UML Model Debugger
UML Model-Based Testing

User's Guide

Andrei Kirshin
Dolev Dotan

IBM Haifa Research Lab

Last Modified: May 15, 2007

Table of Contents

1. Introduction

UML 2.0 provides powerful modeling capabilities. While it is best known for its strong structural modeling capabilities using class, component and object diagrams, UML also allows modelers to describe the system's behavior using state machines, activities, interactions, and use cases. In addition, UML is also very flexible, as it can be extended with user-defined elements. These extensions allow defining domain-specific graphical modeling languages, which can still enjoy the full support of UML modeling tools. For all these reasons, UML 2.0 has become widely adopted in the software development discipline.

However, this is not sufficient for some of the present day scenarios of model usage. This is especially true since UML 2.0 provides an enriched and better defined behavioral description mechanism. For instance, to enable model driven development (MDD), programmers must be provided with an environment that allows model debugging and execution. The ability to execute models provides additional avenues for the exploitation of the models in validation, verification, and simulation. The use of executable models enables the visualization of behavior and discovery of defects early in the development cycle avoiding costly rework at later stages.

The move from the traditional to model driven development means the shift not only in methodology but also in the tools required to support it. In the same way as modern development is inconceivable without debuggers, MDD is impractical without model debuggers.

2. Installation

2.1. Requirements

- RSx (Rational Software Architect or Modeler):
  - for release 6.x – install version 3.6.0
  - for release 7.x – install version 3.7.0
- A fast processor and at least 1GB of RAM

2.2. First time installation

- Open RSA's Update Manager (Help -> Software Updates -> Find and Install).
- Choose "Search for new features to install"
- Setup a new update site.
- Choose features from the update site and install. This package contains the following features:
  - UML Model Execution and Simulation (required)
  - Model-Driven Testing Utilities (optional, see Section 5).
  - Voice Applications Extensions for UML Model Debugger (optional, see Section 6)
  - Voice Dialogs Transformations

2.3. Upgrade

- Using the Product Configuration Manager:
  - Open Help-> Software Updates - > Manage Configuration
  - Open the install site where the features are installed, and select the features that you installed (multiple features can be selected by using CTRL-clicks).
  - Right-click and select “Find Updates” from the popup-menu.

- Using the Installation Manager (might take a longer time):
  - Open Help-> Software Updates -> Find and Install
  - Select "Search for updates of the currently installed features".
  - Choose the latest version of MEX and install.

2.4. After installation:

- In the Modeling Perspective click Window -> Show View -> Snippet.
  - If the Snippet View does not stack above the Properties View, move it there (this is a known issue).
- If upgrading, in the Debug Perspective click Window -> Reset Perspective.

3. Designing Executable Models

3.1. Supported UML Subset

The Model Debugger supports structural modeling specified in class and component diagrams, as well as behavioral modeling specified in activity and state machine diagrams. As described in Section 3.3, the actual execution semantics of elements (activity actions, state transitions, class operations, etc) may be specified using Java code snippets. In addition, the Model Debugger supports the integration between the structural and behavioral aspects of a model, as behaviors (state machines and activities) may be used to specify class life cycle as well as operation implementation. Note that behaviors in UML are also classes, and thus may also be instantiated and used outside of the context of any "normal" class. This may be used to specify processes such as business processes, and user interaction dialogs.

The following subset of UML is supported (with limitations described below):

- Structural modeling:
    - Classes, interfaces, enumerations, and relationships
    - Model references
    - Class attributes can reference external Java classes (see Section 3.4).
    - Classes may be designed using class and composite structure diagrams.
    - Operation pre-conditions
- Behavioral modeling:
    - State machines
    - Activities

The following are the current limitations:

- Structural modeling:
    - Class hierarchy may use single-inheritance only (as in Java).
    - Classes may not inherit from external Java classes
    - Relationship classes are not supported
    - Connections in composite structure diagrams are not treated (they must be manually connected in the constructor code snippet for the specified class).
- Behavioral modeling:
    - State machines: Time triggers are not supported
    - Activity diagrams:
        - Actions: we currently support only Opaque Action and Call Action
        - A CallOperationAction can call only the operation of the current activity ("this")
        - Structured activities are not supported.
        - Central Buffer / Data Store nodes are not supported

3.2. Modeling with RSx

Refer to the RSx documentation.

Note: The UML Debugger can be used to debug models in UML projects as well as Java projects. In particular, a Java project should be used in case the model will be used for test generation or for models that reference external Java code.

3.3. Java Snippets

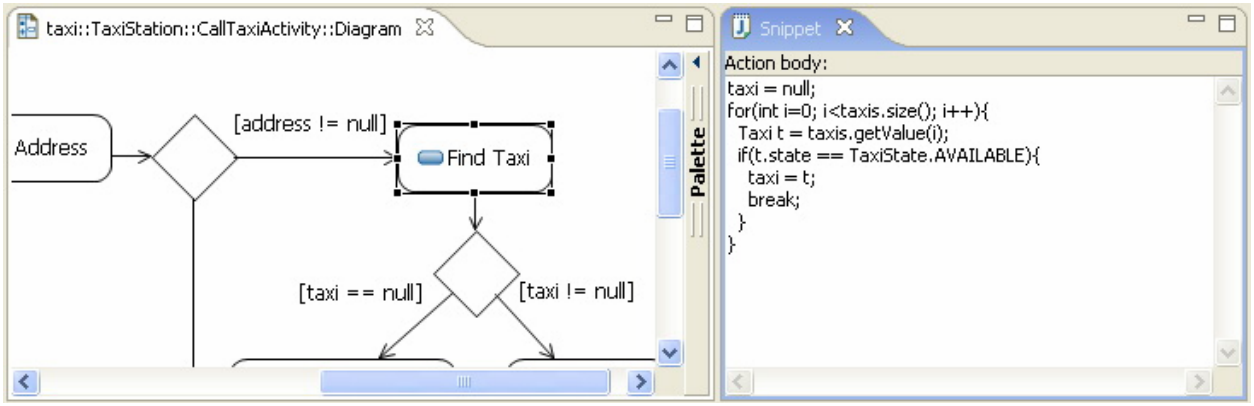
Java snippets can be used to assign executable semantics to the model. They can be attached to:

- actions
- operations
- operation pre-conditions, body conditions and post-conditions (only pre-conditions are treated by the debugger)
- attribute default values
- opaque behaviors
- guards
- change events

The code can make use of the structural aspect of the model, access a standard library of debugger functions (described below), and may reference external libraries.

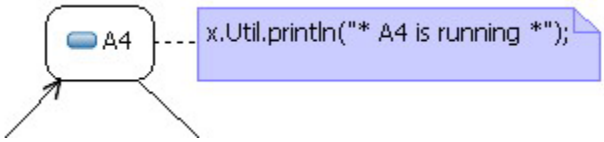
3.3.1. Attaching Java snippets

- Select the element in the diagram or the Model Explorer (RSA 6) / Project Explorer (7)
    - Note that guards are edited by selecting the activity edges or transitions
- Write the java code (or Boolean expression) in the Snippet view

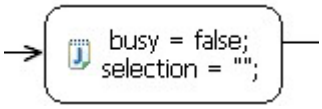


For Actions, there are two more options:

- **Writing the Java code in a comment on a diagram:**
  - Create a Comment.
  - Write the code in the Snippet view.
  - Attach the comment to the action.



- **Writing the Java code as Action name on a diagram:**
  - Create an Action.
  - Write Java code as the Action name.
  - Add a «java» stereotype to the Action.



Default values of class attributes:

- When the attribute is selected, a Java expression can be inserted in the Default Value field of the properties view (General tab).

3.3.2. Guidelines for Writing Java Snippets

- Snippet code can reference the structure of the enclosing class, e.g.:
  - Access attributes.
  - Call operations.
  - Instantiate objects of nested classes.
- Operation code can use its parameters (as in a normal Java method).
- In state machines: the code of effect, entry, do, and exit behaviors, as well as transition guard can access the event that caused the trigger, according to the following rules:
  - If the transition's event type can be unambiguously inferred to be a Signal Event with a particular signal type – the signal is available to the code using the parameter "signal" whose type is the correct signal type (taking into account the signal hierarchy).
  - If the transition's event type can be unambiguously inferred to be a CallEvent with a particular operation – all the operation's parameters are available to the code.
  - On other cases – the event can be accessed as a parameter "event" with the type IEvent. To extract signals from an event, cast it into a SignalEvent and then use the getSignal() method.

- For multi-valued attributes (multiplicity 0..*, *) a typed list (named <Type>List, e.g. TaxiList) is created. The list conforms to interface java.util.List, and adds to it the following type-safe methods:
    - <Type> getValue(int index)
    - void addValue(<Type> element)
    - void addValue(int i, <Type> element)
    - void addValue(<Type> element)
- For attributes whose multiplicity is exactly 1, if no default value is inserted, a default value is automatically generated by the debugger by invoking the default constructor of the referenced class (if such constructor exists). This saves the need to specify most of the instantiations of attributes, and contributes to a smoother designing experience. To override this behavior, specify a null default value (but remember to instantiate the attribute in the constructor) or specify the multiplicity to be 0..1.

3.3.3. The System Library

The UML Debugger provides a library of useful operations which can be called in Java snippets. These are static operations which can be found in a class called MexSystem.

Behavior-related:

- invoke(instance) – invoke the classifier behavior of the given instance.
- send(target instance, signal instance) – send a signal to the target.

Printing to the debugger console:

- println(…) - works with objects, strings, primitives…
- printStackTrace(exception)

UI - related:

- messageDialog(String message) - display a message in an information dialog
- messageDialog(String title, String message) – display a message in an information dialog with the given title
- messageDialog(String title, String message, String button) – display a message in an information dialog with the given title and button text
- String inputDialog(String message) – ask the user for a string input using a window with the given message.
- String inputDialog(String title, String message) – ask the user for a string input, using a window with the given title and message.

Others:

- String getCurrentProjectPath() – returns the full path name of the current project (useful for accessing files in the project).

3.4. The Uml2Debug Profile

MEX provides a UML profile (Uml2Debug) which can be used to add several execution-related and testing-related capabilities to the model (see model "Configuration"):

Execution-related stereotypes:

Stereotype	Applies to:	Owned by:	Semantics
«java»	Comment	Operation	Operation body is specified in Java
		Class	Class-level imports
		Activity Diagram	When connected to an opaque action, the action's body is specified in the comment
		Action (RSx 6)	The action's body is specified in the comment
	Action		The action's body is specified in its name
	Primitive Type		External java classes as primitive types.
«create»	Operation		Constructor operation
«init»	Class		Initial system configuration – an instance of that class will be created at the beginning of each debugging session and each test case.
«classpath»	Comment	Model	The comment contains the external classpath, for linking with existing code.
«external»	Comment	Package	An external java package
«boundary»	Class		System boundaries – classes that are visible to the user and test generator (if such classes are present, others will not be visible).

Note that «java» comments are automatically created when using the Snippet view.

Test-generator – related stereotypes:

Stereotype	Applies to:	Owned by:	Semantics
«values»	Class		The class' attributes instantiate predefined values for parameters
«parameter»	Comment	«values» Class	The comment's text specifies the fully-qualified name of a parameter which will be given the values specified by the attached class.

To apply the debugging profile on a new model:

- Select the model in the Project Explorer. In the Profiles tab of the Properties view, click on Add Profile. Under Deployed Profile, choose Uml2Debug.

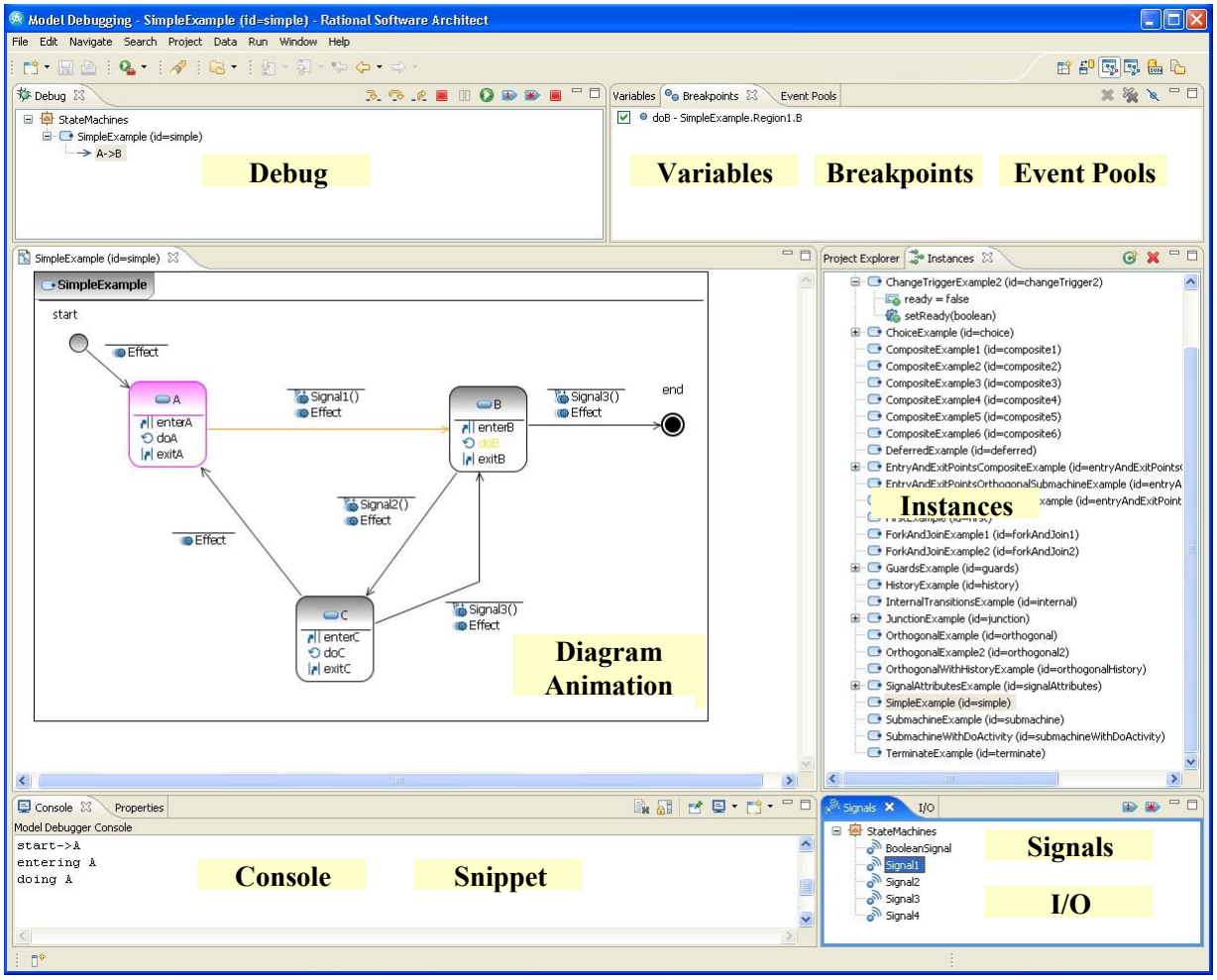
4. Model Debugging

4.1. Starting the Debugger

- Right-click the model. Choose Debug Model.
- The Model Debugging perspective opens.

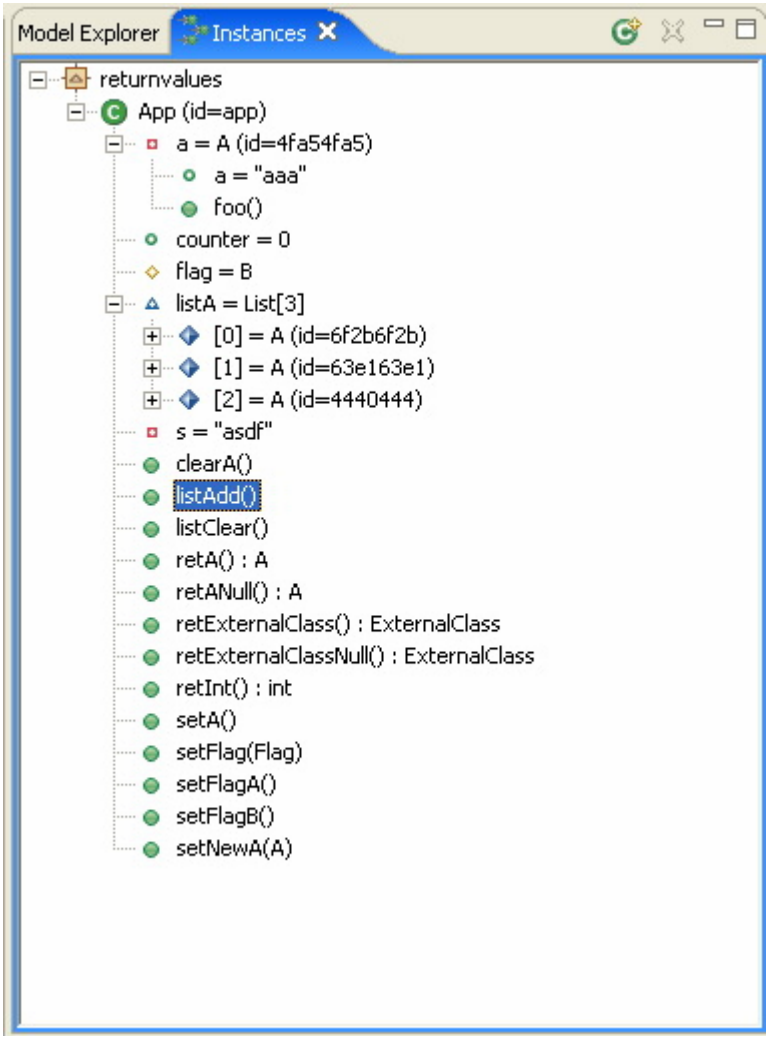
4.2. The Model Debugging Perspective

The Model Debugging perspective resembles the familiar Debug Perspective provided by Eclipse. In addition, it contains several other specialized views.



4.2.1. The Instances View

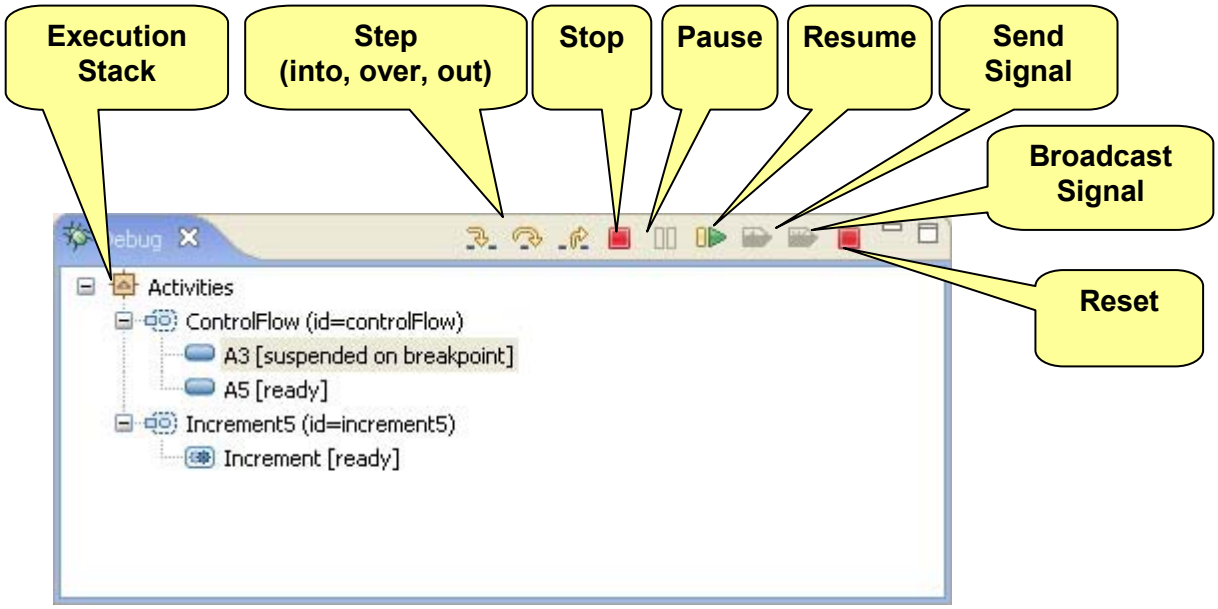
Debugging UML models is often a dynamic exercise. Using the instances view, you can dynamically create and destroy instances, observe the current values of attributes, and invoke the operations and behaviors of instances.



- Use the New Instance button to create a new instance. Each instance should be given an ID to help reference it.
- Use the Destroy Instance button (or the Del keyboard button) to destroy the selected instances. Running classifier behaviors will be cancelled, as well as classifier behaviors of attributes (unless the attributes reference top-level instances in the Instances view).
- Double-click on an operation to invoke it.
- Double-click on an instance to invoke its classifier behavior. If it is an instance of a behavior, the behavior itself will be invoked.
- If the operation or behavior return objects, they will be displayed in a dialog which will allows you to save them back into the Instances view.

4.2.2. The Debug View

This view shows all running behaviors organized according to the call history. It resembles the debug view of provided by Eclipse, but uses model elements as the granularity of control and display. Since UML allows concurrent flows in behaviors, the execution stack is a tree.

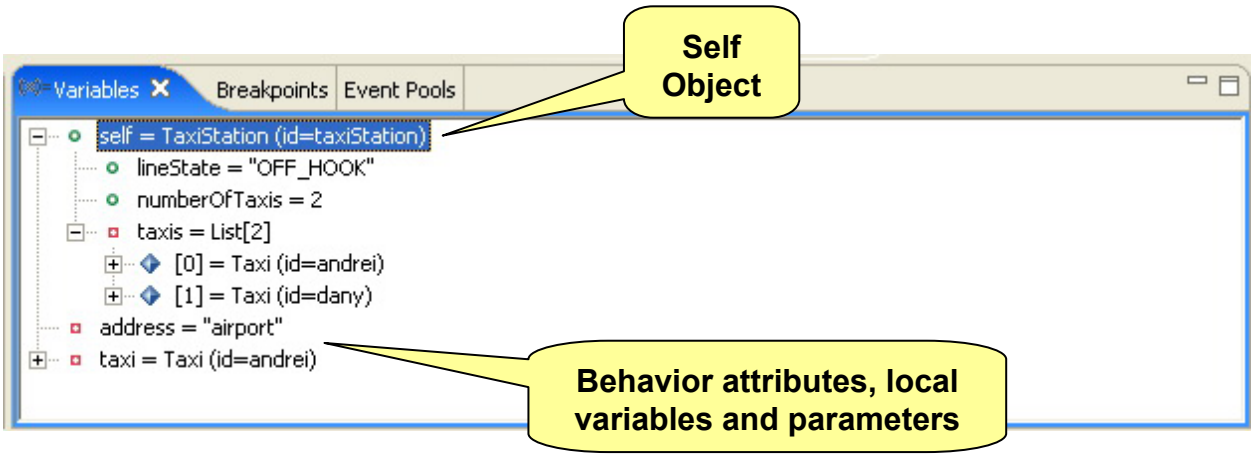


Use the buttons in this view to control the execution process (you can also use the keyboard shortcuts as seen on the Run menu during debug sessions):

- Press the Pause button to go to Step Mode.
- While in Pause mode, you can use the Step buttons to step through behavior execution.
    - Double-clicking an element is equal to clicking Step Over.
- Press the Resume button to go back to Run Mode.
    - Hitting a breakpoint while in this mode will switch the debugger to Step Mode.
- Use the Send Signal button to send signals.
    - A dialog will open which will allow you to choose the signal and the target instance.
    - The selected instance from the Debug view will be selected, but you can select another target.
    - If the signal has attributes, the dialog will also allow editing them before sending.
- Use the Broadcast Signal button to send signals to all running instances.
    - A dialog will open which will allow you to choose the signal.
    - If the signal has attributes, the dialog will also allow editing them before broadcasting.
- Use the Stop button to stop the debug session.
- In case of errors, use the Terminate button to terminate the debug session.

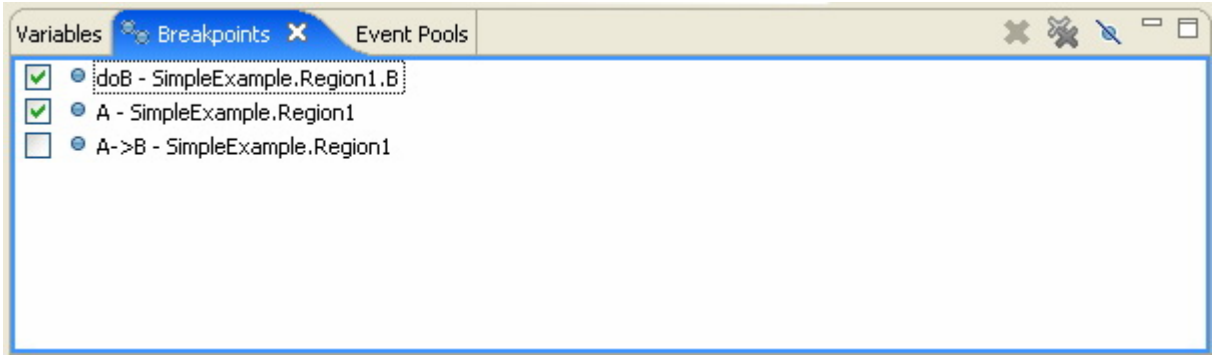
4.2.3. The Variables View

This view resembles the variables view provided by Eclipse. It allows observing the values of the attributes and variables available in the current context (as selected in the Variables View).



4.2.4. The Breakpoints View

This view lists the currently defined breakpoints and allows deleting, enabling and disabling them.
Breakpoints can be easily added using the popup menus of model elements in diagrams and the RSx Project Explorer. Currently, breakpoints are only displayed in diagram animation windows.

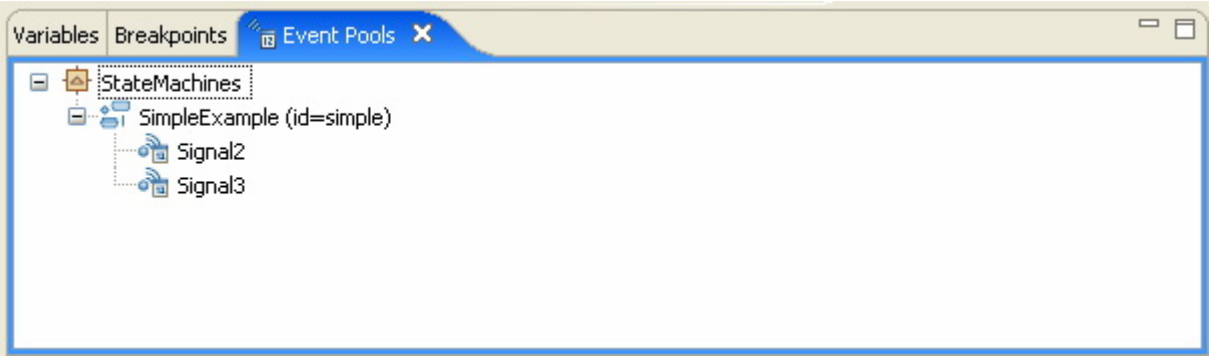


Breakpoints are supported on:

- Activity nodes:
  - Actions
  - Initial and Final Nodes
- State machine nodes:
  - Entry / Exit / Do/ Effect Behaviors
  - States
  - Transitions

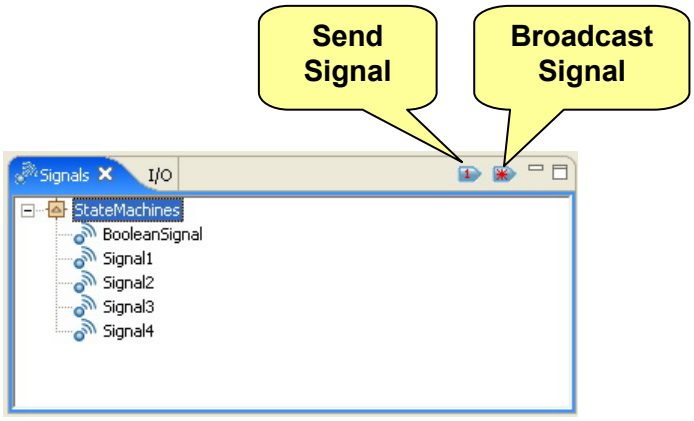
4.2.5. The Event Pools View

This view shows pending events in the event pools of active objects.



4.2.6. The Signals View

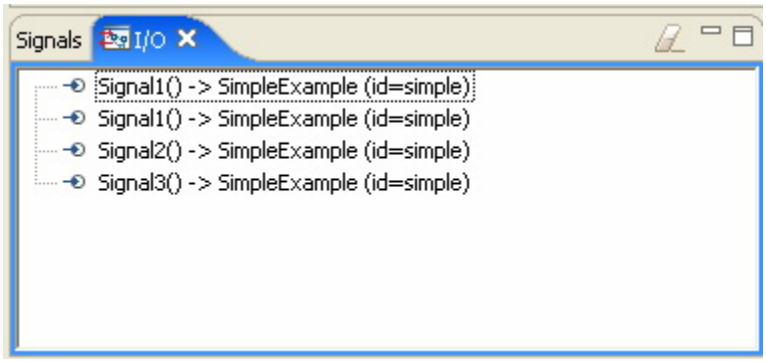
This view provides convenient ways to send signals to running behaviors.



- Select a signal from the list.
- Double-click the signal, or use the Send Signal button to send the signal to the currently displayed behavior (or, if there is more than one visible behavior, the one most recently brought to top).
- Use the Broadcast Signal button to send signals to all running instances.
- If the signal has attributes, a dialog window will be displayed to allow editing them before sending / broadcasting.

4.2.7. The I/O View

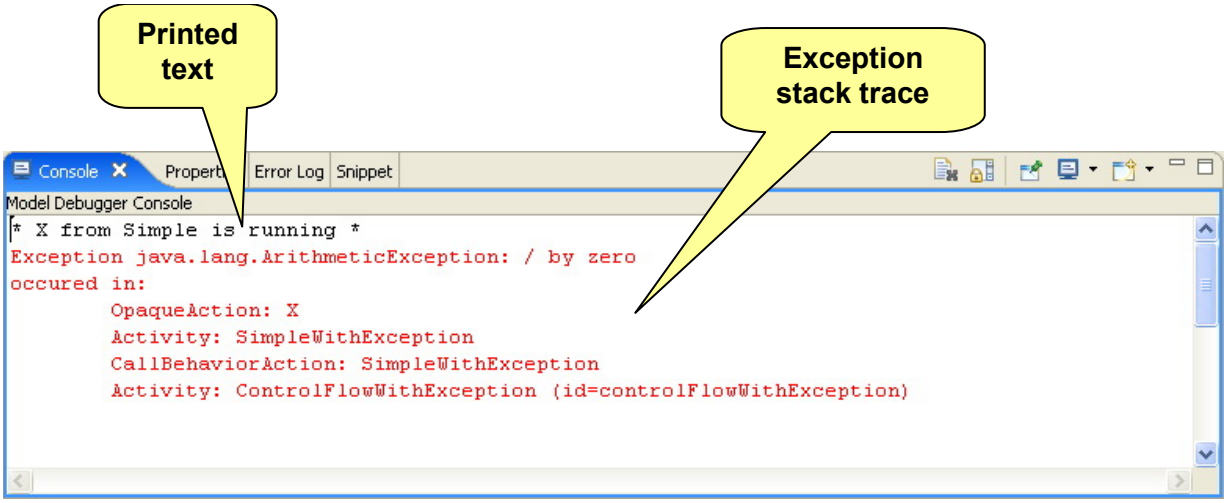
This view shows the history of signals sent to and from the running instances.



4.2.8. The Console View

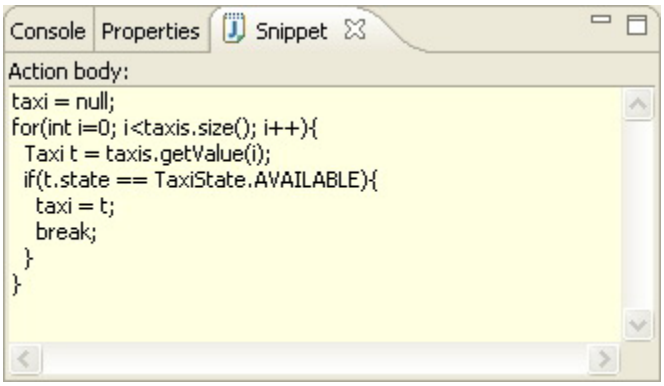
This view is used to display:

- Compilation errors for snippet code.
- Text printed by snippets
- Unhandled exceptions thrown by snippets



4.2.9. The Snippet View

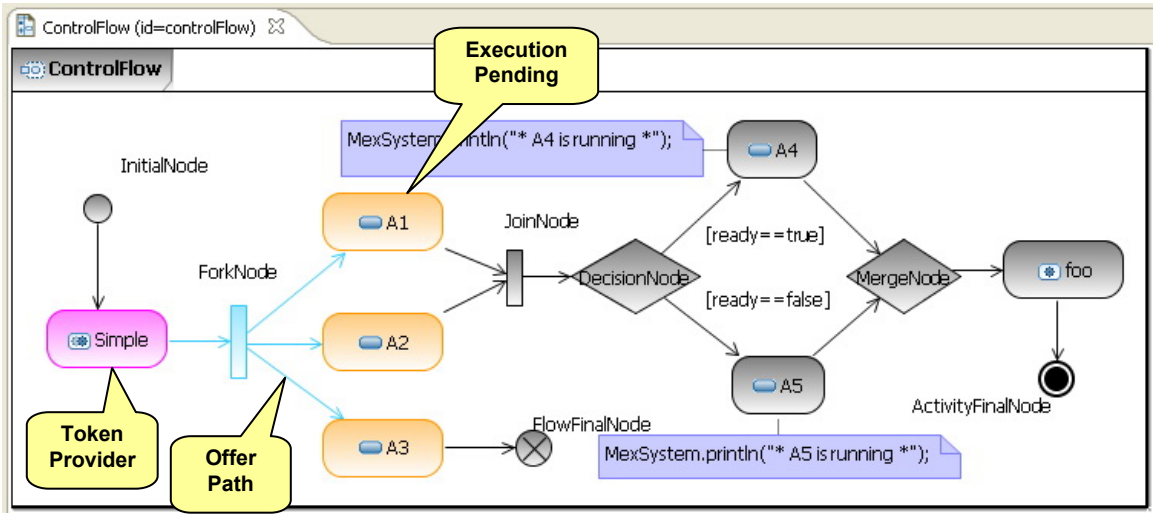
In addition to editing Java snippets (as described previously), during debug sessions this view allows viewing the snippets for the element currently selected in the diagram animation, debug view or instances view (the last selection will be shown). In this mode, the view will not be editable, and its background will be colored, as is shown in the following figure:



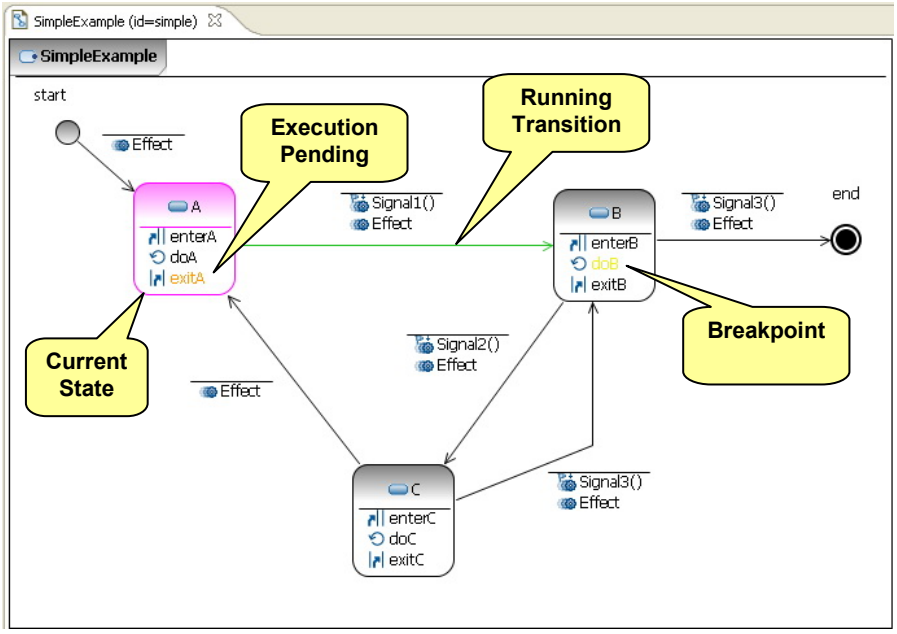
4.2.10. Diagram Animation

This powerful feature provides a visual representation of the executing behavior.

When running an activity, the user can see which nodes are ready for execution (orange), which edges pass token offers (blue), or which node provides the token (magenta).



When running a state machine, the user can see the current state (magenta), which elements are ready for execution (orange), and running elements (green).



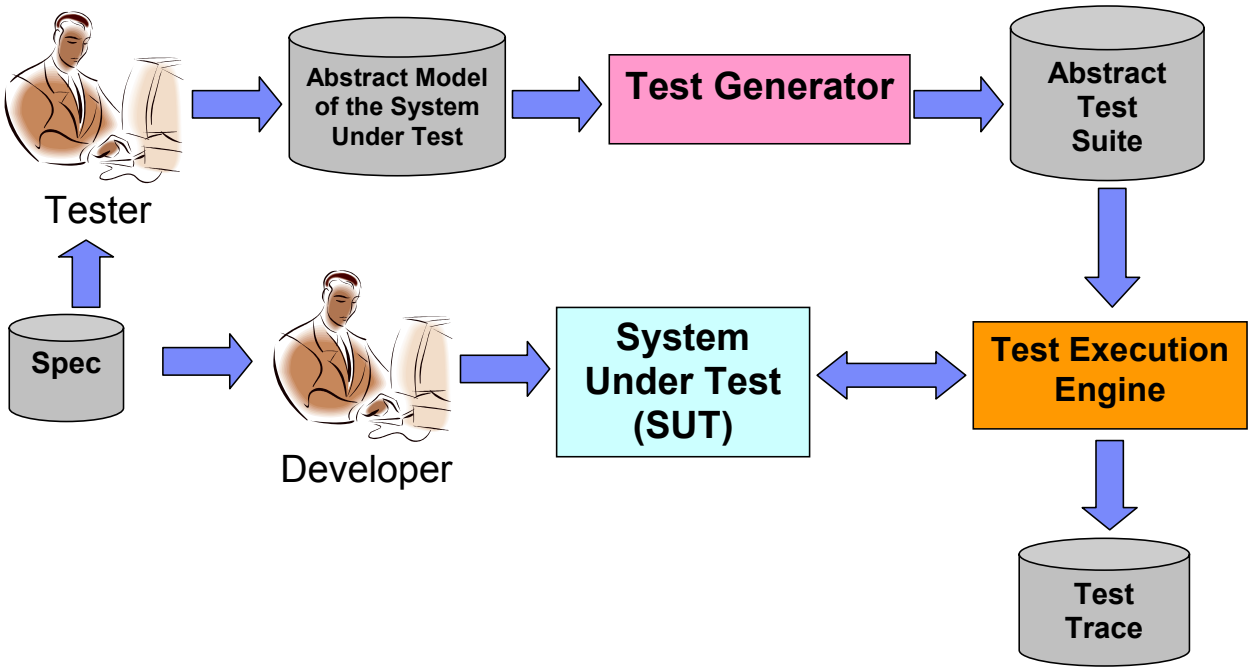
Note: In RSA 7, you might experience performance issues during step-mode debugging if gradient-style appearance is enabled (as is in the above figures). To turn off this feature:

- Open the Preferences window (Window -> Preferences)
- Go to Modeling -> Diagrams -> Appearance.
- Under Global Appearance, uncheck “Show gradient”.

5. Model Driven Testing

Briefly, the model driven testing process looks as follows:

- Create an abstract model of your application. Usually you will create multiple models. Each model will describe part of your application, an aspect/feature you want to test. You omit implementation/platform/concrete details of your application under test from the model - that's why we call it abstract model.
- Generate test suites from this model using different test generation algorithms.
- Execute the generated test suites on your application (the System Under Test – SUT).



5.1. Current capabilities

The test generator generates test suites that cover different inputs to the SUT. By executing the abstract model, it can predict the expected outputs and insert assertions to the test suites.

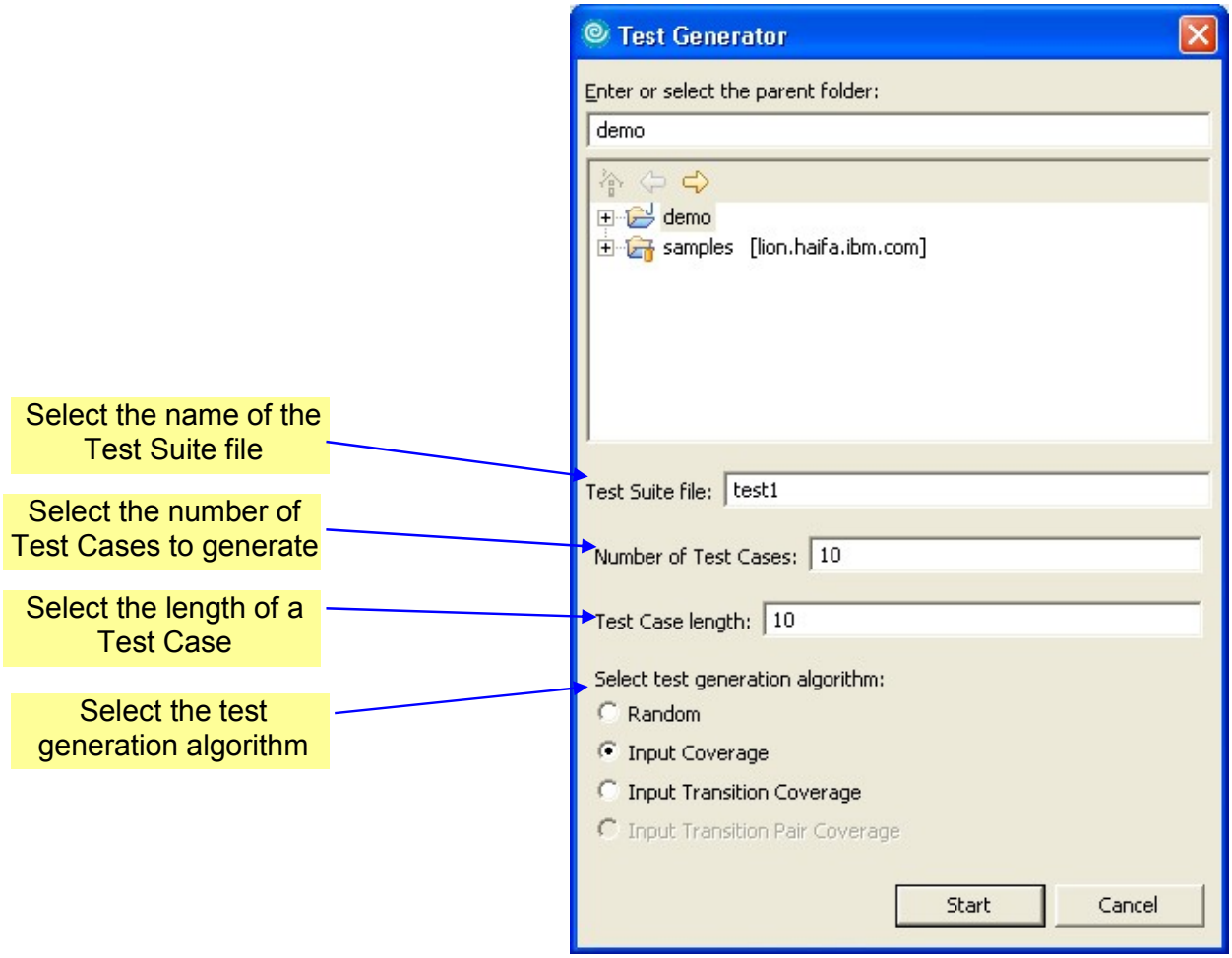
An input to the SUT is defined as the tuple: <class, operation, parameter values>.

Output is defined as the tuple: <class, attribute value>.

5.2. How to generate tests

We use the Calculator model to demonstrate this:

- Right-click model, Generate Test Suite
- Type the file name (without extension – it will be appended automatically) and select the algorithm
- Switch to Resource perspective to see the generated test suite



5.3. How to run the generated test suite

- open Spider (Test Execution Engine) view: Window, Show View, Other..., Test Execution, Spider
- right-click a test suite (.ats file), Execute Test.

5.4. Test Execution Directives

ATS is an input to the Test Execution Engine telling it **what** to execute against the System Under Test (SUT). Test Execution Directives (TED) is another input to the Test Execution Engine providing the information on **how** to execute the ATS.

It includes:

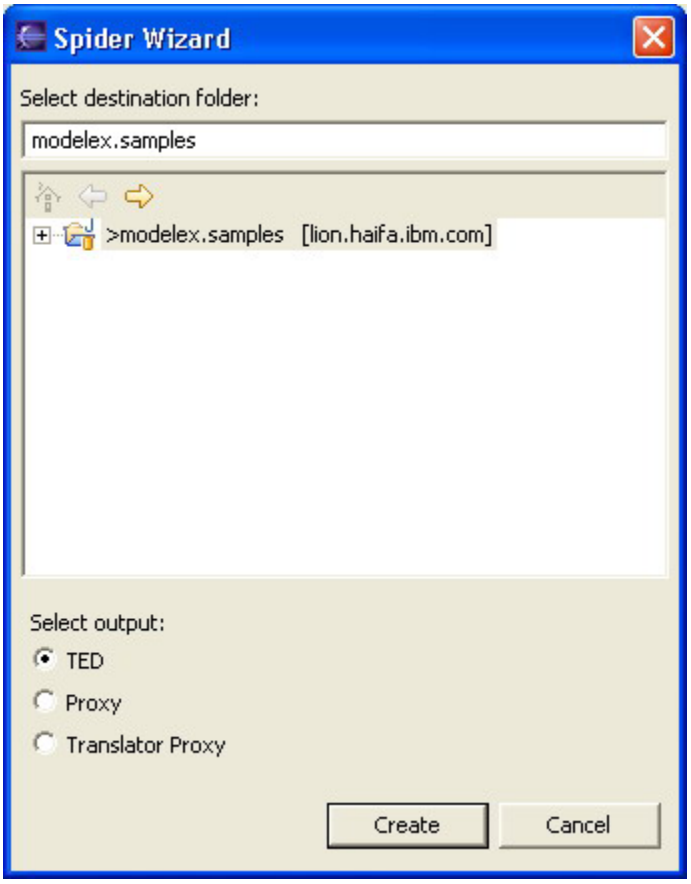
- constants – global constants for substitute throughout the test execution directives
- definitions – description of SUT (host machines, processes, objects and classes) for distribution, synchronization and multiplication (cloning)
- initialization – initialization directives for all objects, processes, and hosts
- invariants – repetitive actions to extend the ATS (setup, cleanup, additional evaluations)
- options – various global execution options
- mappings – ATS to SUT mapping directives for types, constants, controls and observables.

So you have two options:

- execute the generated test “as is”
- supply the execution directives

To create a template of the TED file from the generated ATS file:

- switch to the Resource perspective
- right-click the generated test (ATS file)
- select Spider Wizard
- select the destination folder and choose Select output: TED
- press Create button
- the TED file will have the same name as the ATS file



Edit the TED file and run the Execution Engine with the directives:

- Right-click a directives file (.ted), click Execute Test.

5.5. Test Object Proxies

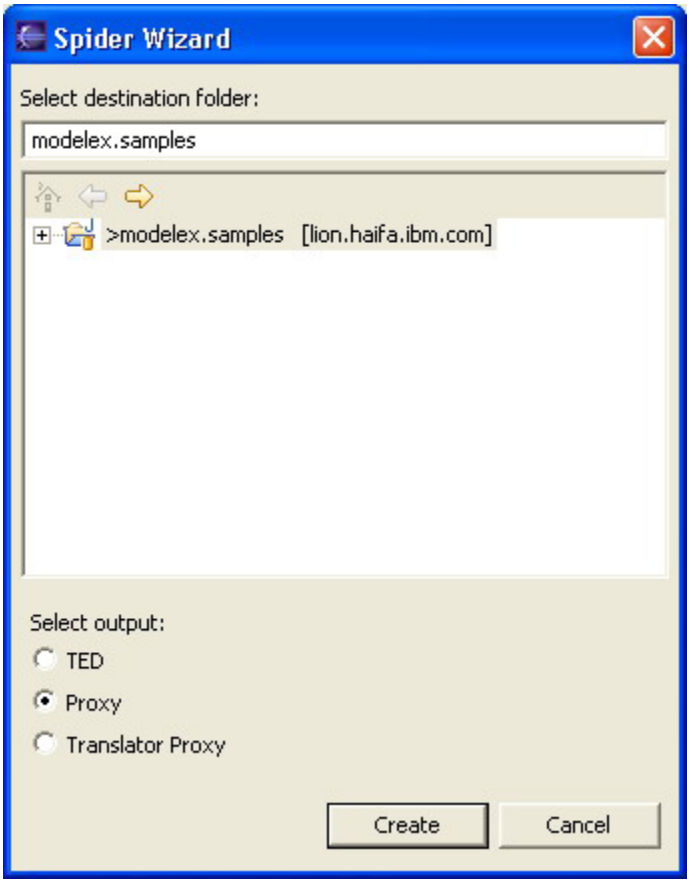
The test executes directly on the System Under Test (SUT) – test objects or Java objects. But in many cases there is a gap between the test abstraction and the SUT.

To bridge this gap you can create test object proxies. In this case the Test Execution Engine will interact with the proxies as with the SUT objects. The proxies are responsible for the interaction with the SUT: transfer the method calls to the SUT, and get the SUT's state.

For example: If the method openFile() returns a handle, and the method readFile() gets this handle as a parameter, then the handle may be stored in the proxy.

To create proxy templates from the ATS or TED file:

- switch to the Resource perspective
- right-click the generated test (ATS file) or the TED file
- select Spider Wizard
- select the destination folder and choose Select output: Proxy
- Press the Create button.



5.6. Translation of tests

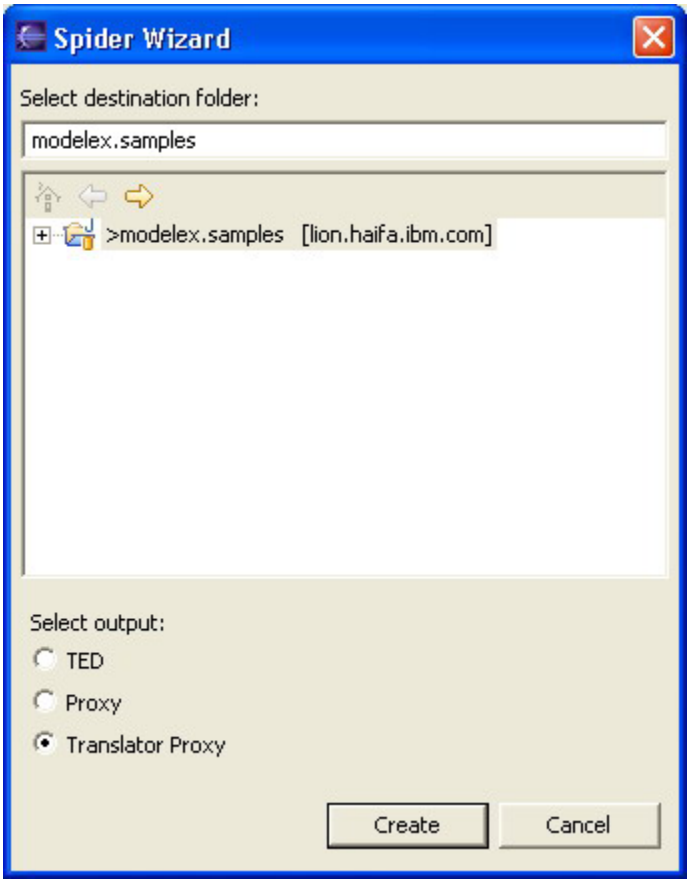
Usually the Test Execution Engine executes the test directly on the SUT (or via proxies). But it is possible to translate the test to a text file to be used by other test execution harness.

To do this:

- create translation proxies
- edit the proxies to output the desired text file
- run the test translator

To create translation proxy templates from the generated ATS file:

- switch to the Resource perspective
- right-click the generated test (ATS file)
- select Spider Wizard
- select the destination folder and choose Select output: Translator Proxy
- press Create button



To run the translator:

- switch to the Resource perspective
- right-click the generated test (ATS file)
- select Translate ATS

6. Modeling Voice Dialogs

Model execution can be extended by plug-ins to support the semantics of UML profiles. For example, this release supports the profile for Voice applications, as defined by OMG. It also demonstrates several transformations from UML with Voice Profile to prominent standards for defining voice applications: X+V and SCXML.

6.1. The Voice Dialogs Profile

The Uml2Voice profile contains the following stereotypes:

Stereotype	Applies to:	Semantics
«Dialog»	Activity	Unchanged
«SubDialogState»	CallBehaviorAction which calls a «Dialog»	Unchanged
«Message»	Operation whose return type is String	The string returned by the operation will be played by «Play» actions.
«Play»	CallOperationAction which references a «Message» operation	Play the string returned by the operation
«InputEvent»	Signal	Unchanged
«WaitState»	Action	Wait for a signal
«Trigger»	Action, immediately downstream from a «WaitState» action	Accept the signal specified in its name. Signal attributes can be assigned to activity attributes using the following format for the trigger action name: SignalName(attr_1_name, attr_2_name…).

To Model voice dialogs apply the Uml2Voice profile on a model:

- Select the model in the Project Explorer. In the Profiles tab of the Properties view, click on Add Profile. Under Deployed Profile, choose Uml2Voice.

6.2. The UML2 Voice

If needed, import the UML2 Voice library, which contains the definition of input events such as DTMF tones:

- Right-click on the model and click on Import Model Library. Under System Library, choose UML2 Voice.

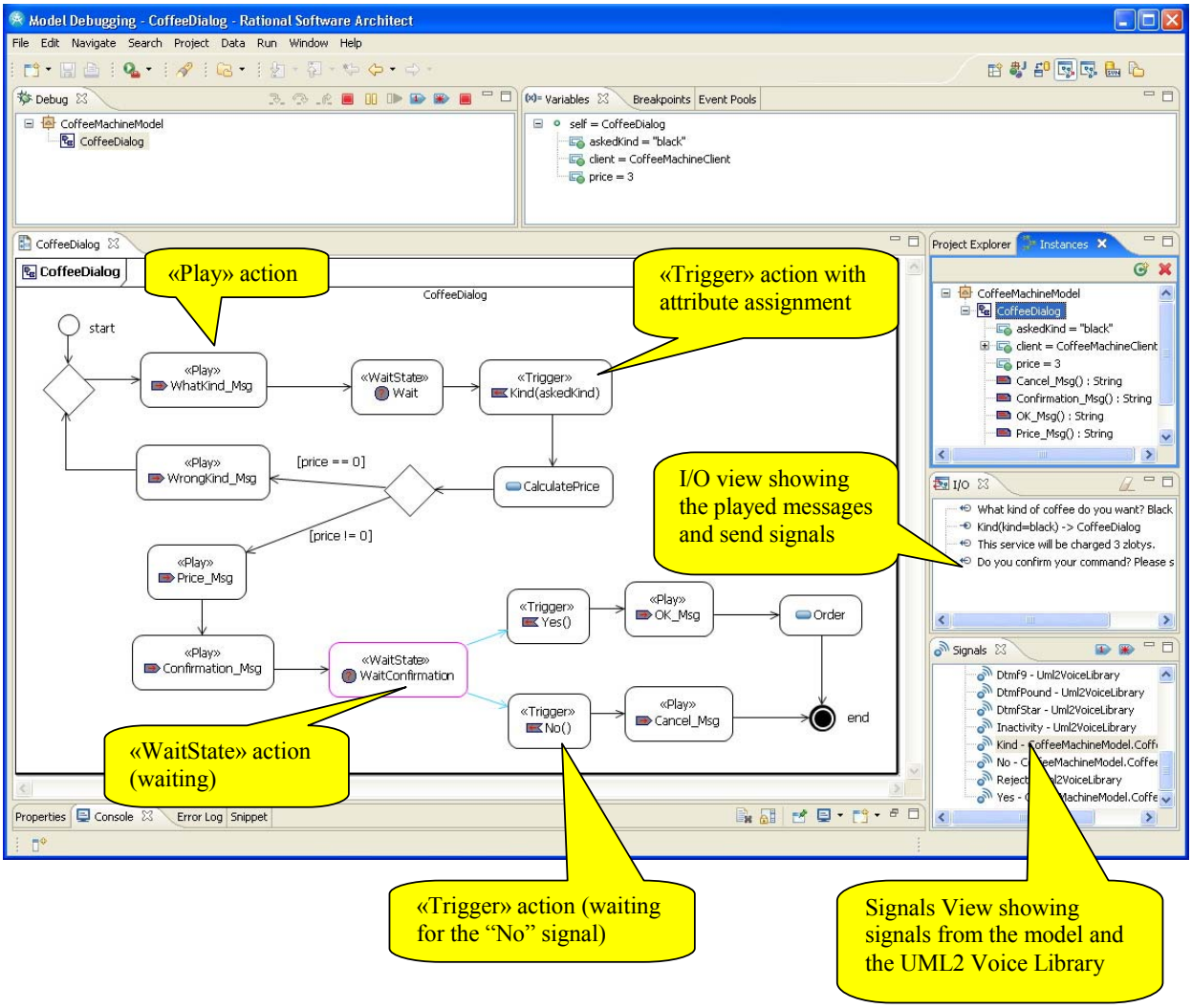
6.3. Debugging Voice Dialog Models

We have extended the Model Debugger with several capabilities specific for voice-dialog models:

- After executing a WaitState, the debugger waits for a signal. The WaitState action passes tokens to all Trigger actions, but they do not take the token until receiving the signal.
- Trigger actions only take the tokens offered to them after the signal defined in their name is received. When this happens, the token offers to the other waiting triggers are cancelled. Thus, the path downstream from the accepting trigger action is chosen. Trigger action can also assign signal attributes to activity attributes as specified in the trigger name.
- Messages played by Play actions are displayed as an output line in the I/O view.

Known limitations:

- The signal hierarchy is not supported by Trigger actions.



6.4. Transformation into domain-specific code:

To transform the UML activities into a format that can be executed by voice application engines:

- Right-click on the model
- Choose Transform
- Choose the target language of your choice (X+V servlets, StatechartXML, text)
- If requested, choose the transformation target directory

7. Samples

To install the samples import the ZIP file (mex.samples.v3.YYYYMMDD.zip) using project interchange wizard: File, Import, select Project Interchange, press Next, Browse the ZIP file, select all the projects and press Finish.

The project **mex.samples** contains several examples of concrete systems:

- **IntersectionSystemModel** – An analysis-phase model of an intersection. Structure is specified in composite structure diagrams. Use cases are captured in activity diagrams and sequence diagrams (not supported).
- **IntersectionDesignModel** – A design-phase model of an intersection. Extends the system model by inheriting from it and adding behaviors specified in state machines.
- **MeetingRoom** – A meeting room with a light-bulb that is turned on only when the room is used. Demonstrates operations implemented by activities.
- **Taxi** – A taxi station. Demonstrates operations implemented by activities.
- **TrafficExample** – Another example of an intersection. Demonstrates state machines as classifier behaviors, as well as usage of external Java code (for UI).
- **TransmitterDesignModel** - An analysis-phase model of a sender-receiver system. Structure is specified in composite structure diagrams.
- **TransmitterSystemModel** - A design-phase model of a sender-receiver system. Extends the system model by inheriting from it and adding behaviors specified in state machines.
- **Windeich** – An example GUI application modeling (activity represents a dialog).

The project **mex.features** contains simple example which demonstrate the different UML features supported by the model debugger:

- **Activities** – all supported activity elements.
- **StateMachines** – all supported state machine elements
- **Precondition** – operation pre-condition
- **Calculator** – a sample which can be used for the Test Generator.
- **Configuration** – the stereotypes «boundary», «classpath», «parameter», «values».
- **Dispatcher** –data flow, and operation and activity parameters
- **Loop** – an implementation of aloop using activity
- **Parameters** – operation and activity parameters

The project **mex.voice** contains examples of voice application:

- **Bruxelles** – A voice application which specifies a general-knowledge quiz (was used in MODELWARE demo in Bruxelles…)
- **CitiesList –** A voice application which allows maintaining a list of cities.
- **CoffeeMachine** – A voice interface for a simple coffee machine.
- **PhoneBook** – A voice application which allows maintaining a persistent phone book.
- **TvRecorder** – A voice interface for a TV recorder.
- **Weather** – A voice-based weather forecast answering service.
- **VoiceDialog** – Demonstrates miscellaneous features of voice dialogs.
- **XvTest** – Demonstrates miscellaneous features of voice dialogs.