token is a theoretical concept which is used to describe the execution semantics of a UML activity. A token represents the right to execute. A simple rule that applies to most kinds of activity nodes, is that if an activity node has tokens on all of its incoming edges it is ready to execute.

In the above picture we can see that both A1 and A2 has a token on their incoming edge, and hence they are both ready to execute. Still the "next to execute" decorator is only placed on A1. Why?

The answer is that although many elements may be ready to execute, only one of these elements will actually execute the next time Step Event is performed. And the "next to execute" decorator shows which element this is.

It is possible to manually control the next element to execute in case more than one element is ready to execute. This is done by clicking on a token decorator while holding down the Shift key. The meaning of this operation is to attempt to switch the element that is next to execute, to become the element to which the clicked token is offered. For example, if the right token in the above example is Shift-clicked, the "next to execute" decorator is moved from A1 to A2:
If the clicked token is offered to an element that already was marked as the next element to execute, then that element will execute. Hence, it is possible to Shift-click on tokens as an alternative to performing the Step Event command. Note, however, that tokens are only shown on activity diagrams. For other kinds of behavior it is not possible to control which element to execute next in case more than one element is ready to execute.

If the clicked token is offered to an element that is not ready to execute, nothing happens and an information message is displayed. For example:

Note that a green circle in an activity diagram may represent multiple tokens. In this case the number within the circle tells you how many tokens that are located at that place in the activity. If there are more than 9 tokens at the same location the number won't fit within the circle and you then have to rest the cursor over it and read the tooltip to see how many tokens it represents.

4.3. Restarting a Simulation Session

If you want to run the same simulation session from the beginning again, it is not necessary to terminate the session and then launch it again. Instead you can invoke the Restart command which is available in the context menu of a Session.

Restarting a simulation session means to delete all active instances within the session, and to create new active instances to be executed. Hence you will notice that after a restart all active instances have new ids in their names. However the Session keeps its unique id when it is restarted.

4.4. Resume and Suspend

Sometimes you don't want to step through a simulation step by step, but instead run it at full speed until it either completes its execution or a breakpoint is hit. To do this use the Resume command.

If the simulation session does not suspend automatically after a Resume (for example because execution enters a loop that won't terminate) you may invoke the Suspend command. This will break the execution the next time a UML element is about to execute, and you can then continue stepping through the simulation session from that element.
5. Simulation of Informal and Incomplete Models

A behavior model may be more or less formally specified in UML. Also, the specification of a model may be more or less complete. At the one extreme we have a totally informal model where the majority of the UML behavior is specified using an informal language such as English. The model may only be an early draft and still lack many parts that are planned to be added later. At the other extreme we have a very mature, complete model specified using a formal expression and action language such as UAL. Rational Software Architect supports simulating any UML model regardless of its level of formality and completeness.

Sometimes a model starts as a rather informal specification, and is then gradually refined to become more formal during later development phases. Other models are never intended as anything but informal specifications.

In this section we will discuss the support for simulating models that are incompletely specified and which contain informal elements.

5.1. Simulate – Modify – Continue

The model simulation support has been designed to be applicable from the very beginning of modeling. The ambition has been that you should just have to add a few symbols and lines to a diagram before it can be simulated. It is not required that the model is complete or semantically correct to enable simulation. For example, assume you have created the following very simple activity diagram:

At this early stage you only know that the activity needs to start and stop, and that there are a few actions that somehow have to be performed. There may also be a need for a decision. The above model is an early draft which is not complete as it has several unconnected nodes. Still it is possible to launch a simulation session for this model. Of course, not much will happen except that the start node will execute and then a token is offered to the final node, which then will execute:

However, while the model still is being simulated you can modify it so that one of the actions now will execute. Note that you do not have to terminate the simulation session but can directly...
edit it, save the model, and then continue the execution. However, in order to trigger the modifications you made you may need to use the *Restart* command.

![Diagram](image1)

A model simulation session is updated each time the model is saved. Sometimes the changes that have been made to the model may be impossible to incorporate into the running simulation session. If this happens the following dialog will appear:

![Dialog](image2)

Both the Terminate and the Relaunch buttons will terminate the existing simulation session, but the Relaunch button will in addition launch a new simulation session which contains the new changes.

### 5.2. Run-time Prompting

When simulating an informal model, dialogs may appear that ask you how a particular informal model construct should behave. This happens because due to the lack of formality Rational Software Architect cannot know how to execute the construct without your input at run-time.

As an example consider the following informal decision of an activity:

![Diagram](image3)

This decision is informal since both the decision question ("Decide payment method") and the answer guards ("card"/"cash") are expressed using plain English. When this decision node executes during simulation the below dialog appears:
In this dialog you can select which of the outgoing activity edges from the decision node that shall receive a token. If you select "card" then the left activity edge will get the token, and the "Validate card" action will execute next. If you select "cash" then the right activity edge gets the token, and the "Deposit" action is the next to execute.

The following UML constructs can be informal and hence lead to run-time prompting dialogs:

<table>
<thead>
<tr>
<th>UML construct</th>
<th>Purpose of prompting</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity decision node</td>
<td>Decide the outgoing flow from the decision node.</td>
<td>see above</td>
</tr>
<tr>
<td>Alternative combined fragment</td>
<td>Decide which interaction operand to execute.</td>
<td><img src="alt_example.png" alt="Example" /></td>
</tr>
<tr>
<td>Loop combined fragment</td>
<td>Decide if a loop with an unlimited upper number of iterations should continue to run.</td>
<td><img src="loop_example.png" alt="Example" /></td>
</tr>
<tr>
<td>Option combined fragment</td>
<td>Decide if the interaction operand should execute or not.</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Choice pseudo state</td>
<td>Decide which outgoing transition to execute.</td>
<td></td>
</tr>
<tr>
<td>Guarded transition</td>
<td>Decide which outgoing transition to execute.</td>
<td></td>
</tr>
</tbody>
</table>
6. **Breakpoints**

Breakpoints can be set on executable elements in a behavior. If such an element is about to execute, and the attached breakpoint is enabled, then the simulation session will be suspended just before that element executes.

A breakpoint can be turned on or off (toggled) on an element from its context menu using the **Toggle Breakpoint** command. If the selected element can have breakpoints at more than one location a submenu appears that lets you choose where to toggle the breakpoint.

Breakpoints are shown in diagrams using small blue circles. For example:

If a breakpoint is enabled the circle is filled, while if it is disabled the circle is hollow.

Breakpoints are persisted in your workspace. This means that they remain also when you restart Rational Software Architect.

Breakpoints are enabled and disabled by using the Breakpoints view. This view also provides many other useful commands for working with breakpoints, such as

- **Remove.** Delete one or many selected breakpoints.
- **Remove All.** Delete all breakpoints.
- **Go to Breakpoint Location.** Navigate to the element to which a breakpoint is attached.
- **Skip All Breakpoints.** Ignore all breakpoints during simulation. This command does the same as disabling all breakpoints, except that it is remembered whether a breakpoint was enabled or disabled, so when the command is performed again, all breakpoints go back to their previous state. This is a useful command in order to temporarily skip all breakpoints. Skipped breakpoints are visualized by an overstrike circle. For example:

  ![Diagram](image)

  - **Export Breakpoints.** Save all breakpoints to an external file which later can be imported into another workspace.
  - **Import Breakpoints.** Import previously exported breakpoints into the workspace.

### 6.1. Run to Here

One common use for breakpoints is to temporarily set a breakpoint on an element and then resume the simulation until the breakpoint is hit, and then finally remove the breakpoint again. This scenario is so common that it deserves its own command *Run to Here*. This command automates all the above three steps so that you can use it for quickly running the simulation up until a certain element in the simulated behavior is next to execute.

The *Run to Here* command is available in the context menu of all executable elements (the same elements for which a breakpoint can be set). Note that there is no guarantee that the simulation will suspend on the selected element. *Run to Here* only means that the simulation will be suspended *at the latest* when execution reaches the selected element. It may be suspended before that element is reached, for example because a breakpoint is hit on another element that executes before the selected element.
7. Events
Many UML behaviors are event-driven, meaning that during their execution they expect certain events to arrive. If these events do not arrive, some parts of the behavior may not execute. During execution of a UML behavior events may arrive in two ways:

1. By the execution of an element that sends the required event.
2. By the user manually sending the required event.

An example of the former scenario is the SendSignalAction node of an activity which sends an event to an active instance.

In this section we will focus on the second scenario, where events are manually sent by the user during the simulation session.

One way to send an event manually is to select an active instance in the Debug View and in its context menu perform the Send Signal Event command. This command brings up the following dialog:

A static analysis is made to determine which signals that the selected active instance can receive. All these signals are listed in the dialog and the user can pick one of them to send. An event will then be created for the selected signal and sent to the selected active instance when the Send button is pressed.

If the selected signal has formal parameters the event should have corresponding actual arguments. Such arguments can be provided in the dialog using a comma separated list of UAL
expressions. Here is an example of a signal with formal parameters, and the syntax to use for specifying actual arguments for an event for that signal.

```
<signal>
  SomeSignal
  a : Integer
  b : Boolean
  c : String
</signal>
```

*Actual arguments example: 5, true, "Hello World!"

If a large number of events have to be sent during a simulation session it can be cumbersome to send them using the "Send signal event" dialog. This is especially true if the events have actual arguments which then need to be provided each time an event is sent. To support more large scale usage of events there is a dedicated view, called the Events view. In this view it is possible to define events which are persisted in the workspace and hence can be reused any number of times. To distinguish the events defined in the Events view from events that exist in the run-time model the term "formal event" is sometimes used. An event that is sent during a simulation session can be created based on a formal event. You can think of a formal event as a template for an event.

The Events view looks like this:

Formal events can be added to the view by means of the Create Event command ( ). This command brings up a dialog that lets you browse for a signal and to specify event arguments. A formal event may have wildcard arguments, denoted by an asterisk (*). When a formal event is sent during a simulation session all wildcard arguments have to be replaced with real values. This is done in a dialog that appears when such an event is sent:
If you type too few or too many actual event arguments, or values that are not type compatible with the corresponding formal signal parameters, an error dialog will appear when you press OK. In this case the event will not be sent.

You can define many formal events for the same signal, each of which can have different arguments. This is useful when you have some common events that need to be sent many times during a simulation session.

To send an event from the Events view you can select the target active instance (or port) in the Debug view and then press the Send Event button ( ). As an alternative you can drag the formal event from the Events view and drop it onto the target active instance (or port).

In addition to the above mentioned commands, the Events view provides several other useful commands in its toolbar, context menu and view menu:

- **Delete Event.** Delete one or many selected formal events.
- **Remove All.** Delete all formal events from the Events view.
- **Open Definition.** Navigate to the signal of the selected formal event.
- **Export Events.** Save all formal events to an external file which later can be imported into another workspace.
- **Import Events.** Import previously exported formal events into the workspace.

### 7.1 Working with a Large Number of Events

It is not uncommon to have a large number of formal events in the Events view. To help you find a particular event to send, the Events view supports filtering the list of formal events based on typing a regular expression in the Filter text field. For example you may type "A" in this field to only see events for signals whose name starts with the letter A.

Another way to filter the list of events is to click the checkbox "Show only events applicable for the selection". When this checkbox is checked the Events view will automatically be filtered based on the selection in the Debug view, the Project Explorer and in UML diagrams. For example, if an active instance is selected then only the signals which can be received by that active instance will be shown. And if a class is selected in the Project Explorer or UML diagrams then only the signals which can be received by behaviors of this class will be shown.

There is also a command which makes it easier to create the large number of formal events in the first place. This command is called **Populate Event View** and is available in the context menu of elements in the Debug view, Project Explorer and in UML diagrams. For example, you may select a UML package and perform **Populate Event View** in order to automatically populate the Events view with formal events corresponding to all signals defined in the selected package. All formal events that are created this way will have wildcard arguments for all formal signal parameters. After the Events view has been automatically populated you can replace wildcard arguments with real values if you want to. You can also use Copy/Paste in order to duplicate formal events, for example in order to have multiple events for the same signal but with different actual arguments.
7.2. Controlling the Event Queue

Events that have been sent to an active instance (either directly or indirectly through one of its ports) are shown in the Debug view in the form of an event queue for the active instance. The event at the front of the queue is shown first followed by the other events that are waiting to be dispatched to the active instance.

As a rule a new event that is sent to an active instance is placed last in its event queue.

It is possible to modify the event queue by deleting events from it. To do this, select an event in the Debug view and press the Delete key.
8. Variables

The Variables view shows model variables of different kinds and their values. It is updated when an element is selected in the Debug view. It can be used for viewing and editing:

- the actual arguments of an event in the event queue of an active instance
- the values of properties for an active instance
- the values of local variables for a UAL stack frame

Here is an example of what the Variables view may show when an event is selected in the Debug view:

![Variables View Example]

You can edit the value of a variable (in this example the actual argument of a signal parameter) by clicking on the old value and typing a new value. Note that when editing string values you do not have to type the enclosing double quotes. They are automatically added.

8.1. Special Variables

For some kinds of selected elements the Variables view shows a few special variables:

- **$Owner.** This is a read-only variable which is shown when an active instance is selected. If the active instance is a part the value of this variable is the active instance that owns the part. It can then be expanded to reveal the variables of the owner active instance. For example:

  ![Special Variables Example]

  If the selected active instance is not a part but a top-level instance, then $Owner has the value null.

- **$State.** This is a read-only variable which is shown when an active instance is selected, if the behavior of the instance is a state machine. The value of the variable is the name of the state that is currently active in the state machine. If the
active state is a substate, then the name will be qualified with enclosing active states too. Below is an example. Note that active states are animated by a green border in the state chart editor during simulation.
9. Execution History

During simulation certain historic information about the execution is collected. One example of such historic information is the set of elements which have executed at least once in the simulated behavior. This information can be shown graphically by animations in UML behavior diagrams, and this can be useful as a simple means of studying which parts of the behavior that have executed at a certain point in time.

The animation of this information is by default turned off, but it can be turned on from the Model Execution – Animation preference page:

There are two alternatives for how to show the information about executed elements:

1. **Colorize executed elements.** With this choice all executed elements will be colorized using a color which by default is pale red, but it can be customized in the Animation Colors preference group ("Executed element color").

2. **Mark executed elements by icon.** With this choice all executed elements will be marked with an icon that is a "greyed out" version of the "next to execute" decorator ( ).

The difference between these two animation choices is shown in the picture below:
You can clear the execution history information for a session by performing the command *Clear Execution History* which is available in the session's context menu. If you have multiple sessions in the Debug view you can clear the execution history for all of them by performing the *Clear Execution History* command that is available in the view menu of the Debug view:

Note that execution history remains for a session even after its launch has been terminated. If a terminated launch is removed from the Debug view, then the session will also be removed and therefore all execution history associated with the session will be lost.
10. Composite Structure and Topology Animation

Up until now we have only mentioned animations in UML behavior diagrams that are shown during the execution of behaviors (activities, interactions or state machines). To repeat:

- Next element to execute (👇) – shown in all behavior diagrams
- Executed element (👆) – shown in all behavior diagrams if the preference is enabled.
- Active states (🟩) – shown in state chart diagrams
- Tokens (1) – shown in activity diagrams

In this section we will discuss some other animations which may appear in diagrams that describe the composite structure or topology of an executing model.

10.1. Composite Structure

As an example consider a class with the following composite structure:

```
sender : ISender [0..1]  C  receiver : IReceiver [0..1]
```

This class has three parts that are connected by connectors.

Now assume that the class has an interaction that describes one scenario of communication between these parts:
It is now possible to simulate this interaction and to show the messages that are being sent and received in the composite structure diagram at the same time. However, these animations are only shown in a composite structure diagram if a preference "Animate interaction messages in composite structure diagrams" is enabled. This preference can be found in the Model Execution – Animation – UML Animation preference page:

In order to see both the sequence diagram and the composite structure diagram at the same time you will need to arrange your windows in Rational Software Architect so that they do not cover each other. Below are screen shots from the first two steps of simulating the above interaction:

Model Simulation in Rational Software Architect: Simulating UML Models
© Copyright IBM Corporation 2010. All rights reserved.
A similar kind of animation can be shown when simulating an activity instead. In this case the preference to enable is called "Animate token flows in composite structure diagrams". As this name suggests what will be shown in this case are token flows that take place when the activity executes. However, only those token flows are animated that go between different activity partitions. The partitions represent the parts that are shown in the composite structure diagram, and hence play the same role as the lifelines do in a sequence diagram. Here is an example of an activity diagram with partitions that correspond to the parts shown in the above example:

When simulating this activity the token flows that traverse the edges that go between partitions will be animated, as shown in the screen shots below.
It should be noted that the above "token flow" animations will not appear by default if the activity edge has a control node as its source or target. This is because by default Rational Software Architect does not support placing control nodes in partitions. There is a preference which can be enabled in order to support control nodes in partitions. The preference is called "Enable control node re-partitioning on diagram" and is found on the Model Execution – Animation – Topology Animation preference page:

It is recommended to turn on this preference before you create any activity diagrams that you plan to simulate.

10.2. Topology
The "interaction message" and "token flow" animations that have been described above for composite structure diagrams can also be enabled for topology diagrams. The main difference between a topology and a composite structure is that the topology is an external description of how different components are linked, while a composite structure always describes the structure of a particular class. For these animations to appear in topology diagrams a preference has to be enabled in the Model Execution – Animation – Topology Animation preference page. There is one preference for showing interaction messages and another for token flows.

It is also required that the topology units are connected to UML elements so that it is clear which UML element they represent. This connection can be established in two different ways:

1. By dragging a UML component from the Project Explorer and dropping it onto a topology diagram. The UML component should be the type of a property which is represented by a UML lifeline and/or activity partition.

2. By dragging a topology unit from the Project Explorer and dropping it onto a sequence diagram. In this case a lifeline will be created which represents a
property that is directly typed by the topology unit. This scenario only works for sequence diagrams; not for activity diagrams.

Once these links have been set-up interaction messages and activity token flows will be animated in topology diagrams just like in composite structure diagrams. The screen shots below show what it may look like:

The ReceiveAccountRequest message is just about to be sent from the customer to the account coordination unit.

The 'account request' edge is being traversed by a token flow. This means that the customer is communicating with the account coordination unit, and this will be shown in the topology diagram.
All deployment links that are either constraint links, dependency links or hosting links can be animated in this way. For constraint links the constraint should be some form of communication constraint (for example "application communication" or "network communication").

It is not required that there is a direct connection between a UML element and a topology unit. Several different "indirections" are supported:

- **Showing communication between container units.**
  A typical topology is hierarchical where a unit may contain other units. For example, a Location unit may contain a Node unit which in turn may contain a Component unit. In this case the Component unit may be connected to a UML element, but links to/from the containing Node or Location unit can still be animated. This is useful if you want to study the communication that takes place at a higher level, for example the communication between different locations.

- **Showing communication between realized units.**
  Often it makes sense to create both a logical (a.k.a. conceptual) topology where units represent conceptual entities, and a physical (a.k.a. concrete) topology where units represent real-world entities such as servers and buildings. In this case it is common to only connect the logical units to UML elements. A topology diagram for a physical topology will still be animated, provided that the physical topology units have been linked properly to the logical topology units by means of realization links.

- **Support for interface realization and generalizations.**
  Sometimes the UML properties that are represented by lifelines and/or activity partitions are not directly typed by the component to which topology units are linked. Instead they may be typed by an interface which is realized by the component, or they may be typed by a general component from which the linked component inherits. All these cases are supported during simulation and
animations will still appear in the topology diagrams. Both ways in which an interface can be realized in UML are supported: the interface realization relationship, or realization based on a port of the component.

This is illustrated in the pictures below, where we assume that the interface I is the type of a property that is represented by a lifeline and/or activity partition, and T is the component to which a topology unit is linked.

Interface realization relationship

Realization by means of port

generalization

Interface realization and
Also note that you can have several topology diagrams open at the same time. All open topology diagrams will be animated. There are many situations when this can be useful:

1. The different topology diagrams may be at different levels of detail. For example, you may have one diagram that shows the detailed communication between individual components, and one that only shows the communication that goes between different nodes.

2. The different topology diagrams may correspond to different alternative topologies. For example, you may have two different topologies that are candidates for being implemented. By showing the communication in both topologies for different simulated behaviors you can make a decision which of the two that is more optimal from a communication point of view.

3. One topology diagram may correspond to an existing (already implemented) topology and others may show suggested modified topologies. By studying communication in the new topologies and comparing them with the communication in the existing topology, you can see if some of the new topologies would be better than the old one.

4. One topology diagram may show a logical topology while the others may show different physical topologies that are linked to the logical topology. This makes it possible to see the communication both in the logical topology and in the various physical topologies that are intended to realize the logical topology.

### 10.3. Historic Messages

The execution history information that is collected during a simulation session can also be visualized by means of animations in composite structure and topology diagrams. In these diagrams the information is visualized by means of arrows which show how interaction messages or token flows have taken place between the components or units shown on the diagram. Here is an example of a topology diagram showing such historic message arrows:
Each colored arrow represents one historic message, and is marked by a number which indicates the order in which the message occurred. For a pair of interaction messages that represent an operation call the corresponding arrows get the same color to indicate that they belong together in a call-reply relationship. In the above picture we can see this for message number 2 and 3.

There are several preferences available which can be used to customize how these historic message arrows are rendered. They are located in the Model Execution – Animation – Historic Messages preference page:

- **Spacing between arrows.** This preference controls how close to each other different arrows are drawn. If you think the diagram looks too cluttered with arrows that partly overlap each other, you can try to increase this number to get more separated arrows.

- **Arrow style.** By default one arrow represents exactly one historic message. If a diagram shows a large number of historic messages it may therefore become cluttered. To avoid this you may change this preference to "Multiple". Then all historic messages that go between the same pair of sender and receiver elements will be represented by the same arrow. Since this means that one arrow may represent multiple historic messages no numbers are shown on the arrows in this case.
- **Show real execution order number of interaction messages.** For interaction messages this preference provides a possibility to let the numbers of the arrows correspond directly to the numbers that are shown in the sequence diagram that defines the interaction. Note that if this preference is disabled it is possible to get multiple arrows with the same number (just like in a sequence diagram).

- **History depth.** This preference determines how old historic messages to show in the diagram. By default all historic messages are shown, but if a number is entered for this preference then only those historic messages that have occurred more recently are shown. For example, if you type 3 for this preference, it means that only the 3 latest historic messages will be shown. Older historic messages will not be shown.

### 10.3.1. Historic Messages View

In addition to being visualized in topology and composite structure diagrams, historic messages can also be viewed in a dedicated view called the Historic Messages view. This view is by default not open, but it can be opened either through the *Windows – Show View – Other* command, or by means of a command *Show Historic Messages View* that is available in the view menu of the Debug view:
The Historic Messages view shows all messages (interaction messages or token flows) that have taken place in the simulation sessions that are available in the Debug view. It contains more information for each historic message than what is feasible to show on an arrow in an animated diagram. Here is an example where the view contains a few historic messages belonging to two different simulation sessions:

The Session and Kind columns are by default not shown, but can be turned on in the view menu. The following information can be shown for each historic message:

- **Session.** This is the simulation session in which the historic message has taken place. The session is identified by its unique number which also can be seen in the Debug view.

- **Order.** This is a number that indicates the order of occurrence for the historic message within the simulation session. The first message that takes place gets number 1, the second gets number 2 and so on.

- **Kind.** This tells whether a historic message is an interaction message or a token flow.

- **Name.** This is a description of the historic message. For an interaction message the same name is used as is shown in the simulated sequence diagram, while for a token flow the name is taken from the activity edge that is traversed by the token. If this edge has no name, the name is instead constructed from the name of the source and target activity nodes.

There are also some useful commands available in the tool bar and context menu of the Historic Messages view:

1. **Link with Editor and Selection ( ).** When this button is pressed the view will be filtered so that it only shows the historic messages that are related to the selection. For example, if a session in the Debug view is selected then only the historic messages that belong to that session will be shown. Another useful filtering is to select a message line in a sequence diagram or an activity edge in an activity diagram to see only the historic messages that occurred at that location in the behavior.

2. **Clear Execution History ( ).** This command is the same command that is available in the Debug view's view menu. It will clear all execution history for all
available simulation sessions, and hence will make the Historic Messages view empty.

3. **Navigate – Show Historic Arrow.** Navigates to the historic arrows (there can be many) which represent the selected historic message. The links for which the arrow is drawn will be selected in all open diagrams. Note that diagrams will not automatically be opened but should already be open when performing this command.

4. **Navigate – Show in Behavior.** Navigates to the location in a UML behavior diagram where the historic message occurred. For a sequence diagram this would be a message line, while for an activity diagram it would be an activity edge.

5. **Copy.** Copies the selected historic messages to the clipboard. This allows you to paste the information into other documents. For example, you can use this command in order to create a legend for the historic messages that are shown in an animated topology or composite structure diagram.

If an arrow is clicked in an animated composite structure or topology diagram, then the historic messages which it represents will be selected in the Historic Messages view.

### 10.3.2. Image Generation

One typical usage of diagrams that show historic message arrows is to export the animated diagram into an image file. It is possible to generate the image files manually from the topology diagram context menu (*Save As Image File*). However, if you have more than a few UML behaviors to simulate this can be a tedious process. Rational Software Architect therefore provides a command for automating the process of image generation. It is called *Generate Historic Images* and is found in the context menu of a composite structure or topology diagram.

The command is applicable when you have at least one simulation session in your Debug view. If there are more than one a dialog will first appear where you can select for which sessions you want the images to be generated.

![Select Sessions to generate images from](image)

The next step is to specify where to save the image files, the image format and the file names.
The File Name field specifies a pattern to use for the names of the generated image files. The following variables can be used in the pattern:

- **name**
  This is the name of the simulated behavior.

- **qname**
  This is the qualified name of the simulated behavior.

- **id**
  This is the id of the simulation session.

You should make sure to specify the file name pattern in a way that will give each generated image file a unique name.

10.3.3. Controlling for which Sessions to Show Historic Messages

When the Debug view contains multiple simulation sessions the historic message arrows in composite structure and topology diagrams will by default only be shown for the last session. In order to show arrows for another session you may drag that session from the Debug view and drop it onto the composite structure or topology diagram. The animations will then be refreshed to show arrows for the dropped session only. You can also drop more than one session on a diagram in order to show arrows for all those sessions at the same time.

10.4. Running Multiple Simulation Sessions

As was already mentioned previously, it is possible to have any number of simulation sessions in the Debug view at the same time. So far we have only described one way of launching new simulation sessions (by means of the *Execute As – Model* command). That command launches exactly one new simulation session, and allows you to interactively step through the simulation by means of the various simulation commands, use of breakpoints etc.

If you have a large number of UML behaviors to simulate, and just want to run each behavior until it completes its execution, then it is tedious to use the *Execute As – Model* command, since you then have to repeat the command once for each behavior, and also have to press the Resume button after each launch to make it run to completion. In addition you must remember to disable or skip all breakpoints to avoid that the simulation is suspended before it has run to completion.

To simplify this work flow Rational Software Architect provides an alternative command for launching simulation sessions, called *Run As – Model*. This command is also available in the context menu of elements that can be executed, but contrary to *Execute As – Model*, it supports...
multiple behaviors to be simulated at the same time. You can select multiple behaviors before you perform this command, or you can select a class with multiple behaviors and in a dialog select to launch many of these behaviors:

When you launch simulation sessions this way you will get one launch node in the Debug view which may have several sessions below it. Here is an example where we have used Run As – Model to run the simulation of two different behaviors:

![Choose Behavior to Launch](image)

When simulation sessions are run using the Run As – Model command, breakpoints will not be hit, and it is not possible to manually control the simulation. However, if any run-time prompting dialogs appear during simulation you have to respond to those dialogs as usual before simulation can proceed. In the title of these run-time dialogs you can see to which simulation session it belongs.

When simulation sessions launched by the Run As – Model command have run to completion the launch will automatically be terminated.
11. UAL Simulation

A UML model may contain snippets of UML Action Language (UAL) code. Typically such snippets appear where UML allows opaque actions and expressions to be specified. The UAL snippets are best edited in Rational Software Architect using its Code View. One UML element may in general be associated with many UAL snippets. For example a UML transition can have a UAL snippet both as the transition guard and as the transition effect.

If a model with UAL code is simulated it is possible to step into the UAL code. This is useful in order to debug the UAL code, very much like how a Java program can be debugged. In order to step into UAL code press the Step Event button in the Debug view's toolbar. This command will suspend the simulation session as soon as a UAL snippet will execute. If no UAL snippet is associated with the current UML element then Step Into becomes equivalent to Step Event, i.e. the simulation session will then be suspended at the next UML element that is about to execute. If the UML element contains multiple UAL snippets you have to invoke Step Into followed by Step Return multiple times until execution enters the UAL snippet you are interested in.

When the simulation session is suspended in a UAL snippet, the UAL code is shown in a UAL editor. This editor is very similar to the Code view, except that it always shows only one UAL snippet (the one that is currently executing). You can use the regular stepping commands of the Debug view in order to step through the UAL snippet:

- **Step Over.** Executes the next UAL action.
- **Step Into.** Executes the next UAL action. If this leads to the invocation of another UAL snippet, then the simulation is suspended in that snippet.
- **Step Return.** Executes the rest of the current UAL snippet, and returns to the context from which it was invoked. If the UAL snippet was invoked from another UAL snippet then simulation suspends in that calling UAL snippet, at its next action. If the UAL snippet was invoked from a UML element, then simulation returns to model context and suspends when the next UML element is about to execute.

You can use the Debug view to see in which UAL snippet the simulation is currently suspended. The Debug view shows a call stack of UAL snippets below each active instance that is suspended in a UAL snippet. The top-most UAL stack frame is where the simulation is currently suspended. Here is an example:

![Debug view example](image)

Here we can see that the activity "Activity1" of a class called "StepInto" is currently suspended in the UAL snippet of an OpaqueBehavior at line 1. We can also see that this UAL snippet has been invoked from another UAL snippet of an action "StepIntoMe" at line 2. To open the UAL editor on a particular UAL snippet just double click on the corresponding UAL stack frame in the Debug view.

Model Simulation in Rational Software Architect: Simulating UML Models
© Copyright IBM Corporation 2010. All rights reserved.
You can put breakpoints at UAL actions by double-clicking in the ruler of the UAL editor. Such a breakpoint will suspend the simulation just before the action is about to execute. A UAL line breakpoint appears in the Breakpoints view just like a model breakpoint, but it can be distinguished from model breakpoints by the line number that appears in its label. UAL line breakpoints are also shown on the corresponding diagram symbol, if such a symbol exists. Its tooltip tells us that it is a UAL line breakpoint and not a regular model breakpoint on the element.

![UAL breakpoint image]

11.1. Supported UAL

The work on standardizing the UAL is not yet completely finished. However, it is already clear that UAL from a syntax point of view can be considered a subset of the Java language. During editing of UAL actions and expressions you will see error markers in case you type something incorrectly. These markers are very similar to corresponding markers used when developing Java applications.

From a simulation point of view any Java statement is valid and can be used in a UAL snippet. This means that UAL specific restrictions are not enforced during simulation. This makes it possible to use well-known Java code to do common things such as logging. Here is an example of a UAL snippet which during simulation prints some logging to the console before and after invoking a UML operation called "foo":

![UAL snippet image]