



CAMEO SIMULATION TOOLKIT

version 1.0
user guide

No Magic, Inc.
2011

All material contained herein is considered proprietary information owned by No Magic, Inc. and is not to be shared, copied, or reproduced by any means. All information copyright 2010-2011 by No Magic, Inc. All Rights Reserved.

CONTENTS

| | |
|--|-----------|
| CAMEO SIMULATION TOOLKIT | 5 |
| 1. Getting Started | 5 |
| 1.1 Introduction to Cameo Simulation Toolkit | 5 |
| 1.2 Key Features | 5 |
| 1.3 Installation | 6 |
| 2. Model Execution | 6 |
| 2.1 Simulation by Executing Elements | 7 |
| 2.1.1 Behaviors | 10 |
| 2.1.2 Class | 14 |
| 2.1.3 Diagram | 17 |
| 2.1.4 Instance Specification | 17 |
| 2.2 Simulation by Executing the Execution Configuration | 18 |
| 3. Execution Configuration | 20 |
| 3.1 ExecutionConfig Stereotype | 20 |
| 3.2 Execution Log | 20 |
| 3.3 User Interface Prototyping | 21 |
| 3.4 UI Modeling Diagram Execution | 22 |
| 3.5 ActiveImage and ImageSwitcher | 25 |
| 4. Animation | 26 |
| 4.1 Active and Visited Elements | 26 |
| 4.2 Customizing Animation Colors | 27 |
| 5. Simulation Debugging | 28 |
| 5.1 Understanding Simulation Sessions | 28 |
| 5.2 Simulation Debugger | 29 |
| 5.3 Simulation Console | 30 |
| 5.3.1 Console Pane | 30 |
| 5.3.2 Simulation Information | 31 |
| 5.3.3 Simulation Log File | 32 |
| 5.4 Runtime Values Monitoring | 32 |
| 5.4.1 Variables Pane | 32 |
| 5.4.2 Runtime Object created from InstanceSpecification | 33 |
| 5.4.3 Exporting Runtime Objects to InstanceSpecification | 34 |
| 5.5 Breakpoints | 36 |
| 5.5.1 Adding Breakpoints | 37 |
| 5.5.2 Removing Breakpoints | 38 |
| 6. Validation and Verification | 40 |
| 7. State Machine Simulation | 42 |
| 7.1 Supported Elements | 42 |
| 7.2 Adapting Models for State Machine Simulation | 43 |
| 7.2.1 Defining Trigger on Transition | 43 |
| 7.2.2 Using Guard on Transition | 44 |
| 7.2.3 Behaviors on Entry, Exit, and Do Activity of State | 45 |
| 7.3 Running State Machine Execution | 45 |
| 7.4 Sample Projects | 46 |
| 7.4.1 test_regions.mdzip | 46 |
| 7.4.2 test_timers.mdzip | 46 |
| 7.4.3 test_guard.mdzip | 47 |
| 8. Activity Simulation | 47 |
| 8.1 About Activity Execution Engine | 47 |
| 8.2 Creating Model for Activity Execution | 48 |
| 8.3 Executing Activity | 71 |

CONTENTS

- 9. Parametrics Simulation 76**
 - 9.1 About Parametrics Engine 76
 - 9.2 Adapting Model for Parametric Execution 77
 - 9.2.1 Understanding the Flow of Parametric Execution 77
 - 9.2.2 Typing Value Properties by Boolean, Integer, Real, Complex, or Their Subtypes 78
 - 9.2.3 Using Binding Connectors 79
 - 9.2.4 Creating InstanceSpecification with Initial Values 80
 - 9.2.5 Working with Multiple Values 81
 - 9.3 Running Parametric Simulation 82
 - 9.4 Retrieving Simulated Values 85
 - 9.5 Executing Parametric Simulation from Activity 86
 - 9.6 Sample Projects 87
- 10. Interaction Between Engines 87**
 - 10.1 Stopwatch Sample 87
 - 10.1.1 Manual Execution 87
 - 10.1.2 Controlling Execution with Activity Diagram 88
- 11. Mathematical Engine 88**
 - 11.1 Math Console 88
 - 11.2 Exchanging Values between Cameo Simulation Toolkit and Mathematical Engine 90
 - 11.2.1 Exchanging values between Slot and Mathematic Environment 90
 - 11.2.2 Export Runtime Value to Mathematical Engine 92
 - 11.3 Built-in Math Solver 92
 - 11.3.1 Using Built-in Math Solver in Math Console 92
 - 11.3.2 Variables 93
 - 11.3.3 Values 93
 - 11.3.4 Constants 95
 - 11.3.5 Operators 95
 - 11.3.6 Functions 97
 - 11.3.7 Built-in Math Solver API for User-Defined Functions 102
 - 11.4 Using MATLAB[®]¹ as a Mathematical Solver 104
 - 11.4.1 Setting up system for calling MATLAB[®] from Cameo Simulation Toolkit 104
 - 11.4.2 Selecting MATLAB[®] as Mathematical Solver for Cameo Simulation Toolkit 107
- 12. Action Languages 108**

1. MATLAB[®] is a registered trademark of The MathWorks, Inc.

CAMEO SIMULATION TOOLKIT

1. Getting Started

Cameo Simulation Toolkit is a MagicDraw plugin which provides a unique set of tools supporting the standardized construction, verification, and execution of computationally complete models based on a foundational subset of UML.

No Magic is the first in the industry to provide customers with an easy-to-use, standard-based executable UML solution that integrates the semantics of different UML behaviors.

1.1 Introduction to Cameo Simulation Toolkit

The purpose of simulation is to understand the function or performance of a system without manipulating it, either because the real system has not been completely defined or available, or because it cannot be experimented due to cost, time, resources, or any other risk constraints. A simulation is typically performed on a model of a system.

With Cameo Simulation Toolkit, you can execute a model and validate the functionality or performance of a system in the context of a realistic mock-up of the intended user interface. The solution allows you to predict how the system responds to user interaction or predefined test data and execution scenarios.

Cameo Simulation Toolkit contains the Simulation Framework plugin that provides the basic GUI to manage the runtime of any kind of executable models and integrations with any simulation engines. The main functionalities of Cameo Simulation Toolkit are as follows:

- (i) Simulation Window:
 - Toolbars and Debugger Pane: to control the execution/simulation
 - Simulation Console: to execute log outputs and command line for active engine
 - Sessions Pane: to select the interested session of execution
 - Variables Pane: to monitor the runtime values of each execution session
 - Math Console: to communicate with Mathematical engine
 - Breakpoints pane
 - Triggers option
- (ii) Pluggable execution engines
- (iii) Execution animation
- (iv) Model debugger
- (v) Pluggable events and data sources
- (vi) Pluggable mockup panels
- (vii) Model-driven execution configurations
- (viii) Pluggable expression evaluators and action languages

1.2 Key Features

Cameo Simulation Toolkit is capable of executing your UML or SysML models. The key features of Cameo Simulation Toolkit are as follows:

- (i) **Simulation Framework:** general infrastructure (simulation toolbars, simulation context menu, simulation panes, etc.) and Open API for execution.

- (ii) **State Machine execution engine:** W3C SCXML (State Charts XML) standard, an open-source Apache implementation.
- (iii) **Activities execution engine:** OMG fUML (foundational subset of Executable UML) standard.
- (iv) **Parametrics execution engine:** allows Cameo Simulation Toolkit to execute SysML Parametric diagrams. SysML Plugin for MagicDraw is required for the engine to work properly.

The simulation sample projects are available in the `<md.install.dir>/samples/simulation` directory.

1.3 Installation

To install Cameo Simulation Toolkit, either (i) use Resource/Plugin Manager in MagicDraw to download and install the plugin, or (ii) follow the manual installation instructions if you have already downloaded the plugin.

(i) To install Cameo Simulation Toolkit using Resource/Plugin Manager:

1. Click **Help > Resource/Plugin Manager** on the MagicDraw main menu. The **Resource/Plugin Manager** will appear and prompt you to check for available updates and new resources. Click **Check for Updates > Check**.

| | |
|-------------|--|
| NOTE | Specify HTTP Proxy Settings for connection to start MagicDraw updates and resources. |
|-------------|--|

2. Under the **Plugins (commercial)** group, select the **Cameo Simulation Toolkit** check box and click **Download/Install**.
3. Restart the MagicDraw application.

(ii) To install Cameo Simulation Toolkit following the manual installation instructions on all platforms:

1. Download the **Cameo_Simulation_Toolkit_<version number>.zip** file.
2. Exit the MagicDraw application currently running.
3. Extract the content of the **Cameo_Simulation_Toolkit_<version number>.zip** file to the directory where your MagicDraw is installed, `<md.install.dir>`.
4. Restart the MagicDraw application.

2. Model Execution

Cameo Simulation Toolkit allows you to execute the elements in a MagicDraw project. The elements that can be executed must be supported by the execution engines in Cameo Simulation Toolkit. Any number of execution engines can be implemented, as separate plugins, and registered to Simulation Framework as the engines for some particular types of model.

Table 1 -- Current Supported Execution Engines

| Execution Engine | Support Elements |
|---------------------------------------|---|
| Activity Execution Engine | <ul style="list-style-type: none"> • Activity • Activity Diagram • Class whose classifier behavior is an Activity • InstanceSpecification of a Class whose classifier behavior is an Activity |
| State Machine Execution Engine | <ul style="list-style-type: none"> • State Machine • State Machine Diagram • Class whose classifier behavior is a State Machine • InstanceSpecification of a Class whose classifier behavior is a State Machine |
| Parametrics Executuion Enigne | <ul style="list-style-type: none"> • Block that contains Constraint Properties • SysML Parametric Diagram • InstanceSpecification of a Block that contains Constraint Properties |
| Interaction Execution Engine | <ul style="list-style-type: none"> • Sequence Diagram • For more detail, see the Cameo Simulation Toolkit API UserGuide.pdf in the <code><md.install.dir>/manual</code> directory |

You can create a simulation by either (2.1) executing the elements that are supported by the execution engines, or (2.2) creating the execution configuration, set the element to be executed as the execution target of the execution configuration, and then execute the model from the execution configuration.

2.1 Simulation by Executing Elements

Cameo Simulation Toolkit allows you to execute a model through a context menu. You can open the menu by right-clicking the element that you would like to execute.

To execute a model through a context menu:

1. Right-click an element either (i) on the diagram (Figure 1) or (ii) in the containment browser (Figure 2), and then select **Simulation > Execute**.

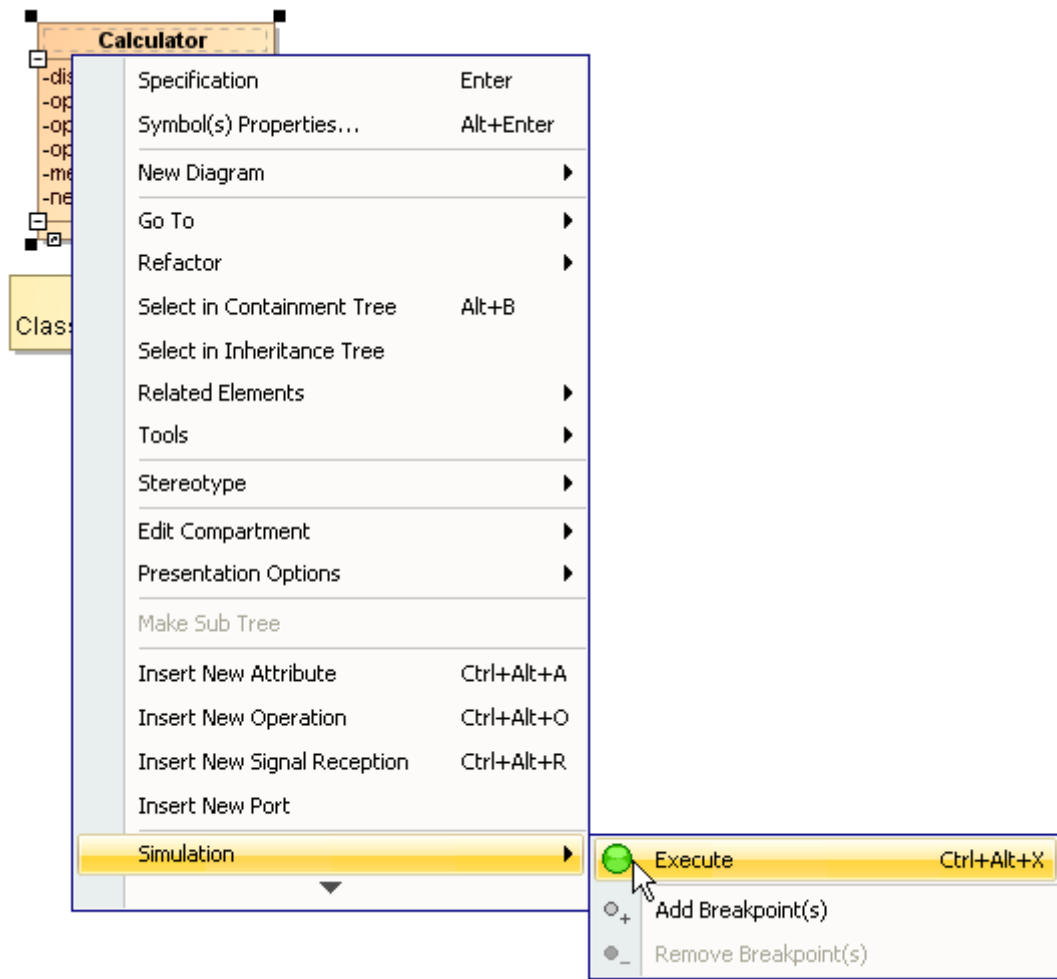


Figure 1 -- Executing Model through Executable Element Context Menu on Diagram

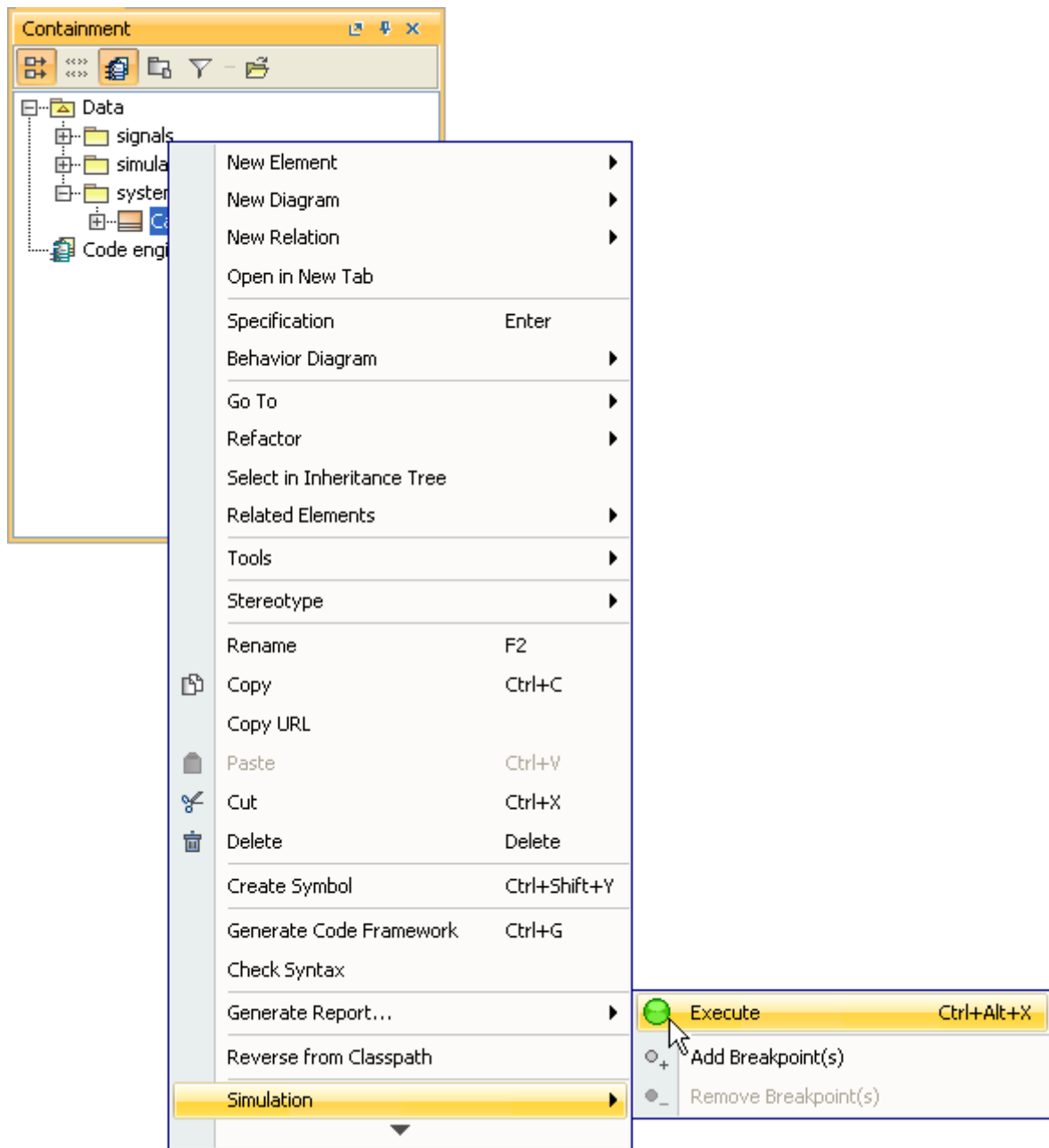


Figure 2 -- Executing Model through Browser Context Menu

2. The **Simulation Window** will open. The Simulation session will automatically be started and displayed in the **Sessions Pane** (Figure 3). The session corresponds to the selected element in the active diagram.

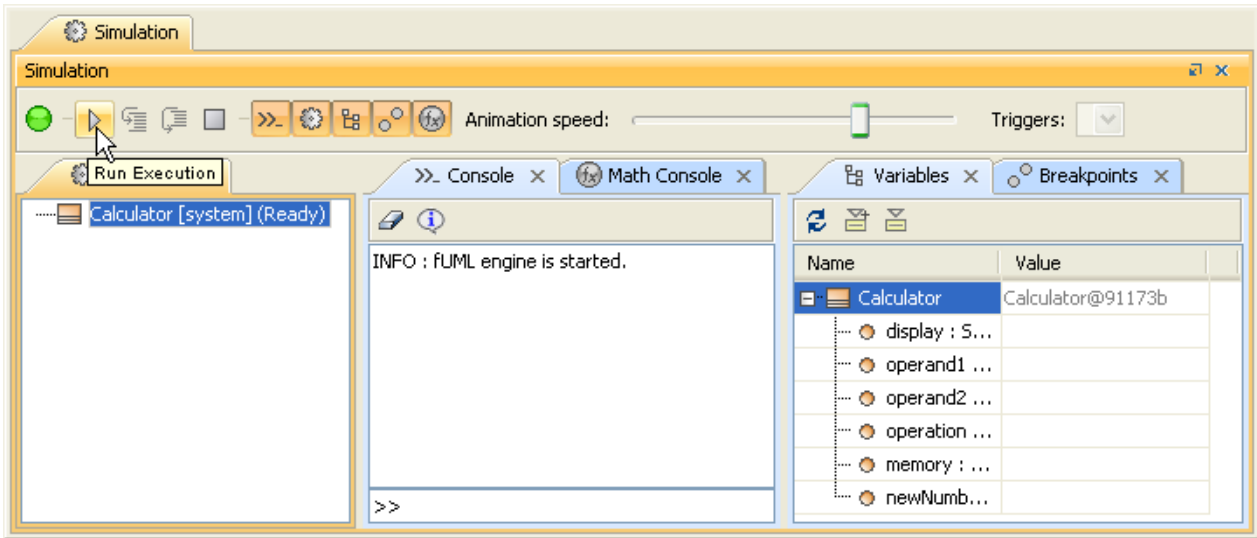



Figure 3 -- Run Execution Button in the Simulation Window

3. Click the **Run Execution** button  on the toolbar (Figure 3) to execute the model.

| | |
|-------------|--|
| NOTE | Cameo Simulation Toolkit will use different execution engines to execute different kinds of elements: <ul style="list-style-type: none">• Behaviors• Class• Diagram• Instance Specification |
|-------------|--|

2.1.1 Behaviors

You can select a behavior, either (2.1.1.1) Activity or (2.1.1.2) State Machine, and execute it.

2.1.1.1 Activity

If you select to execute a behavior, which is an Activity (Figure 4), Cameo Simulation Toolkit will execute it on the Activity diagram whose context is the selected Activity (Figure 5). A new session (Activity) will open in the **Sessions Pane**. If you click the session, the runtime object of the selected Activity will open in the **Variables Pane**.

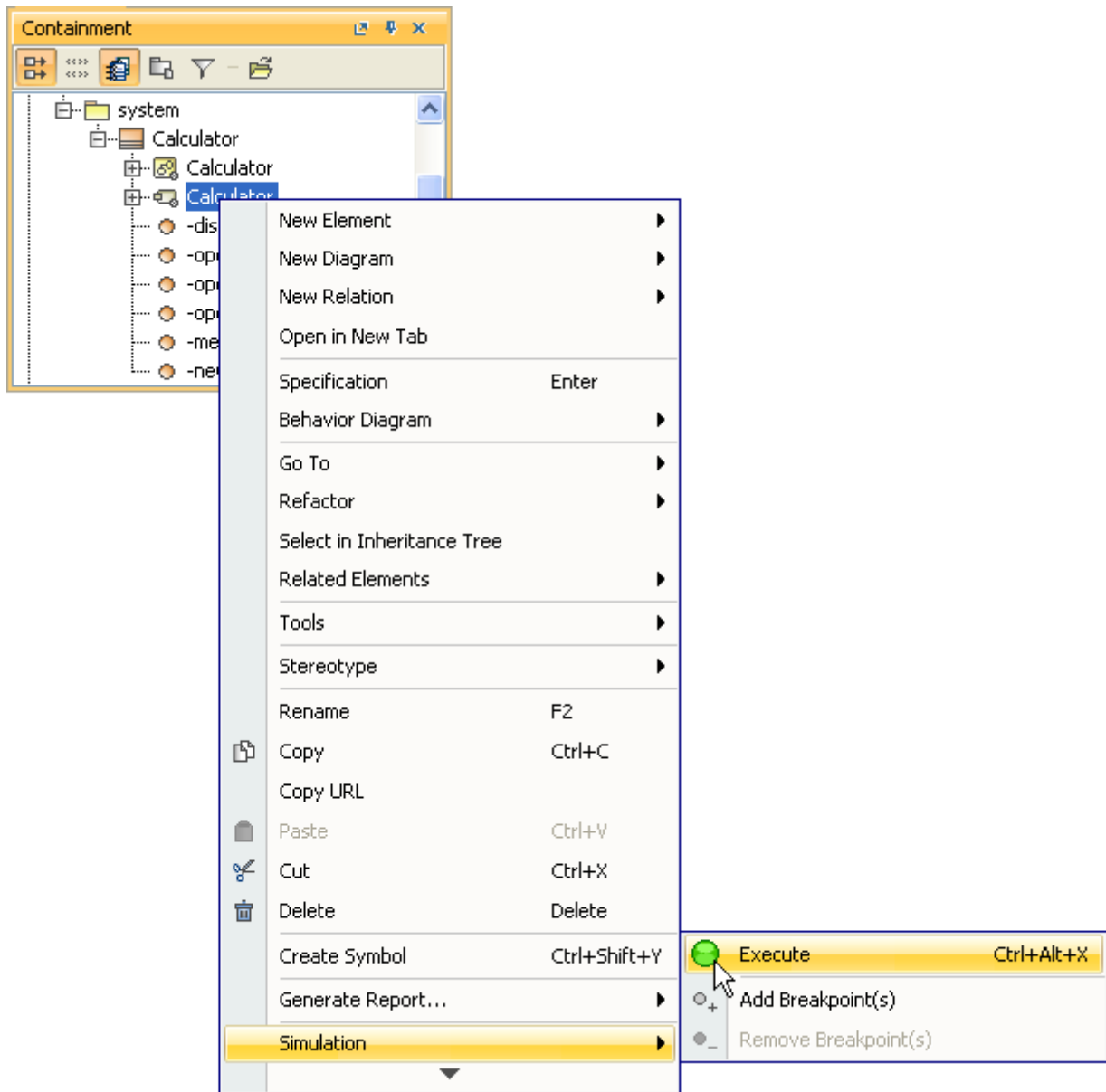


Figure 4 -- Executing Activity

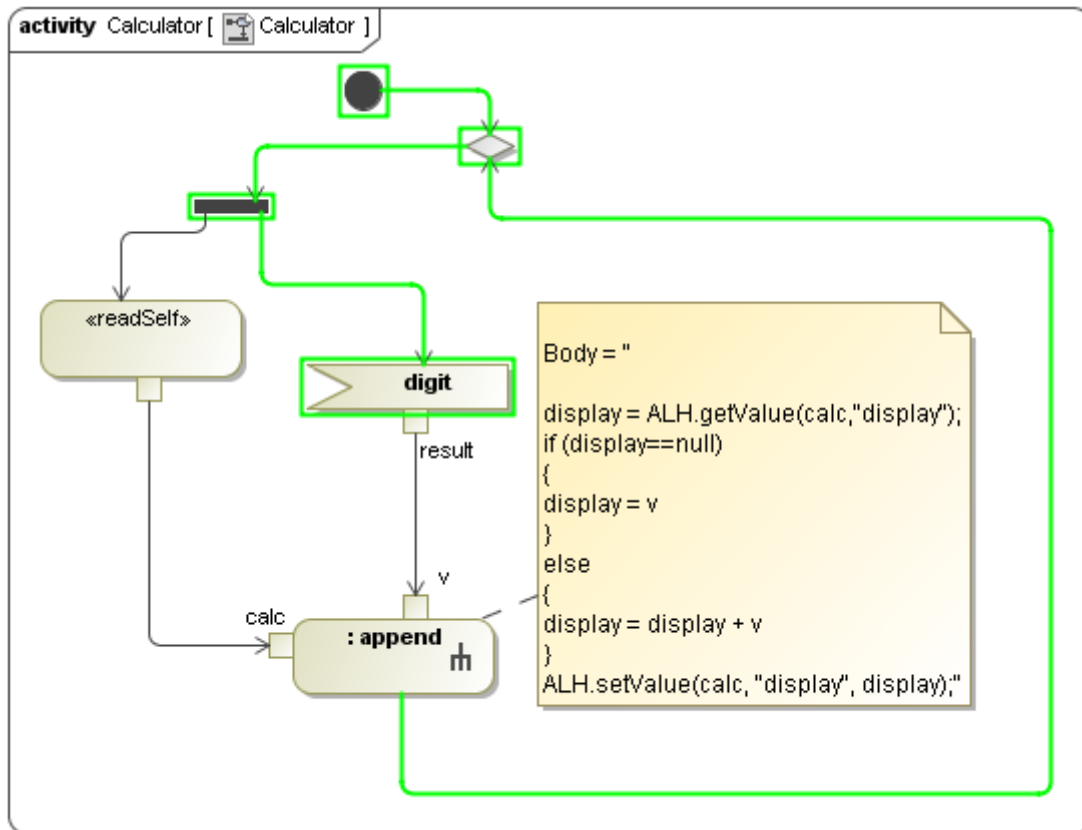


Figure 5 -- Animation of Activity Execution

2.1.1.2 StateMachine

If you select to execute a behavior, which is a State Machine (Figure 6), then it will be executed on the State Machine diagram whose context is the selected State Machine (Figure 7).

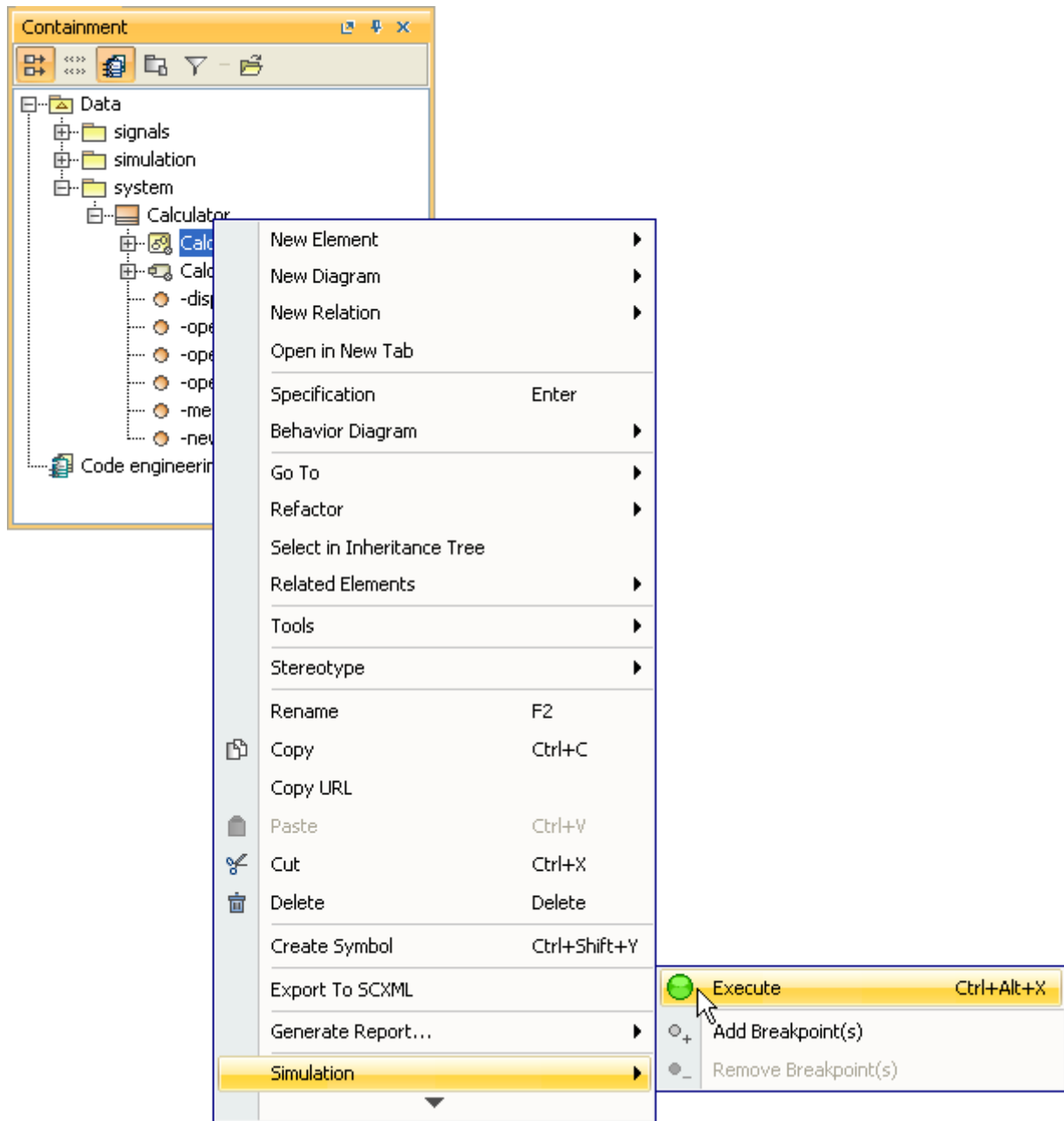


Figure 6 -- Executing State Machine

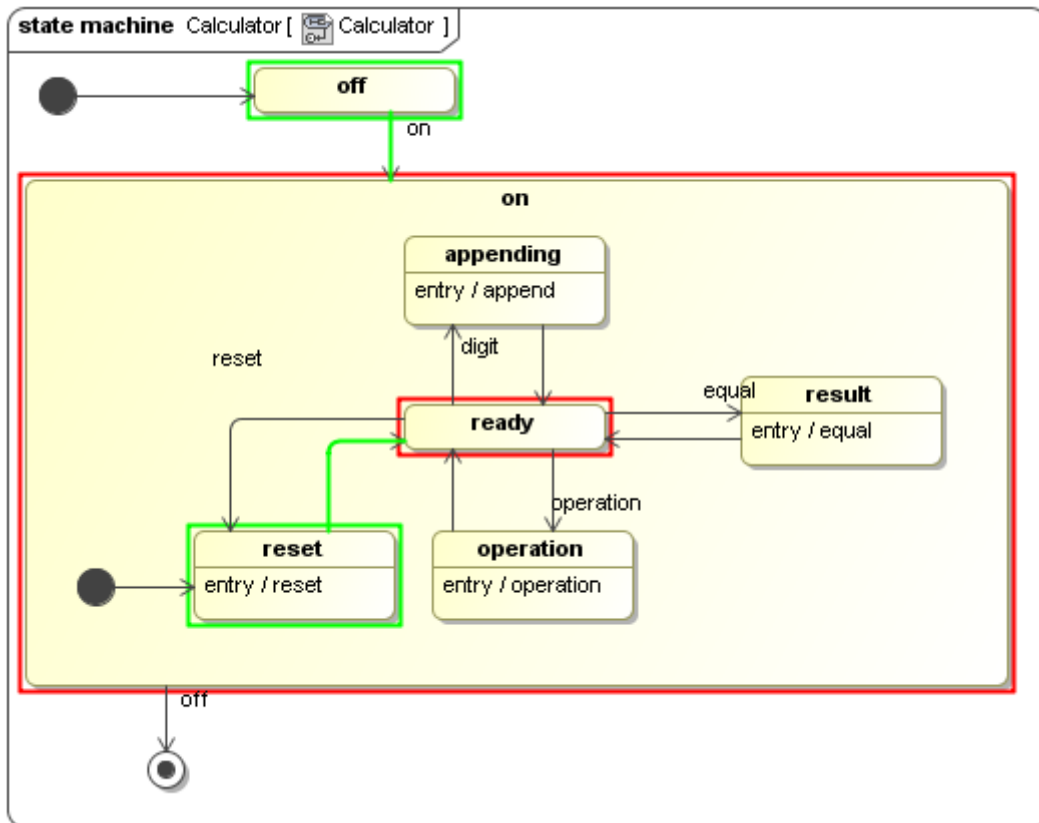


Figure 7 -- Animation of State Machine Execution

You can also select to execute from an Activity or a State Machine diagram directly either by:

- (i) opening the diagram and click the **Execute** button on the **Simulation Window** toolbar, or
- (ii) right-clicking the diagram and select **Simulation > Execute**.

The Behavior, which is the context of the diagram, will then be executed.

2.1.2 Class

You can select to execute a Class element that is not a Behavior. A simulation session will be created to execute the selected class. The runtime value whose type is the selected Class, will be created to store the simulated values. If the selected Class has a defined classifier behavior, either Activity or State Machine (Figure 8), then it will be executed once you have clicked the **Run Execution** button. For example, if you select to execute the Calculator class (Figure 1), the simulation will be performed on the Calculator state machine (Figure 7).

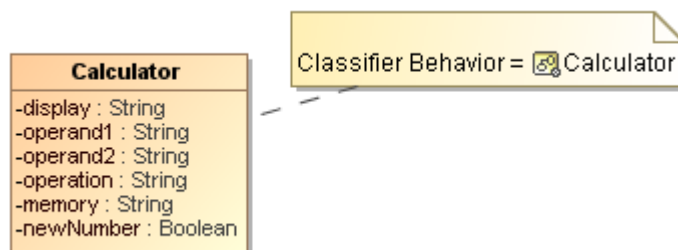


Figure 8 -- Show the Calculator Class, having a State Machine as its Classifier Behavior

If the class does not have a defined classifier behavior (Figure 9), the parametric will be executed instead (only if the selected class is a SysML Block containing Constraint Property(ies)).

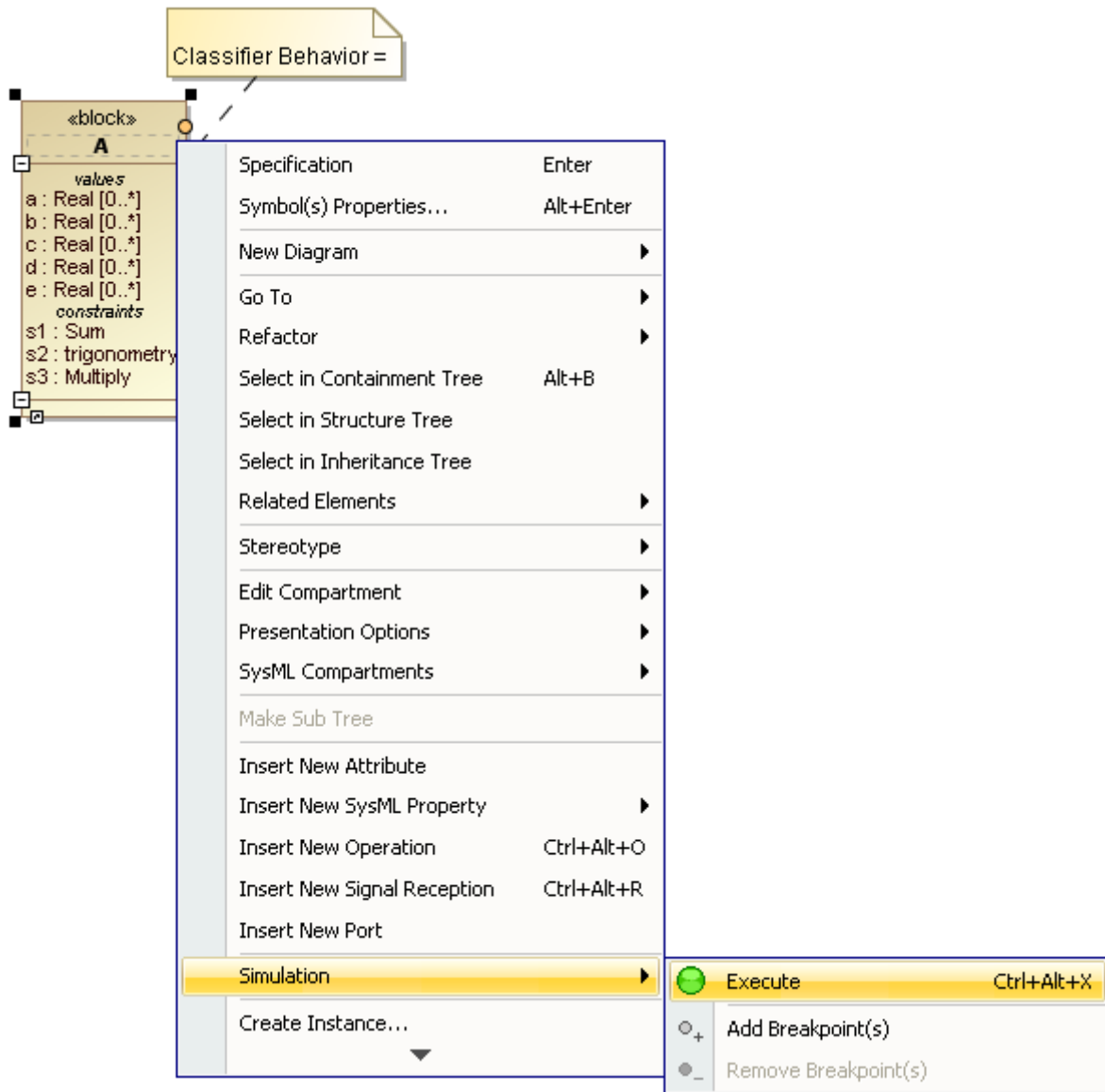


Figure 9 -- Executing SysML Block without a Defined Classifier Behavior

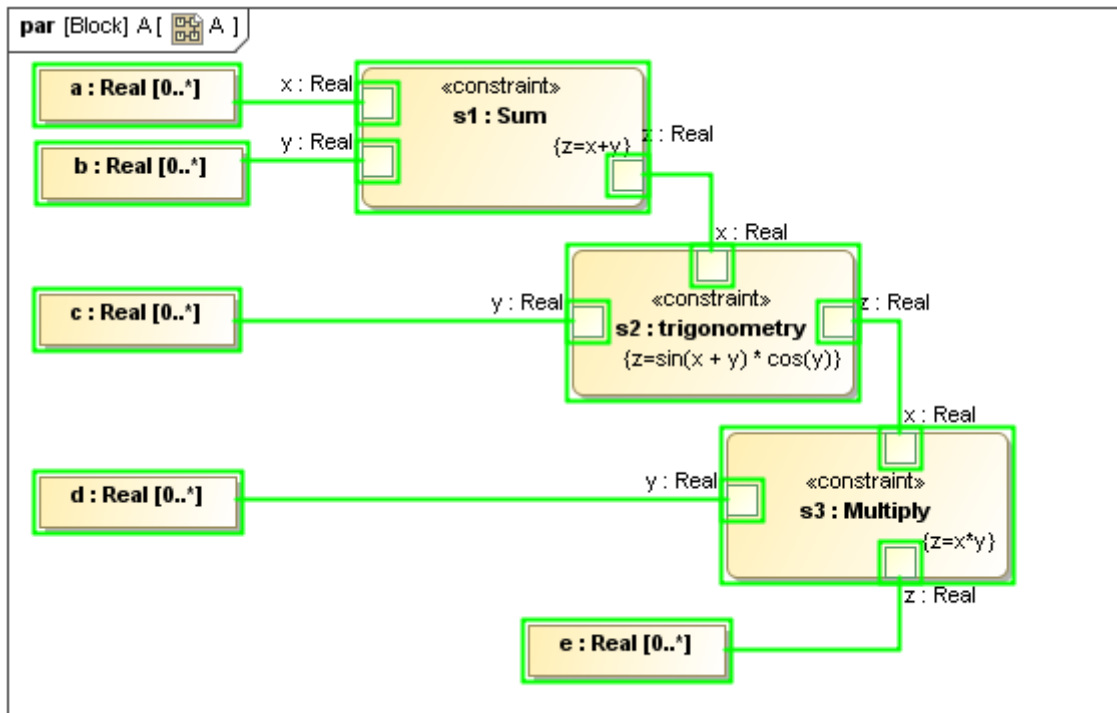


Figure 10 -- Animation of Parametric Execution

2.1.3 Diagram

To execute a diagram:

- Right-click a diagram and select **Simulation > Execute** (Figure 11). The element, which is the context of the diagram, will be executed the same way a behavior or a class is executed.

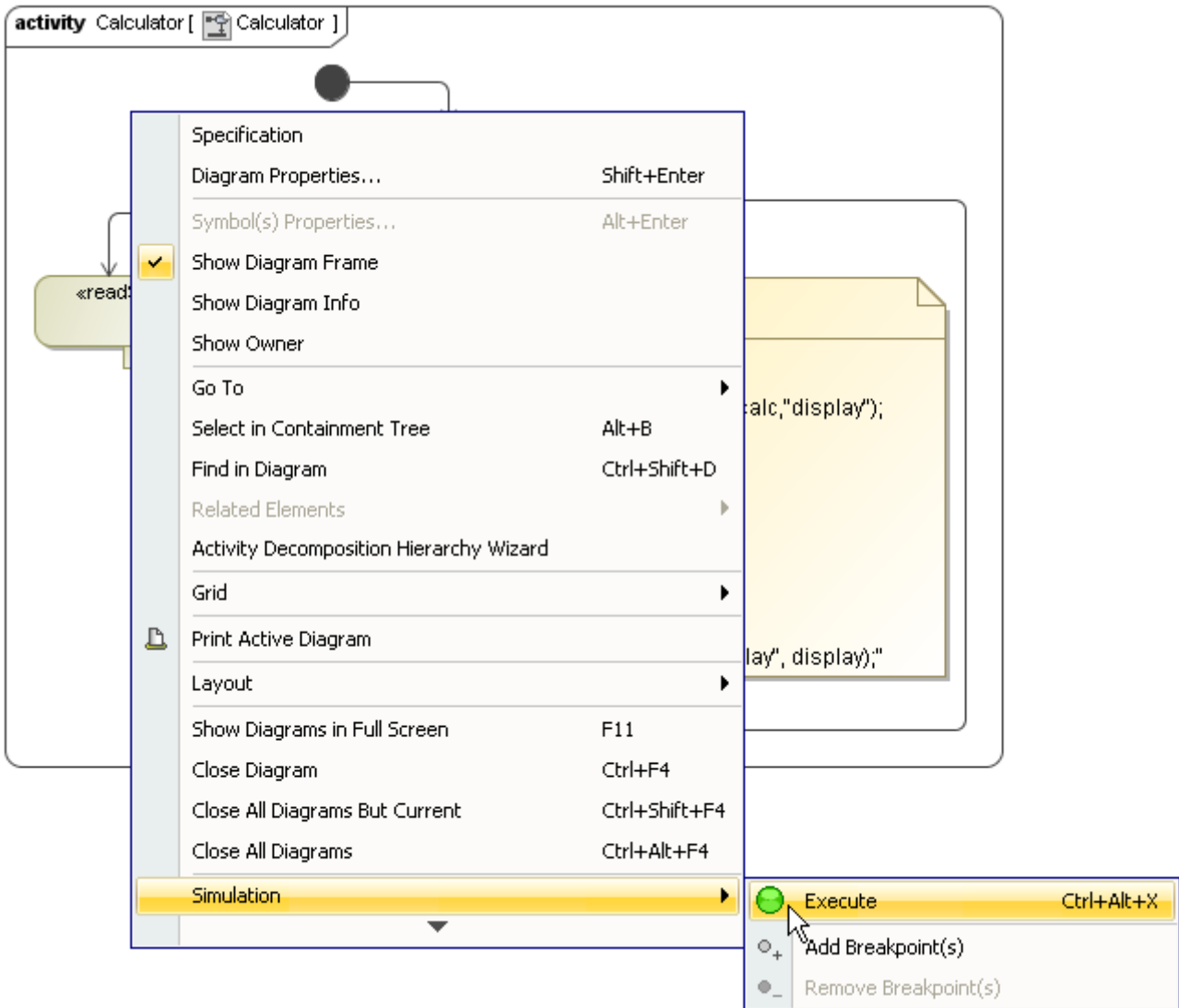


Figure 11 -- Executing Activity Diagram

2.1.4 Instance Specification

You can also simulate an InstanceSpecification. The runtime object and the runtime values will be created from the selected InstanceSpecification and its slot values. These runtime object and runtime values will be used for the execution. You can see more information about runtime object and runtime values in Section 5.4.2.

To execute an InstanceSpecification:

- Right-click an InstanceSpecification and select **Simulation > Execute**. The classifier of the selected InstanceSpecification will be executed the same way a behavior or a class is

executed. However, the slot values of the selected InstanceSpecification will be used to create the runtime values at the beginning of the execution (Figure 12).

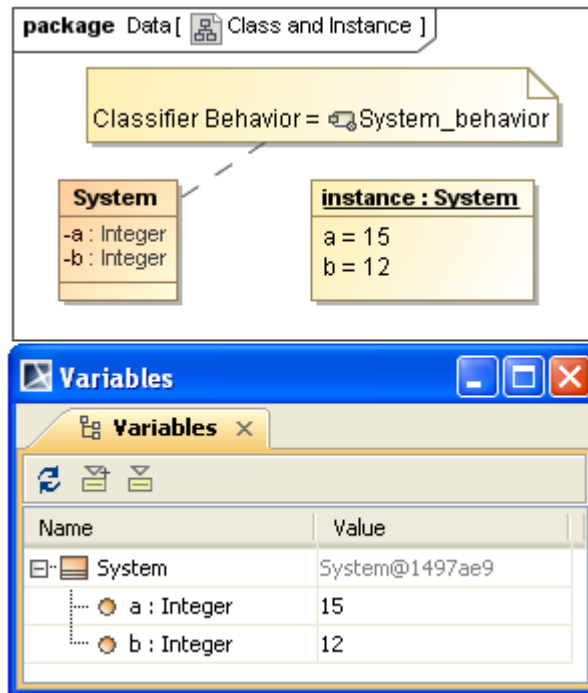


Figure 12 -- Runtime Values when Executing InstanceSpecification

2.2 Simulation by Executing the Execution Configuration

You can create a simulation by executing the Execution Configuration, which is a Class element having the «ExecutionConfig» stereotype applied, through the (i) context menu **Simulation > Execution** or (ii) Simulation Control toolbar.

(i) To execute an Execution Configuration through the context menu:

- Right-click an Execution Configuration and select **Simulation > Execution** (Figure 13).

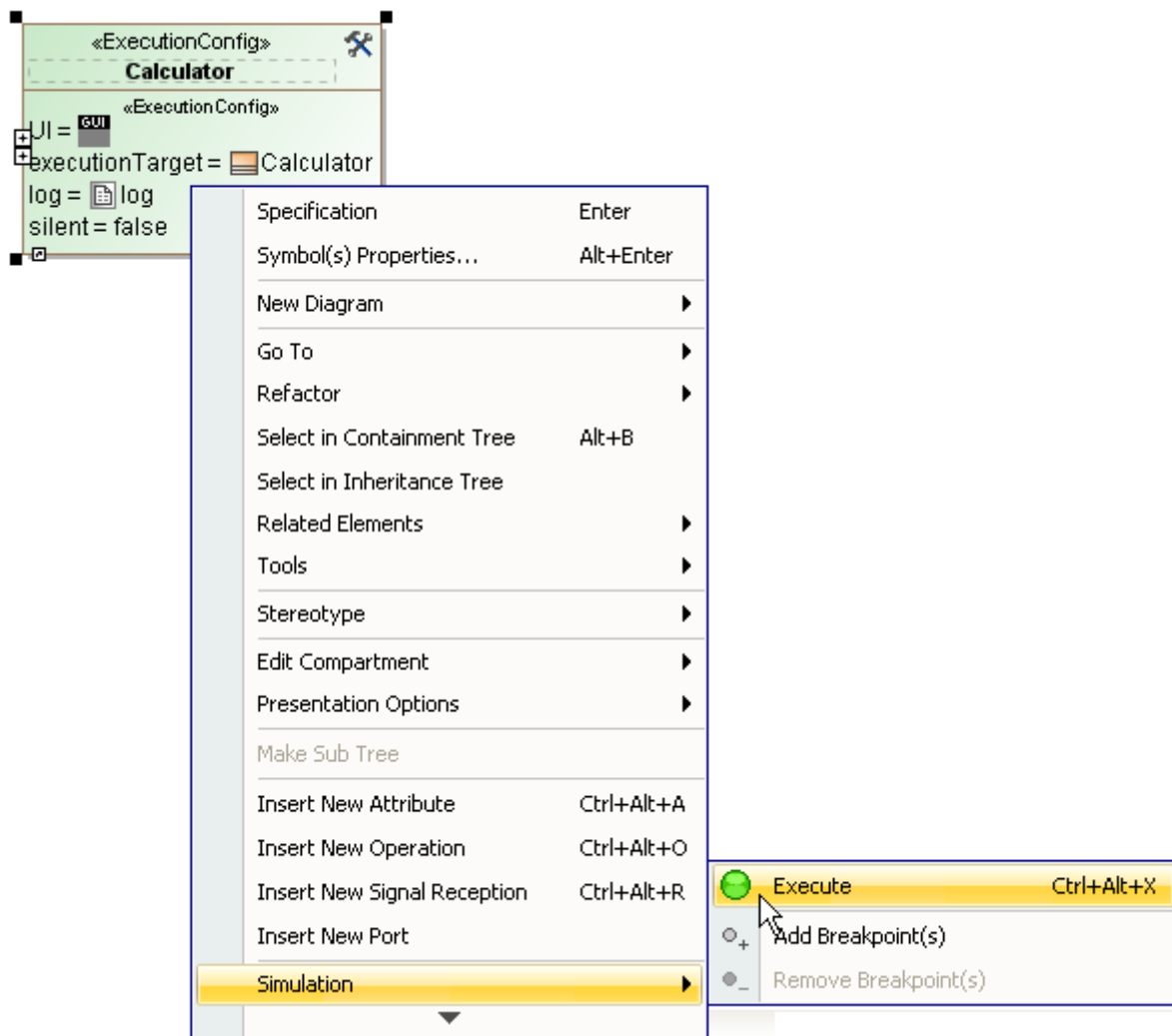


Figure 13 -- Executing from ExecutionConfig

(i) To execute an Execution Configuration through the **Simulation Control** toolbar:

- Select the execution configuration in the drop-down list (all of the execution configurations in the open project will be listed in the drop-down list) on the **Simulation Control** toolbar and click the **Run '<name of execution configuration>' Config** button (Figure 14).



Figure 14 -- Executing Configuration from Simulation Control Toolbar

For more information on how to use Execution Configuration, see Section 3.

3. Execution Configuration

3.1 ExecutionConfig Stereotype

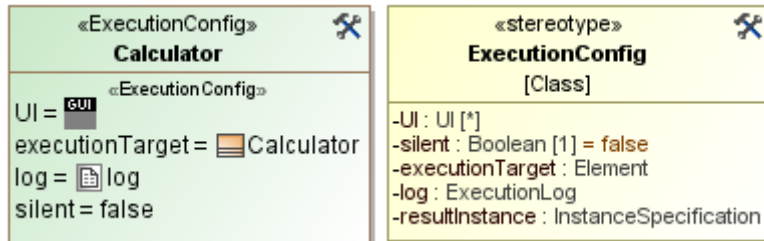


Figure 15 -- An Execution Configuration

Cameo Simulation Toolkit provides a model-based execution configuration through the «ExecutionConfig» stereotype. The «ExecutionConfig» configuration properties consist of:

- **executionTarget** – the element from which the execution should be started.
- **silent** – if the value is true, no animation (nor idle time) will be used.
- **ui** – the user interface mockup configuration to be started with the execution.
- **log** – the element in which the execution trace will be recorded.
- **resultInstance** - the InstanceSpecification which the execution results will be saved as its slot values. If resultInstance is not specified, then the execution results will not be saved event if the executionTarget is an InstanceSpecification.

You can select and execute an execution configuration directly.

| | |
|-------------|--|
| NOTE | You can use Execution configurations as the target elements in other execution configurations in the next release of Cameo Simulation Toolkit. |
|-------------|--|

3.2 Execution Log

You can record all runtime event occurrences into a special model element by creating a new ExecutionLog element (a Class having the «ExecutionLog» stereotype applied) and make a reference to the “log” property in an ExecutionConfig before a simulation (Figure 16).

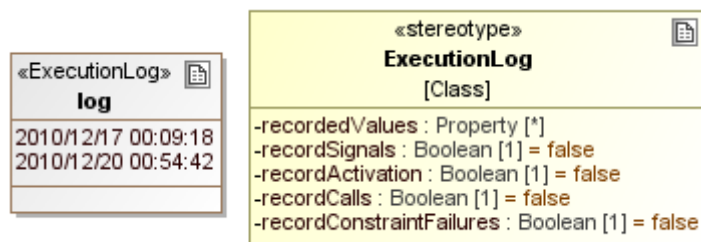


Figure 16 -- Execution Log

A model-based execution log/trace has many advantages and some of them are as follows:

- it can be used as a source for various customized reports and analyses using the MagicDraw validation mechanism (as both are model-based).
- it allows you to import execution data into any other UML compliant tool.

You can record multiple simulation sessions or test results in the same «ExecutionLog» element. The session starting time can be seen as the name of the attribute. Currently, you can record the following runtime data (see Figure 17):

- **Signal Instance** (when recordSignals = true) under the “Signal Instances” node – timestamp (i.e. the relative occurrence time in milliseconds: ‘0’ when start execution), signal type and target (Figure 17).
- **Sequence of Activation** and **Sequence of Deactivation** (when recordActivation = true) under the “Activation sequence” node – timestamp and type of element being activated/deactivated.
- **Behavior Call** and **Operation Call** (when recordCalls = true) under the “Behavior Calls” and “Operation Calls” nodes, respectively – timestamp, type, target and value(s).
- **Runtime Value** (when the recordedValues attribute has at least one Property selected) under the “Value Changes” node – timestamp, select Property and value(s) of the selected Property.
- **Constraint Failure** (when recordedConstraintFailures = true) under the “Constraint Failures” node – timestamp, element, target and value(s).

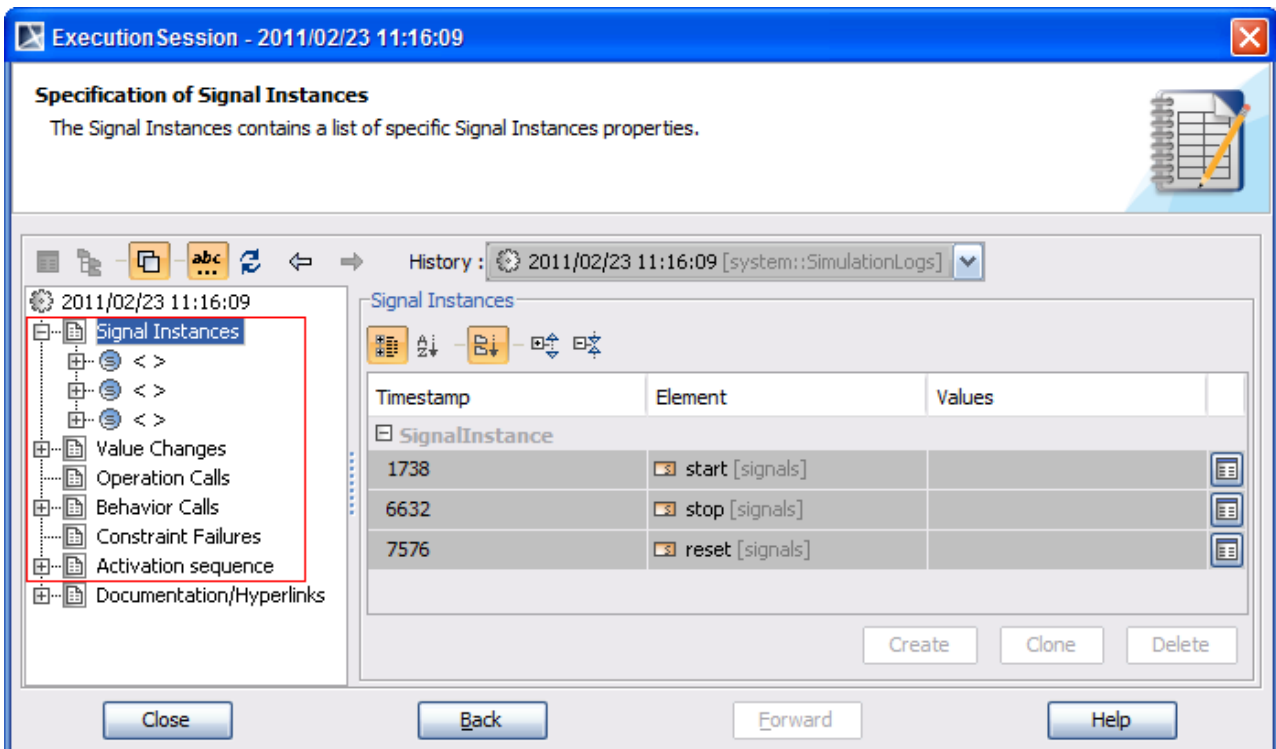


Figure 17 -- Example of Recorded Runtime Data, e.g., Signal Instance (StopWatch_advanced.mdzip)

3.3 User Interface Prototyping

Cameo Simulation Toolkit allows you to use custom mockups, which can be referenced in a model-driven UI config. The most basic UI config has two properties:

- (i) Represented model element (Classifier)

- (ii) External Java class file implementing a MockupPanel interface

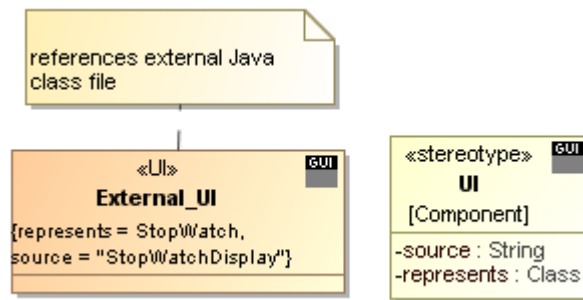


Figure 18 -- Using User Interface Prototyping

Whenever an execution engine creates a runtime object of a referenced classifier, a mockup UI will be instantiated and displayed. A MockupPanel interface allows a mockup to listen to all of the execution events, monitor structural feature values, and trigger signals.

3.4 UI Modeling Diagram Execution

The MagicDraw User Interface Modeling diagram becomes even more powerful and valuable when used with Cameo Simulation Toolkit. Supported UI components include:

- **Frames:** drag a Classifier to a UI Frame to bind the Classifier to the UI Frame (the «UI» stereotype will be automatically applied; its “*represents*” tag will then be set to the Classifier). In this case, we say that “the UI Frame represents the Classifier”. The “*source*” tag of the applied «UI» stereotype will also be set as “com.nomagic.magicdraw.simulation.uiprototype.UIDiagramFrame” by default.
- **Panels:** a UI Panel can hold any supported UI components (buttons, labels, sliders, checkboxes, text fields and even panels themselves).
 - If the UI Panel resides in a UI Frame, drag a Property of the Classifier the UI Frame represents to the UI Panel to bind such Property to the UI Panel (the «NestedUIConfig» stereotype will be automatically applied; its “feature” tag will then be set to the Property; and its “Text” tag will also be set to the name of such Property). In this case, we say that “the UI Panel represents the Property”.
 - If the UI Panel (child) resides in another UI Panel (parent), drag a Property of the Classifier typing the Property the parent UI Panel represents to the child UI Panel to bind such Property to the UI Panel (the «NestedUIConfig» stereotype will be automatically applied; its “feature” tag will then be set to the Property; and its “Text” tag will also be set to the name of such Property). This functionality allows you to bind nested parts (properties) of a Classifier to its correspondent nested UI Panels in a UI Frame representing the Classifier.
 - In addition, you can reuse existing UI components (all supported ones, except frame) in an existing UI Frame in another UI Panel, by dragging the UI Frame model in Containment Tree to such UI Panel (the «NestedUIConfig» stereotype will be automatically applied if not already; and its “config” tag will then be set to the UI Frame).
- **Group Boxes:** similar usage as Panels.
- **TextFields, Checkboxes, and Sliders:** drag a Property to one of these UI components to bind the Property to such UI components (the «RuntimeValue» stereotype will be automatically applied; its “*element*” tag will then be set to the Property; and its “Text” tag will also be set to the name of such Property). In this case, we say that “the UI component represents the Property”.

Once represented, the UI component will reflect the value of the represented Property in the Variables Pane during execution, and vice versa.

- **Labels:** drag a Property to a UI Label to bind the Property to such UI Label (the «RuntimeValue» stereotype will be automatically applied; its “*element*” tag will then be set to the Property; and its “*Text*” tag will also be set to the name of such Property). In this case, we say that “the UI Label represents the Property”. Once represented, the UI Label will display the value of the represented Property in the Variables Pane during execution.
- **Buttons:** a UI Button can be used to 1) send Signal(s), 2) call Operation(s) or 3) call Behavior(s):
 - **Sending Signal(s):** drag a Signal to a UI Button to associate the Signal with the UI Button (the «SignalInstance» stereotype will be automatically applied; its “*element*” (“*signal*”) tag will then be set to the Signal; and its “*Text*” tag will also be set to the name of such Signal). During execution, if this UI Button is pressed, it will send the associated Signal.
 - **Call Operation(s):** drag an Operation to a UI Button to associate the Operation with the UI Button (the «OperationCall» stereotype will be automatically applied; its “*element*” tag will then be set to the Operation; and its “*Text*” tag will also be set to the name of such Operation). During execution, if this UI Button is pressed, it will call the associated Operation.
 - **Call Behavior(s):** drag an Behavior (e.g., Activity) to a UI Button to associate the Behavior with the UI Button (the «BehaviorCall» stereotype will be automatically applied; its “*element*” tag will then be set to the Behavior; and its “*Text*” tag will also be set to the name of such Behavior). During execution, if this UI Button is pressed, it will call the associated Behavior.

Figure 19 demonstrates an example of using MagicDraw’s User Interface Modeling Diagram with Cameo Simulation Toolkit.

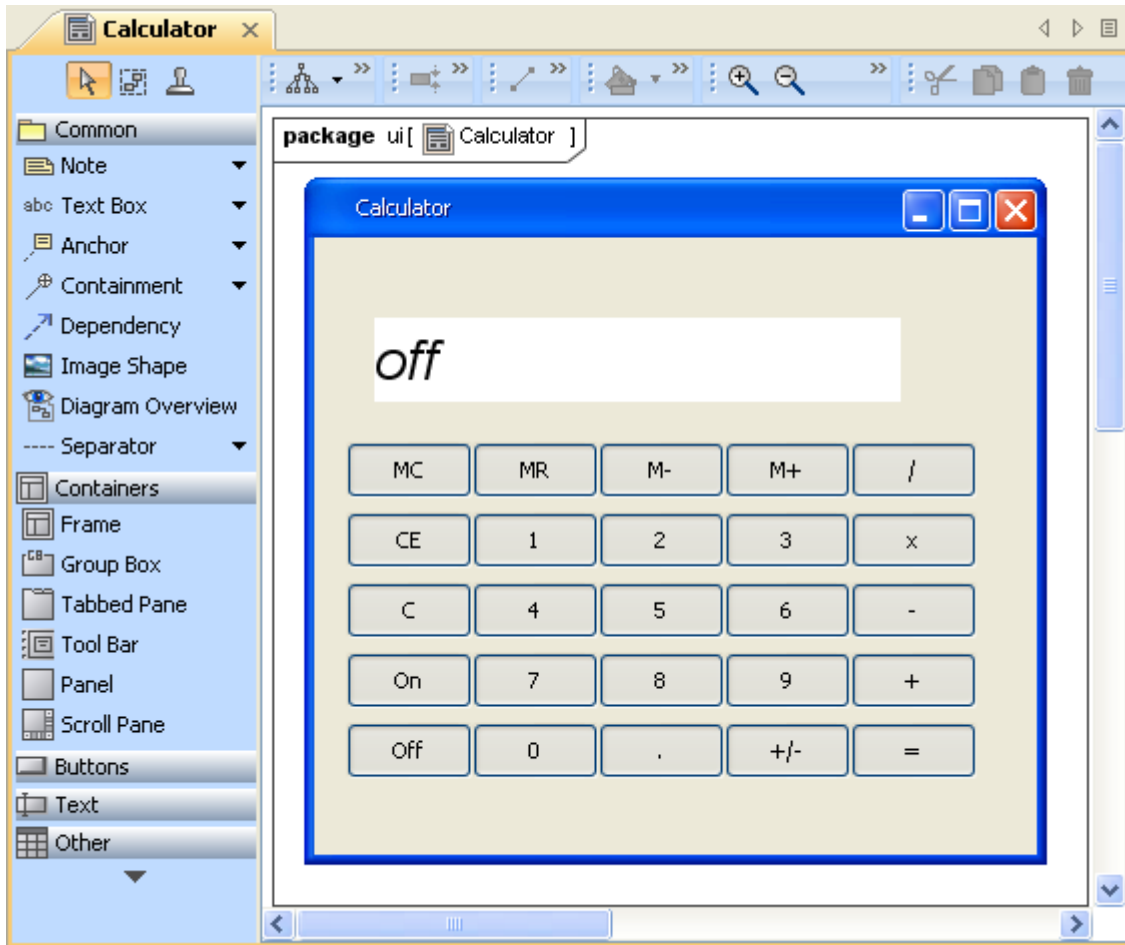


Figure 19 -- Example of Using User Interface Modeling Diagram

- Drag a Classifier to a UI Frame.
- Drag each Signal to each UI Button to associated a Signal.
- Drag any Classifier's Property to a UI Label to be represented.
- Reference the Frame in the "UI" tag of an ExecutionConfig.

See the Calculator.mdzip sample for more details. Just drag any GUI elements to a diagram, click **Execute**, and see them come alive!

NOTE

- The current version of Cameo Simulation Toolkit supports frames, panels, group boxes, labels, buttons, checkboxes, text fields, and sliders only.
- Do not drag and drop model element of existing UI Frame (from Containment Tree) on a diagram to create one more ComponentView/Frame symbol on such diagram. Cameo Simulation Toolkit still does not support 2 UI symbols of the same model element yet.
- Other samples worth trying include: test_nested_UI_panels.mdzip, test_UI.mdzip, StopWatch_advanced.mdzip, and SimpleUI_labelUpdate.mdzip.

3.5 ActiveImage and ImageSwitcher

ImageSwitcher is a predefined subtype of UI config. It is a simple, but flexible and powerful animation tool. All you need is to create an «ImageSwitcher» element, specify a represented Classifier, and then create as many attributes and different states as you wish to see in the animation. Each attribute is called «ActiveImage» and has the following properties:

- **Image** – the image that will be used in animation (browse the file or drag the image directly from web browser).
- **activeElement** – the element that will use the image once it has been activated (normally the state of a represented classifier).
- **onClick** – the signal that will be triggered once the displayed image has been clicked.

Figure 20 demonstrates an example of how to use ImageSwitcher and ActiveImage (see FlashingLight.mdzip sample):

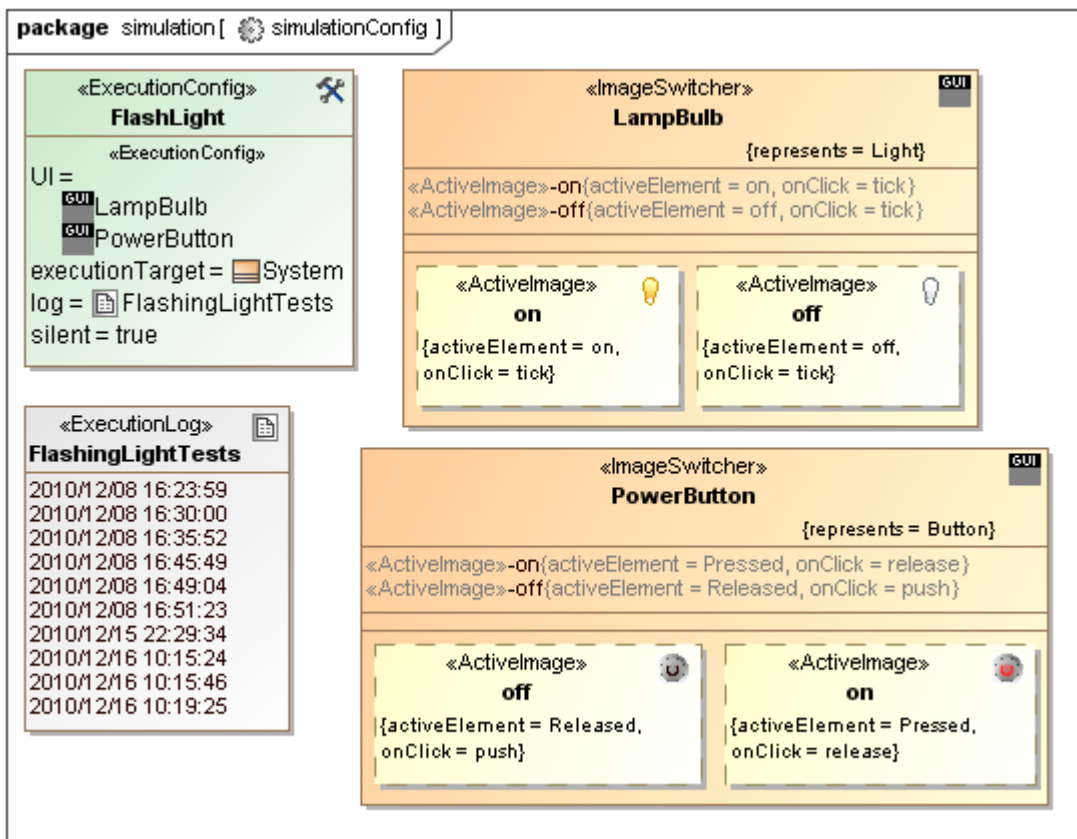


Figure 20 -- FlashLight Sample

Once the **FlashLight** ExecutionConfig is executed, the mockup UI will be displayed (Figure 21). You can then click on the Power button (circle one) to start execution, i.e. making the light bulb blink (see FlashingLight.mdzip sample).

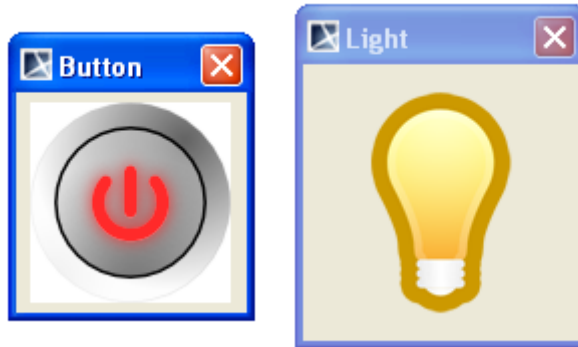


Figure 21 -- FlashLight Sample - Runtime Animation

4. Animation

Active elements in a diagram will be annotated during execution using the same annotation mechanism used in the active validation:

- Active and visited elements will be annotated with red and green respectively.
- Runtime values will be visible in the tooltip text of active elements.

NOTE

- If an execution trace remains visible in a diagram, click the diagram to clear it.
- If a model is executed in silent mode by selecting an execution configuration whose silent tag value is set to 'true', the animations will be disabled (See Section 3. Execution Configuration for more information).

4.1 Active and Visited Elements

Active elements are the elements, which a simulation session is focused on (see Section 5.1 Understanding Simulation Sessions for more information). It can also be considered that the active elements are the elements that are currently being executed in a simulation session. They will be annotated with red (by default). Once the active elements have been executed, they will become visited elements and be annotated with green by default (Figure 22).

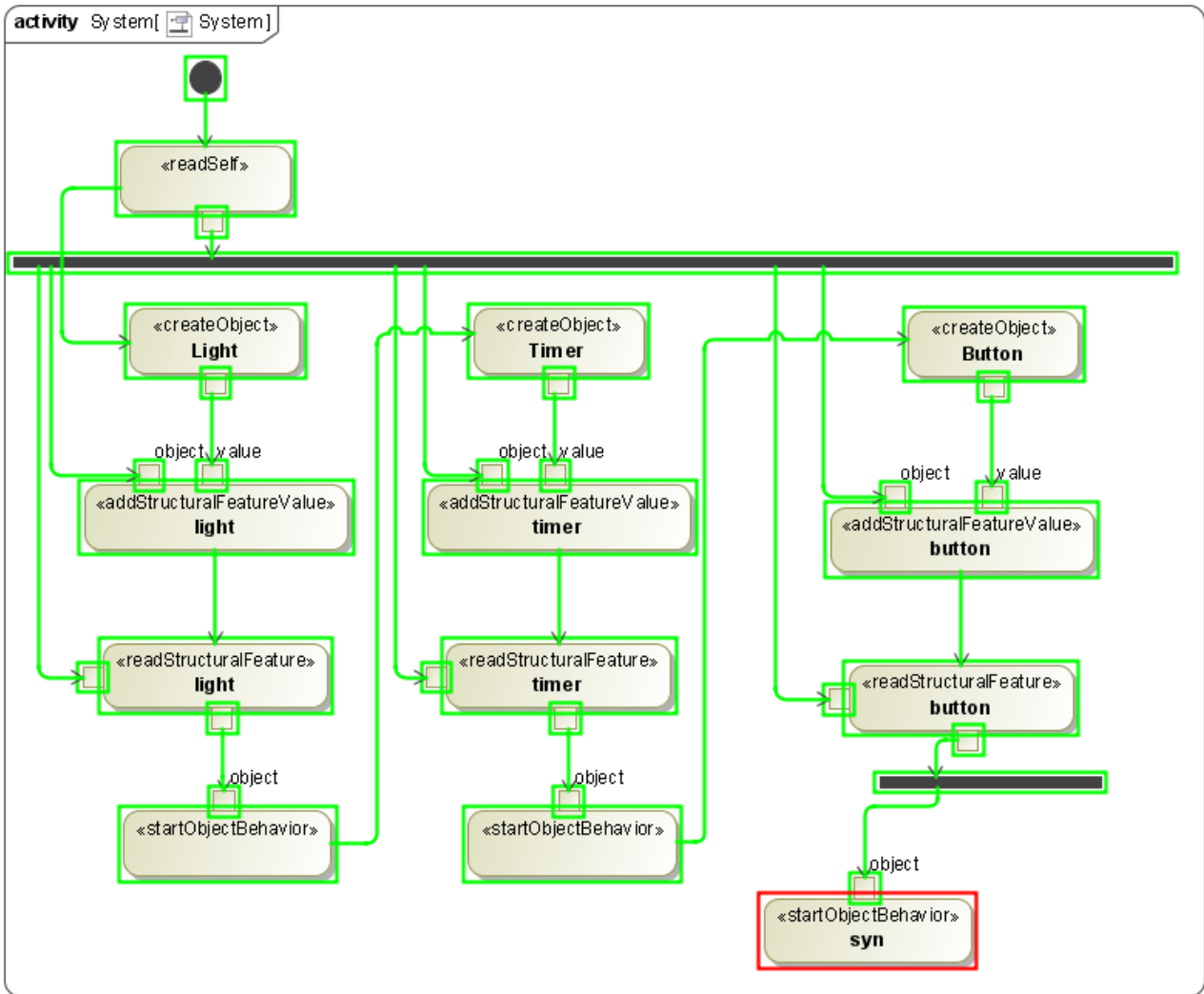


Figure 22 -- Animation: Active Element in Red and Visited Elements in Green

4.2 Customizing Animation Colors

There are three kinds of annotated elements in model execution: (i) active element, (ii) visited element, and (iii) breakpoint element. By default, active elements are annotated with red, visited elements with green, and breakpoint elements with yellow. Cameo Simulation Toolkit allows you to customize the color of annotated elements through the **Environment Options** dialog. You can open the **Environment Options** dialog by clicking **Options > Environment** on the MagicDraw main menu.

To customize animation colors:

1. Open the **Environment Options** dialog.
2. Select the Simulation node on the left-hand side (Figure 23).
3. Customize the colors of the active, visited, or breakpoint elements.

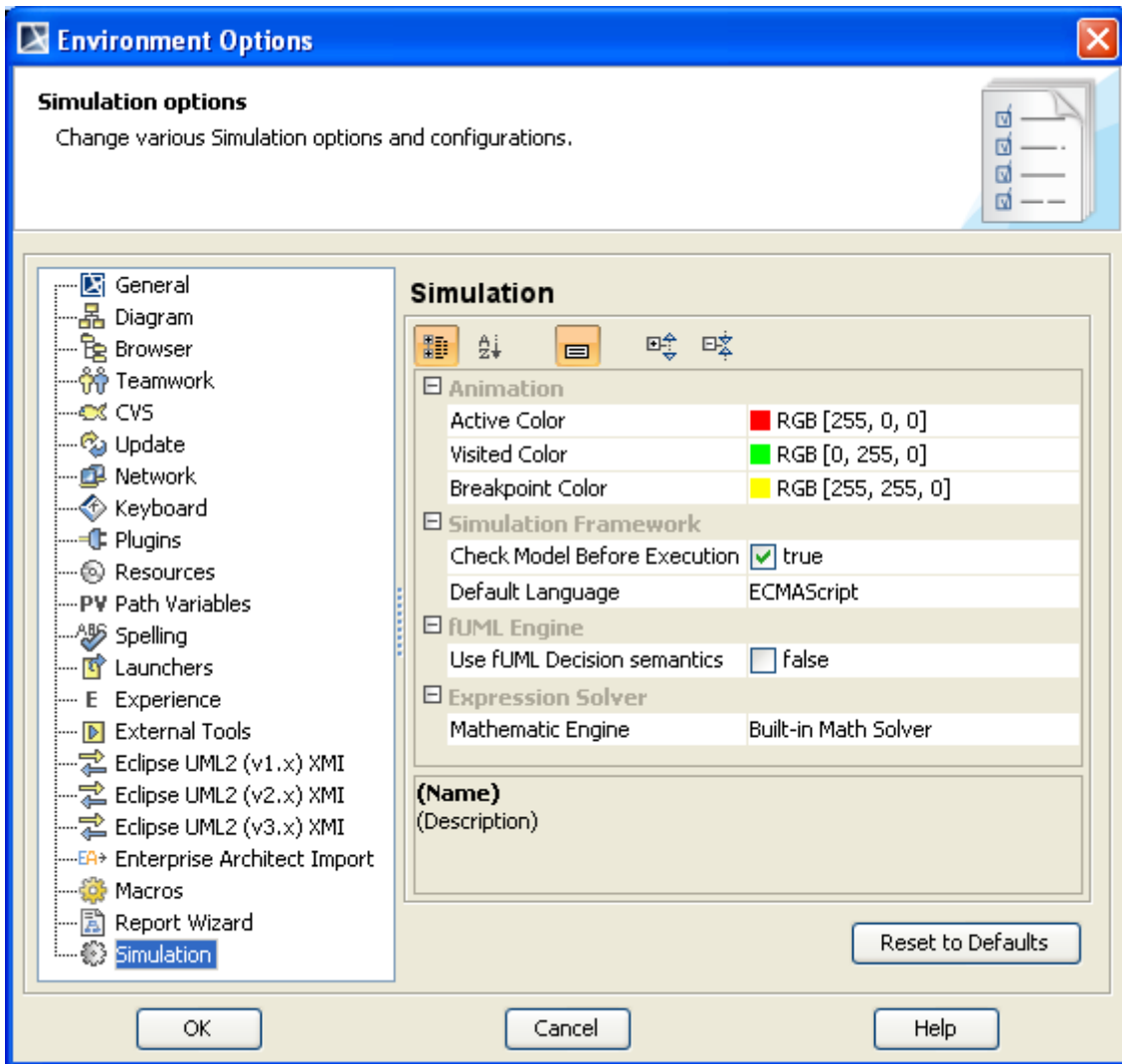


Figure 23 -- Customizing Animation Colors in the Environment Options Dialog

5. Simulation Debugging

5.1 Understanding Simulation Sessions

Cameo Simulation Toolkit creates a simulation session(s) while a model is being executed. The simulation session contains a context with a specified runtime value. The context of simulation session is the executing UML element that can be either a Class element or a sub-type of a Class. When the context element is executed, a runtime object will be created to store the simulated values.

You can create multiple simulation sessions during a single execution, for example, an Activity execution. If the executed Activity contains callBehaviorActions, a new simulation session will be created to execute each callBehaviorAction. All of the simulation sessions will be shown in the **Sessions Pane** during execution. The **Sessions Pane** will display the simulation sessions by their context elements in the tree node (Figure 24).

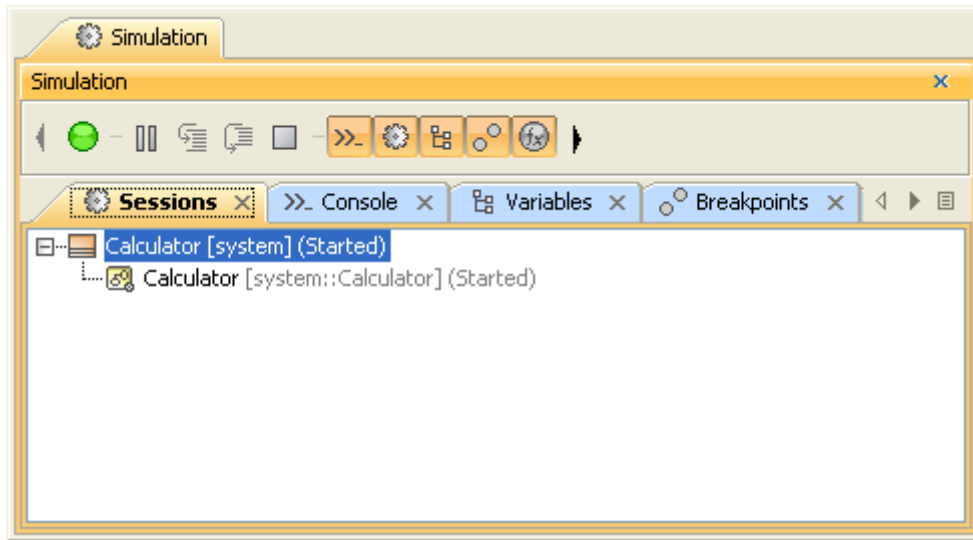








Figure 24 -- Sessions Pane

5.2 Simulation Debugger

Cameo Simulation Toolkit allows you to debug a running model by using the debugger buttons, i.e. Suspend, Resume, Step into and Step over.

Table 2 -- Execution and Debugger Buttons

| Button | Name | Function |
|---|---------------|--|
|  | Run Execution | To run a selected simulation session. |
|  | Suspend | To pause the execution of a selected simulation session in the Sessions Pane. |
|  | Resume | To resume a suspended simulation session. |
|  | Step into | To execute and run animation in the current active element of a selected simulation session in the Sessions Pane . |
|  | Step over | To execute the current active element of a selected simulation session and run animation in the background. |
|  | Terminate | To terminate a selected session in the Sessions Pane . If the selected session contains sub-sessions, all of the sub-sessions will be terminated. |

You can also pause the execution of a model at pre-defined breakpoints (see Section 5.5), examine and edit variables in Variables Pane (Section 5.4.1), or execute element-by-element using **Step into / Step over** button.

The Debugger Pane includes a player-like control panel for a step-by-step execution (see Table 2 above), threads/behaviors with an expandable stack trace (Understanding Simulation Sessions), Variables Pane/run-

time structure (Runtime Values Monitoring), Breakpoints pane (Breakpoints), and input/output console for custom commands or expressions evaluation (Console).

5.3 Simulation Console

5.3.1 Console Pane

Cameo Simulation Toolkit provide Simulation Console for displaying simulation information during model execution. The displayed information could contain hyperlink to the model element in the MagicDraw project. The model element will be selected in containment browser when you click on the hyperlink.

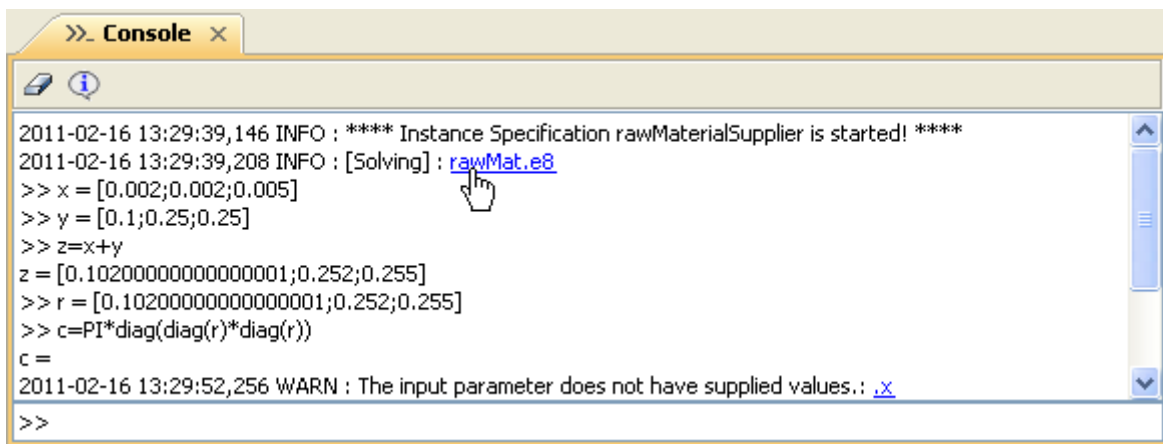




Figure 25 -- Simulation Console with simulation information during parametric execution.

Table 3 -- Console Pane Buttons

| Button | Name | Function |
|---|--------------------------|--|
|  | Clear Console | To remove all displayed simulation information in Simulation Console. |
|  | Show Runtime Information | To the runtime information of the Cameo Simulation Toolkit. The runtime information compose of the registered execution engines, available scripting engines and the active simulation sessions (Figure 26). |

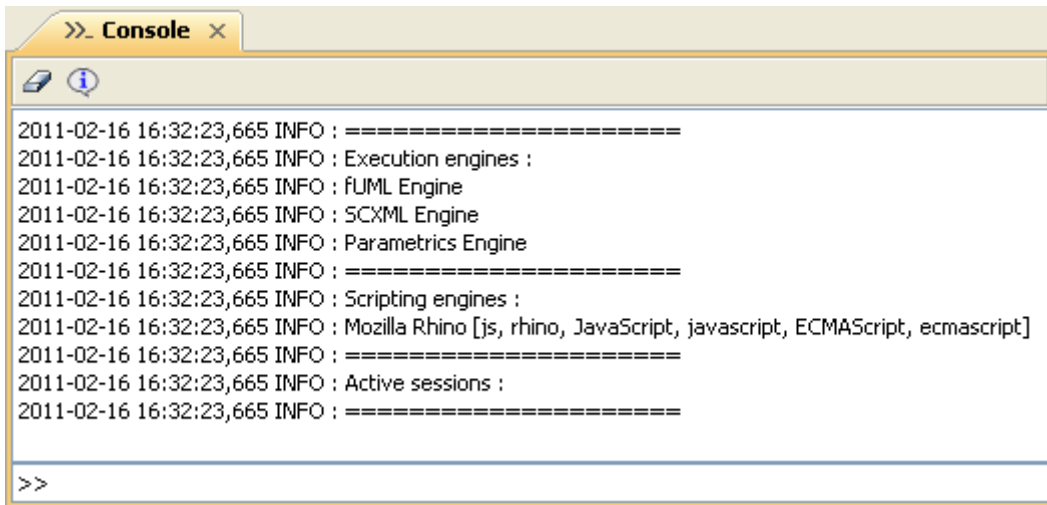


Figure 26 -- Runtime Information of Cameo Simulation Toolkit

5.3.2 Simulation Information

There are 6-levels of information which can be displayed in the Simulation Console (ascending sorted by priority):

- **TRACE**: Trace information.
- **DEBUG**: Debugging information.
- **INFO**: Normal information.
- **WARN**: Warning information.
- **ERR**: Error information.
- **FATAL**: Fatal information.

By default, only the information with priority equal to INFO and higher (WARN, ERR, FATAL) will be displayed in Simulation Console. You can customize the information displayed in Simulation Console by editing the **simulation.properties** file in the **data** directory in MagicDraw installation directory.

You can use text editor to edit this file. Go to **log4j.category.SIM_CONSOLE** and change from first parameter the lowest priority level that can be shown in the Simulation Console (INFO is the default value).

```
log4j.category.SIM_CONSOLE=INFO,SimConsoleApp,SimXMLApp
```

For example, you could change the first parameter of **log4j.category.SIM_CONSOLE** to **TRACE** for allow Simulation Console to display all level of simulation information.

```
log4j.category.SIM_CONSOLE=TRACE,SimConsoleApp,SimXMLApp
```

You can see more information on customizing display in Simulation Console from the comment in the **simulation.properties** file.

5.3.3 Simulation Log File

During execution, the simulated information will be displayed in the Simulation Console. However, the Simulation Console is limited to display only 60,000 characters for the sake of performance. If the simulation information exceed the maximum capacity, only the latest 60,000 characters will then be displayed. Nevertheless, your old simulation information will be automatically archived in the **simulation.log** file in user home directory (<User home directory>/magicdraw/<version>). The **simulation.log** file is an XML file (or a text file - to customize it, see the comment in the **simulation.properties** file) that record all simulation information that had ever displayed in the Simulation Console during model execution.

5.4 Runtime Values Monitoring

5.4.1 Variables Pane

You can select a session in the **Sessions Pane** (Figure 27) to display the runtime objects and values that correspond to the context element of a selected session in the **Variables Pane**.

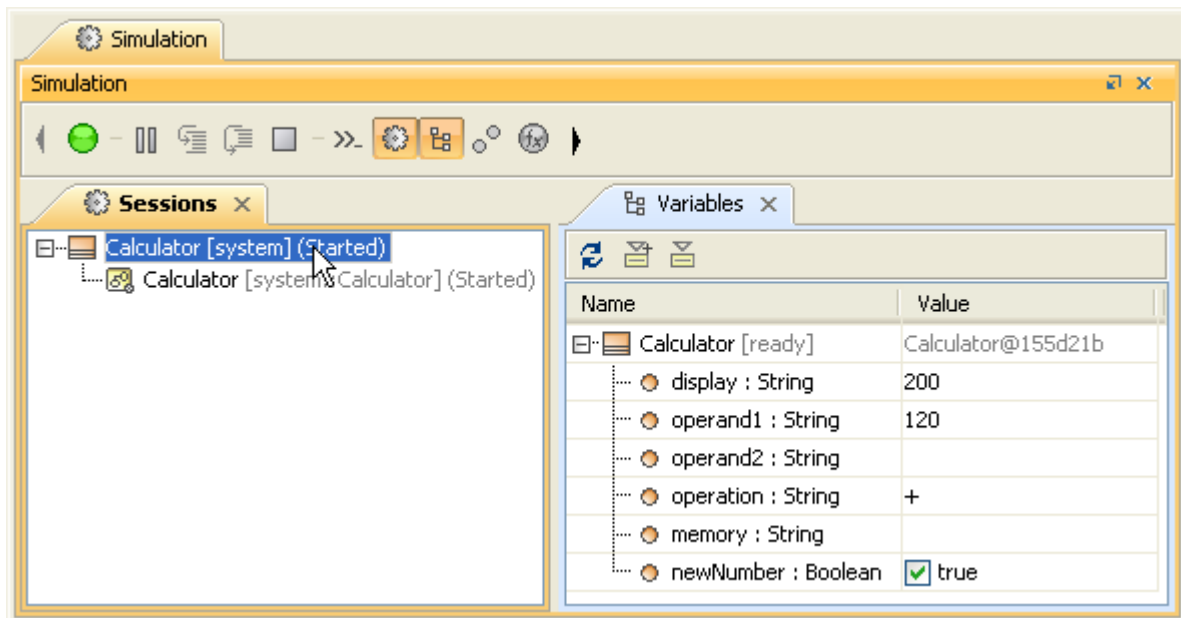


Figure 27 -- Variables Pane of a Simulation Session

When a model is being executed, (5.4.1.1) a context, (5.4.1.2) runtime objects, and (5.4.1.3) runtime values will be created to store the simulated values of the model.

5.4.1.1 Context

A simulation session is always associated with its context of execution. The context of a simulation session is a Class or one of its subtypes. When a context element is executed, a runtime object (of the context's type) will be created to store the runtime values. In Figure 27, the context of the selected simulation session is the "Calculator" class.

5.4.1.2 Runtime Object

A runtime object is the simulated value of a Class. In other words, it is a runtime instance of a Class, and hence of the context as well. In Figure 27, the runtime object of the simulation session context is the "Calculator@155d21b" instance. Since the runtime instance is the "Calculator" Class type's, it could contain structural features (which corresponds to the Class attributes), for example, "display" and "operand1".

5.4.1.3 Runtime Value

A runtime value refers to the value of the structural features mentioned in section 5.4.1.2 above, for example, "200" and "120". However, if the type of a structural feature is a classifier, its runtime value can also refer to another runtime object of a structural feature type.

The **Variables Pane** (Figure 27) displays the structure of an executing model and the runtime values during the execution of the model. This pane contains two columns: (i) Name and (ii) Value:




(i) Name column

The Name column represents the context and its structural features. If the context is a State Machine session's, the current state of the context will be displayed in square brackets. If a structural feature is typed by a Class, which is the context of another State Machine session, the current state of such context will also be displayed in square brackets, after the structural feature.

(ii) Value column

The Value column represents the runtime values of those structural features in a Name column. A runtime value can be the input or output of an execution. You can directly edit the runtime values in the Value column if they are of the following types: Boolean, Integer, Real, and String.

Table 4 -- Variables Pane Toolbar Buttons and Functions:

| Button | Name | Function |
|---|------------------------|---|
|  | Refresh | To refresh the tree and values of the Variables Pane . |
|  | Export to New Instance | To create a new InstanceSpecification and export a selected runtime object to a newly-created InstanceSpecification. |
|  | Export to Instance | To export a selected runtime object to an existing InstanceSpecification. All of the slot values of the InstanceSpecification will be replaced by the runtime values of the runtime object. |

5.4.2 Runtime Object created from InstanceSpecification

At starting point of model execution, the runtime object will be created for storing the runtime values. If the element which is selected for execution is InstanceSpecification or the ExecutionConfig whose executionTarget is

InstanceSpecification. The runtime values will be created from the slot values. They will be assigned to the runtime object's structural features which equivalent to the slot's defining feature.

If the slot of an InstanceSpecification is empty, and the slot's defining feature has defined default value, then the runtime value will be created from the default value and will be assigned to the runtime object's structural feature instead. Figure 28 -- show the runtime object that is created for executing pipe InstanceSpecification. The InstanceSpecification contains only one slot value of length. Then, the runtime value which is created for the length structural feature of the runtime object, will be equal to this slot value (1.0). For the runtime values of radius and thickness, they will be equal to the default values of radius and thickness property of Pipe class (0.05 and 0.002 respectively).

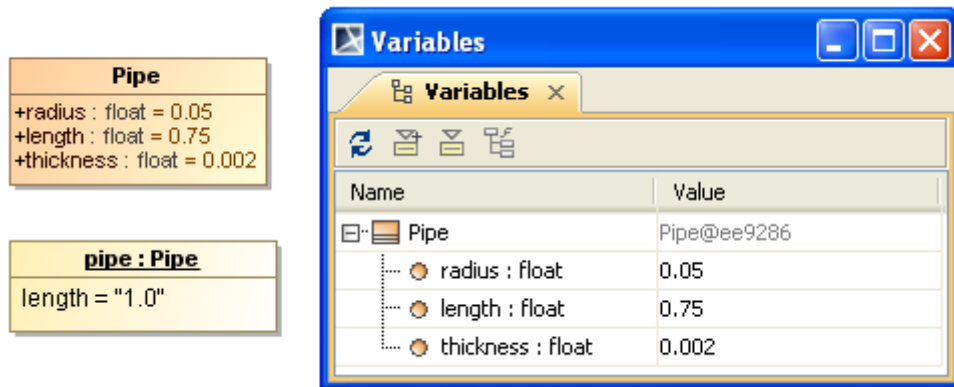


Figure 28 -- Variables Pane showing the Runtime Object

5.4.3 Exporting Runtime Objects to InstanceSpecification

You can export a runtime object, which is shown in the **Variables Pane**, to a model as an InstanceSpecification. You can export it to either (i) a newly-created InstanceSpecification or (ii) an existing InstanceSpecification. The values of a runtime object will be set to the slots of an InstanceSpecification.

(i) To export a runtime value to a new InstanceSpecification:

1. Either (i) click a row that has a runtime object to be exported in the Name column and click the **Export to New Instance** icon on the **Variables Pane** toolbar or (ii) right-click it and select **Export to New Instance** (Figure 26).

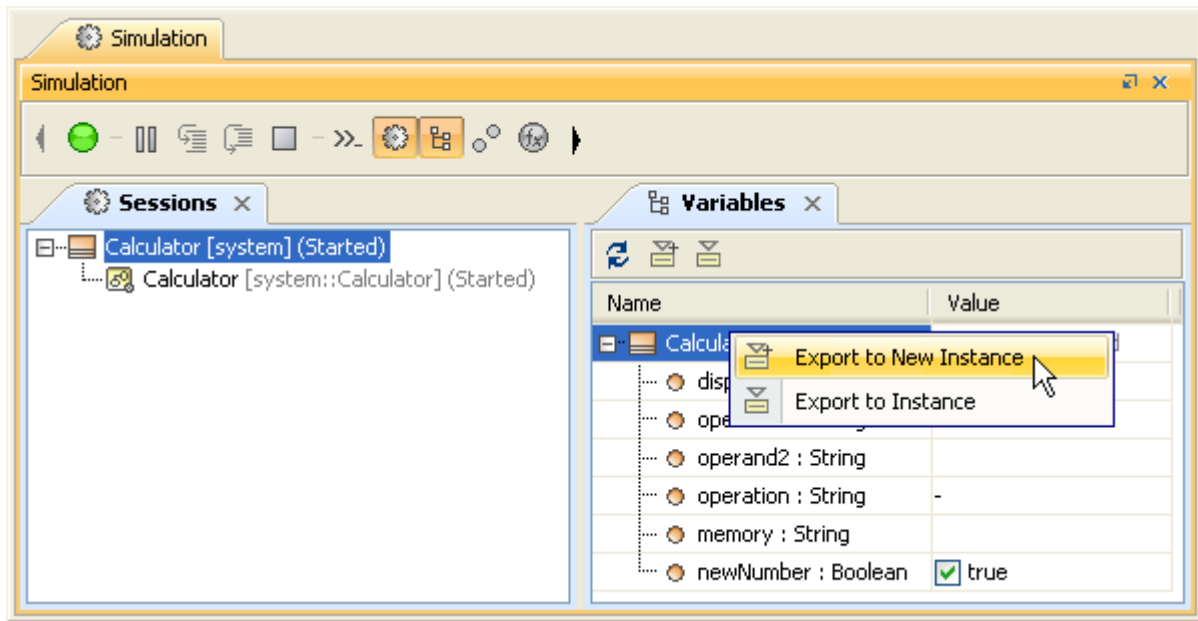


Figure 29 -- Export to New Instance Context Menu

2. The **Select Owner** dialog will open. Select the owner of the created InstanceSpecification and click **OK** (Figure 30).

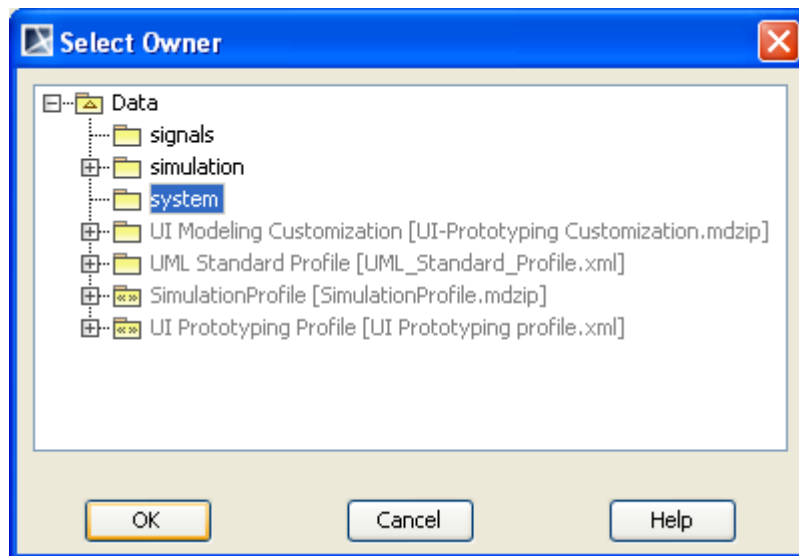


Figure 30 -- Selecting an InstanceSpecification Owner in the Select Owner Dialog

(ii) To export a runtime value to an existing InstanceSpecification:

1. Either (i) click a row that has a runtime object to be exported in the Name column and click the **Export to Instance** icon on the **Variables Pane** toolbar or (ii) right-click it and select **Export to Instance** (Figure 31).

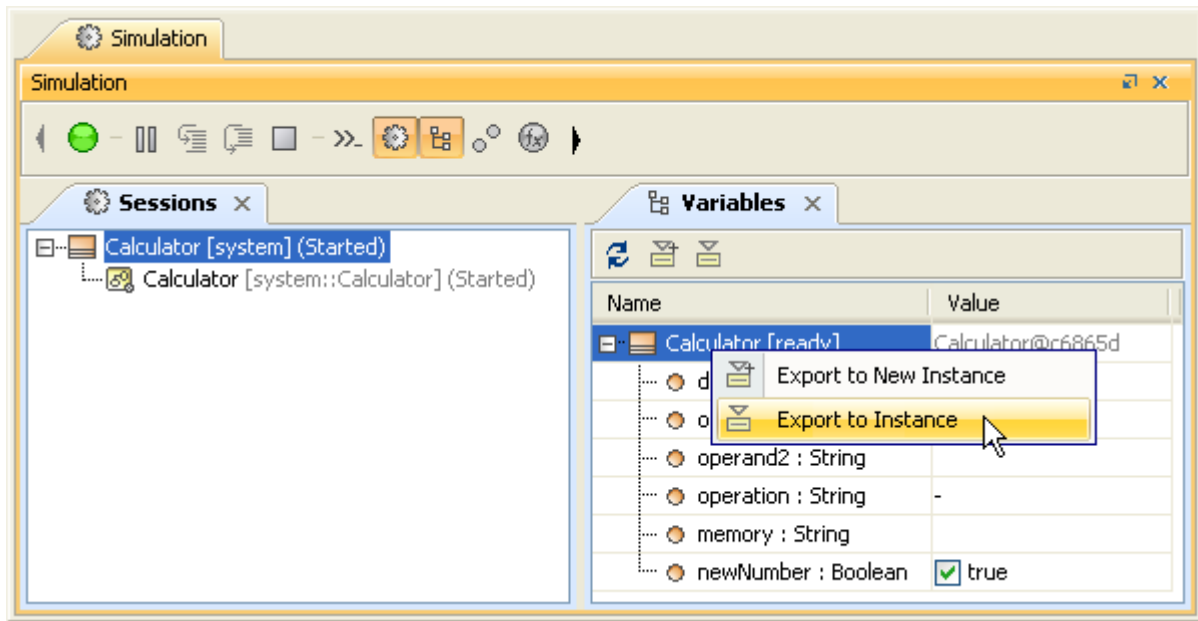


Figure 31 -- Export to Instance Context Menu

2. The **Select Instance** dialog will open. Select an InstanceSpecification that will be used to save the runtime object (you can select only the InstanceSpecification that has the same classifier as the runtime object) and click **OK** (Figure 32).

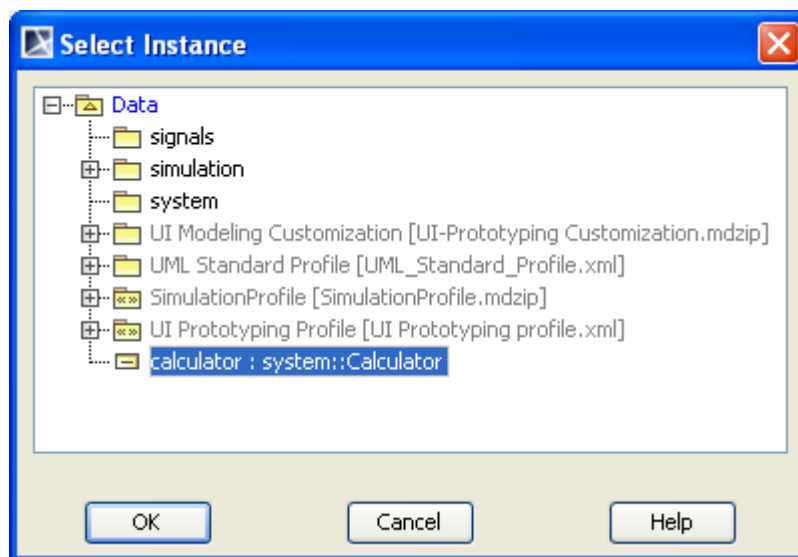


Figure 32 -- Selecting an InstanceSpecification in the Select Instance Dialog

5.5 Breakpoints

Cameo Simulation Toolkit allows you to add or remove breakpoints to or from model elements. The model execution will be paused when these model elements are activated during the execution. You can open the **Breakpoints pane** to see and manage all of the existing breakpoints in an active project. The **Breakpoints pane** lists all breakpoints with their properties shown in separate columns (Figure 33).

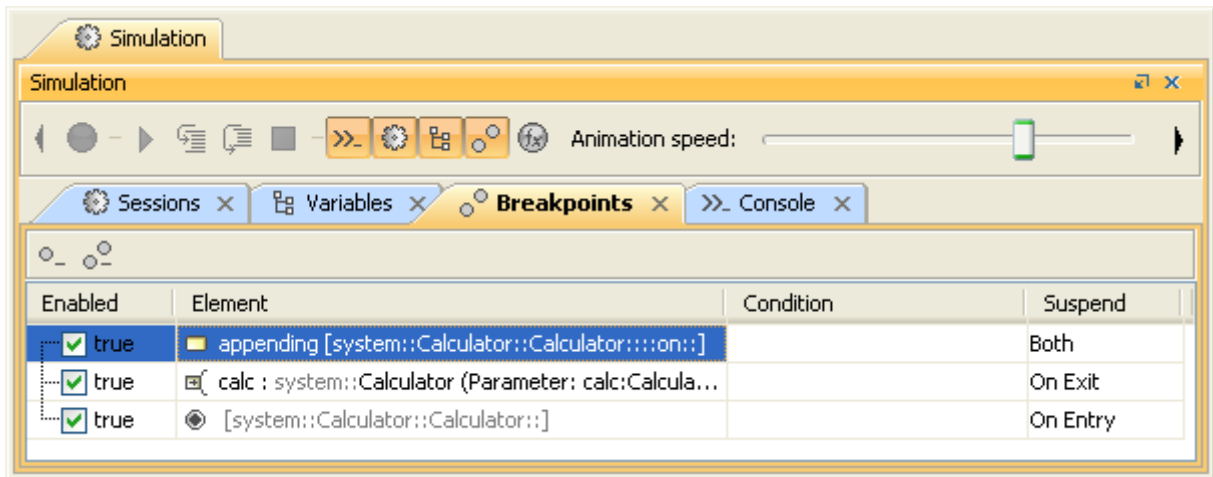


Figure 33 -- Breakpoints Pane

Table 5 -- Breakpoints Pane Columns

| Column | Function |
|------------------|---|
| Enabled | To display the enabled/disabled states of a breakpoint. If the value is true then the breakpoint is enabled. Otherwise, the breakpoint is disabled. The execution of a model will be suspended at that particular breakpoint only when the breakpoint is enabled (true). |
| Element | To represent a model element to which each breakpoint is applied. The execution of a model will be suspended when the symbol of the element is activated or deactivated (depending on the value in the Suspend column). |
| Condition | To represent a breakpoint condition, a boolean expression, that will be evaluated when the execution of a model reaches the element to which a breakpoint is applied. The execution will be suspended at that particular element/breakpoint when the result of the boolean expression is true. If the conditional is not defined, the execution will always be suspended when it reaches that particular breakpoint. |
| Suspend | There are three kinds of execution suspensions: (i) On Entry , (ii) On Exit , and (iii) Both . (i) On Entry: the execution of a model will be suspended when a breakpoint's element is activated. (ii) On Exit: the execution of a model will be suspended when a breakpoint's element is deactivated. (iii) Both: the execution of a model will be suspended on both activation and deactivation of a breakpoint's element. |

5.5.1 Adding Breakpoints

You can add a Breakpoint to a model element using the context menu.

To add a Breakpoint to a model element:

- Right-click a model element in either the containment browser or the symbol of the model element in a diagram, and then select **Simulation > Add Breakpoint(s)** (Figure 34).

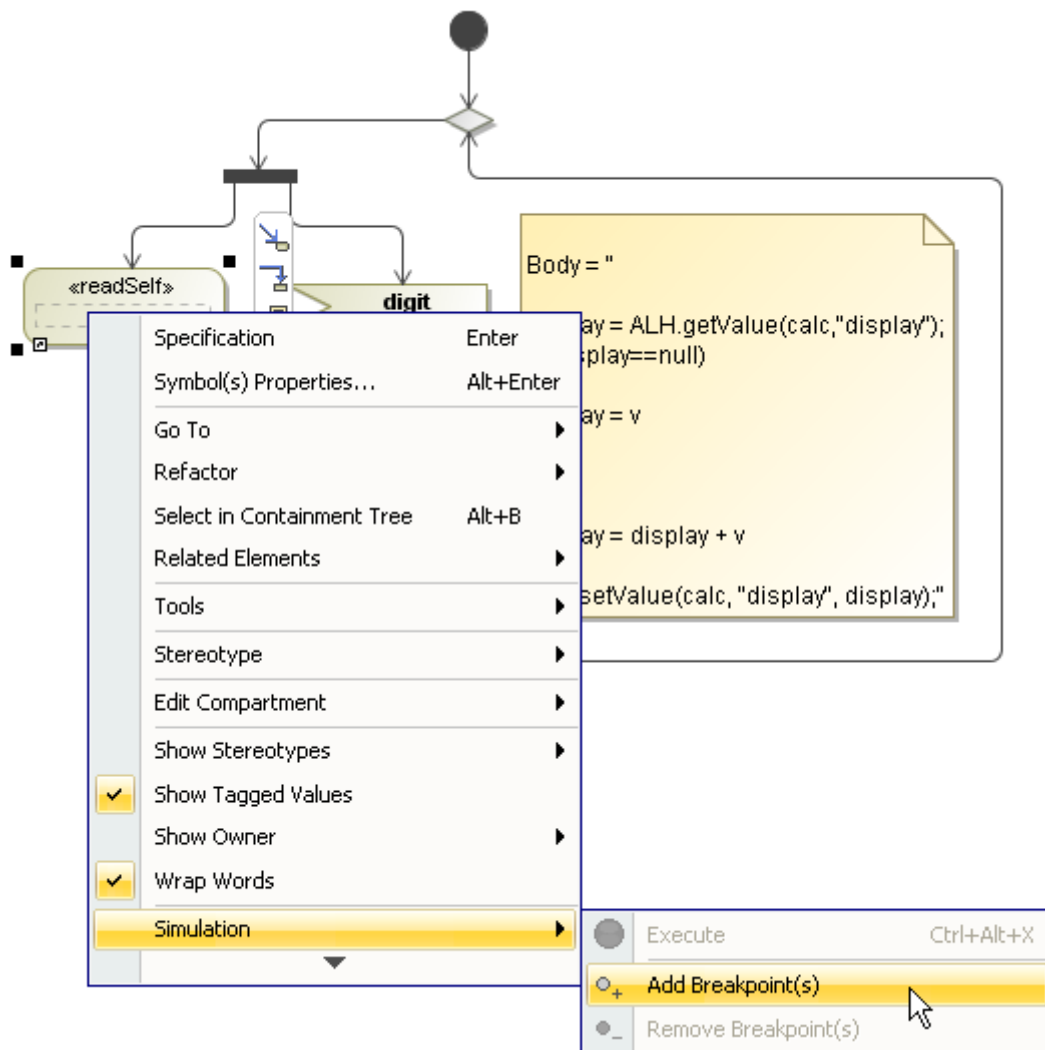


Figure 34 -- Adding a Breakpoint(s)

5.5.2 Removing Breakpoints

You can also remove a Breakpoint can be removed using the context menu.

To remove a Breakpoint:

- Right-click a model element that has a breakpoint(s) and select **Simulation > Remove Breakpoint(s)** (Figure 35).

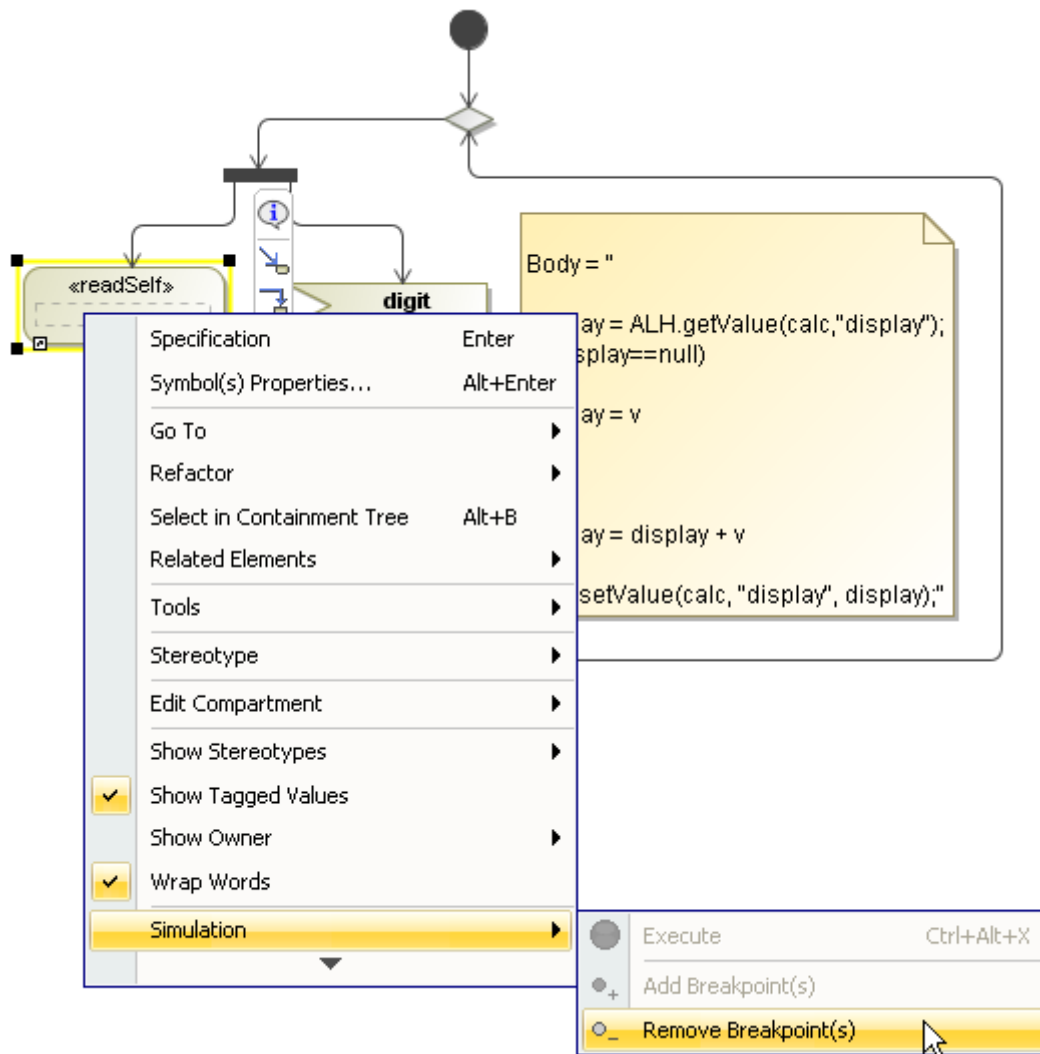


Figure 35 -- Removing a Breakpoint

You can also use the **Remove Breakpoint(s)** or **Remove All Breakpoints** toolbar button or the context menu of the **Breakpoints** pane to remove all of the existing breakpoints (Figure 36).

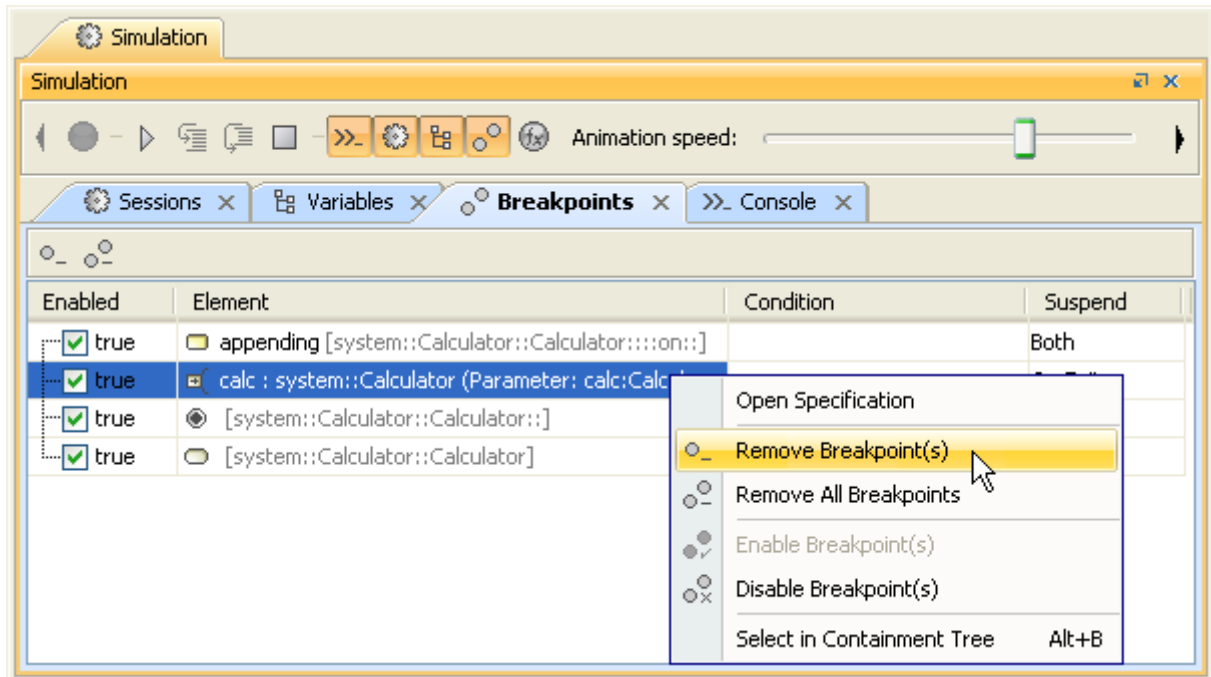


Figure 36 -- Removing Breakpoints Using the Breakpoints Pane Context Menu

6. Validation and Verification

Before executing your UML or SysML model, you need to make sure that it has been modeled correctly. Cameo Simulation Toolkit can help you validate a model against a set of validation rules before executing it.

To validate a model:

1. Click **Options > Environment** on the MagicDraw main menu to open the **Environment Options** dialog (Figure 37).

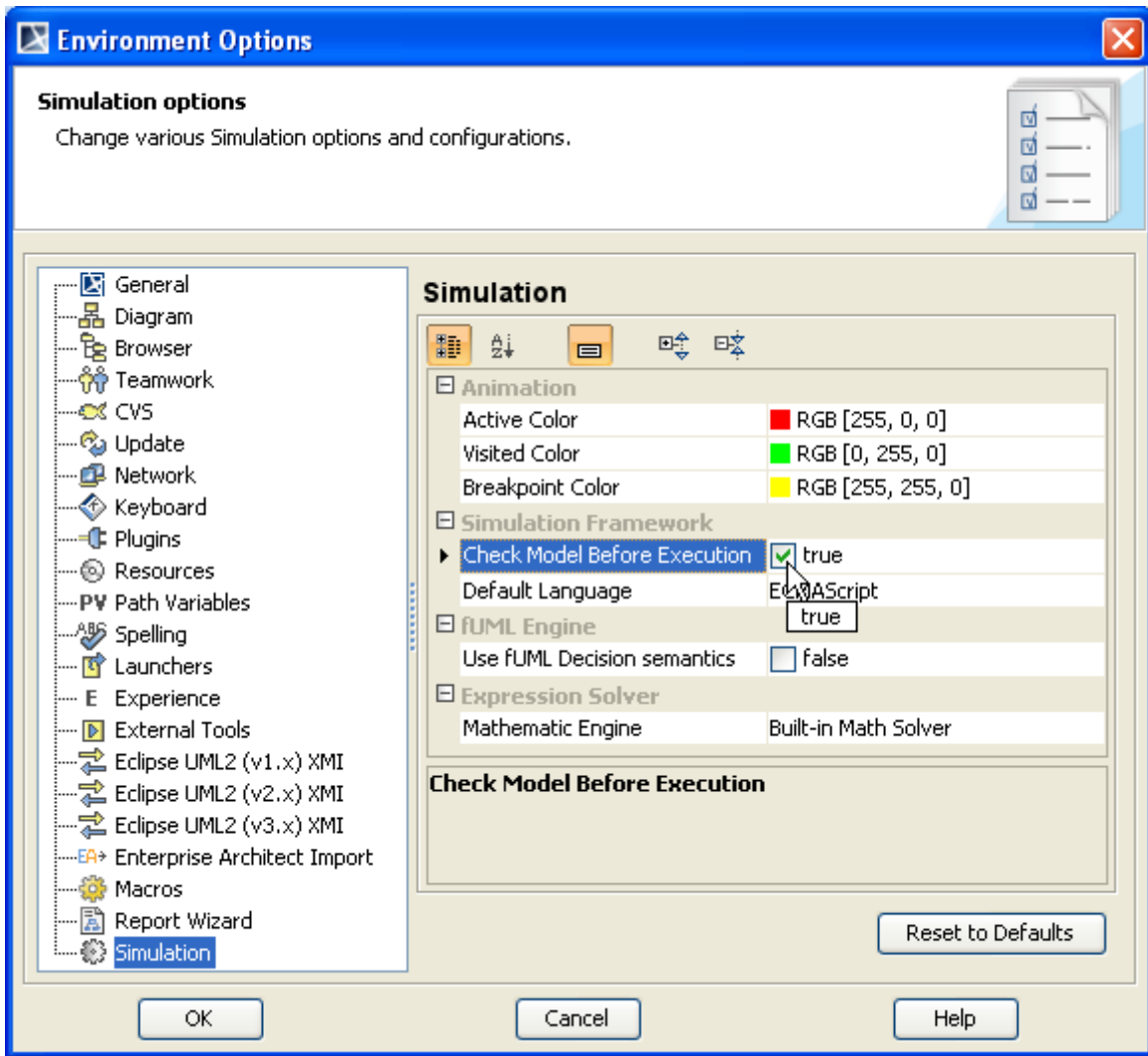


Figure 37 -- Checking Model before Execution in the Environment Options Dialog

2. Select the Simulation node on the left-hand side pane and select the **Check Model Before Execution** check box.
3. Click **OK**.
4. Execute your model. A dialog will open, asking whether you want to load the required profiles that contain the validation rules to validate your model (if your project does not contain the required validation rules) (Figure 38).

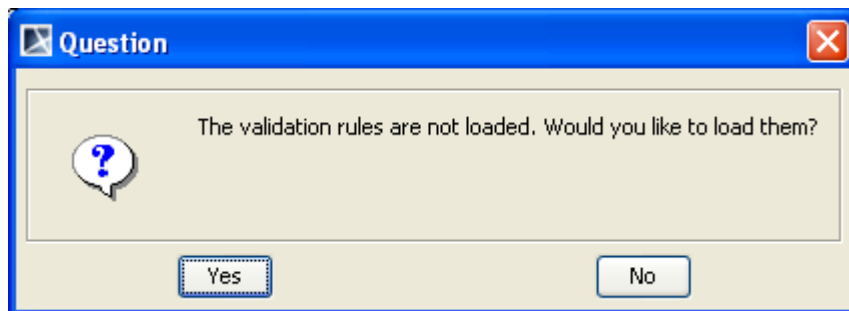


Figure 38 -- Dialog Prompting the Validation Rules

5. Click either (i) **Yes** to load the validation rules and validate the model before the execution or (ii) **No** to execute the model without validating it.

7. State Machine Simulation

Cameo Simulation Toolkit allows you to perform a State Machine Simulation (State Chart Simulation) on existing State Machine diagrams, based on the W3C SCXML standard. This kind of simulation is frequently used in the early state of software development by designers or analysts to test the flow of the software to be developed.

The W3C SCXML standard provides a generic state machine-based execution environment based on Harel statecharts. SCXML is capable to describe complex state machines, including sub-states, concurrency, history, time events, and many more. Most of the things that can be represented as a UML state chart such as business process flows, view navigation bits, interaction or dialog management, and many more, can leverage the SCXML engine.

With the state machine execution build, you can simulate an executable model both as a demonstration tool and to validate and verify the system behavior at key milestone reviews. In addition, Cameo Simulation Toolkit supports the UML state machine export to standard SCXML files for further analysis or transformations (through the state machine context menu).

7.1 Supported Elements

Most of the elements in a State Machine diagram are supported (Table 6).

Table 6 -- Supported Elements in StateMachine Diagram

| Element Type | Executable (Yes/No) | Exportable to SCXML (Yes/No) |
|--------------------|---------------------|------------------------------|
| state | Yes | Yes |
| composite state | Yes | Yes |
| orthogonal state | Yes | Yes |
| submachine state | Yes | Yes |
| initial state | Yes | Yes |
| final state | Yes | Yes |
| onEntry | Yes | Yes |
| onExit | Yes | Yes |
| onTransition | Yes | Yes |
| doActivity | Yes | Yes |
| time event | Yes | Yes |
| deep history | Yes | Yes |
| shallow history | Yes | Yes |
| transition-to-self | Yes | Yes |
| choice | Yes | Yes |

| | |
|-------------|--|
| NOTE | All of the elements in a State Machine diagram (to be executed) must have names. |
|-------------|--|

7.2 Adapting Models for State Machine Simulation

Currently, Cameo Simulation Toolkit can execute only the elements whose types specified in Section 7.1. Thus, you need to modify your model so that only the supported (executable) elements are included in your State Machine diagram.

7.2.1 Defining Trigger on Transition

A runtime object will change its state when it receives a trigger. Therefore, a transition should have a defined trigger. A trigger can be either a signal or time event.

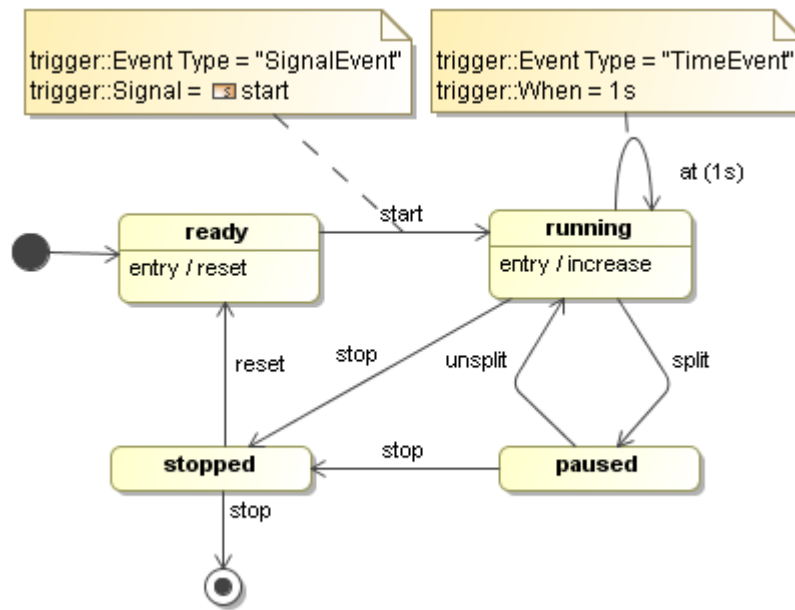


Figure 39 -- Triggers on Transitions (See Sample Project - Stopwatch.mdzip)

7.2.2 Using Guard on Transition

You can specify the guard conditions on transitions using any action language. Open **test_guard.mdzip** to see an example of how to specify guards on transitions.

You can use the properties of a context classifier (the classifier that is the context of a State Machine diagram) in guard expressions as variable names. The real values of the variables will be resolved at runtime. In **test_guard.mdzip**, the values come from the slots of the instance of the context classifier (see the instance diagram in the sample project).

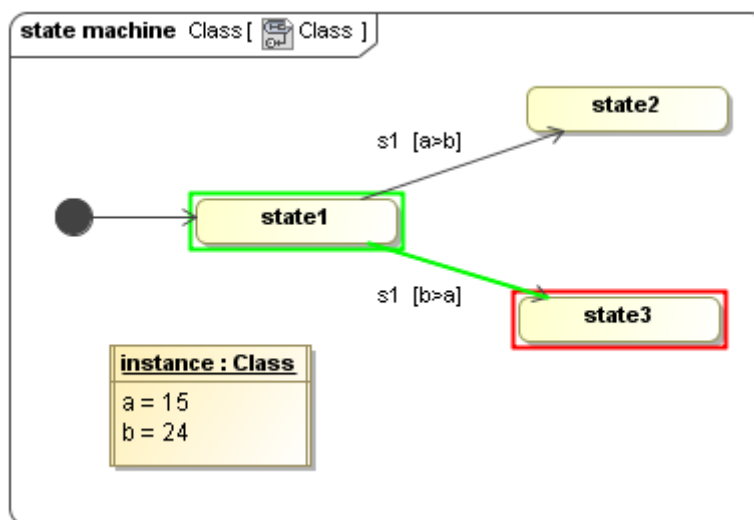


Figure 40 -- Test_guard Example

7.2.3 Behaviors on Entry, Exit, and Do Activity of State

States can have defined behaviors at Entry, Exit, or Do Activity. Cameo Simulation Toolkit will create a new simulation session to execute those defined behaviors. A defined behavior can be an Activity, State Machine, or OpaqueBehavior. The execution engine that corresponds to a defined behavior will be used to execute a model. If the defined behavior is OpaqueBehavior, the ScriptEngine will be used to execute the code in the body of OpaqueBehavior.

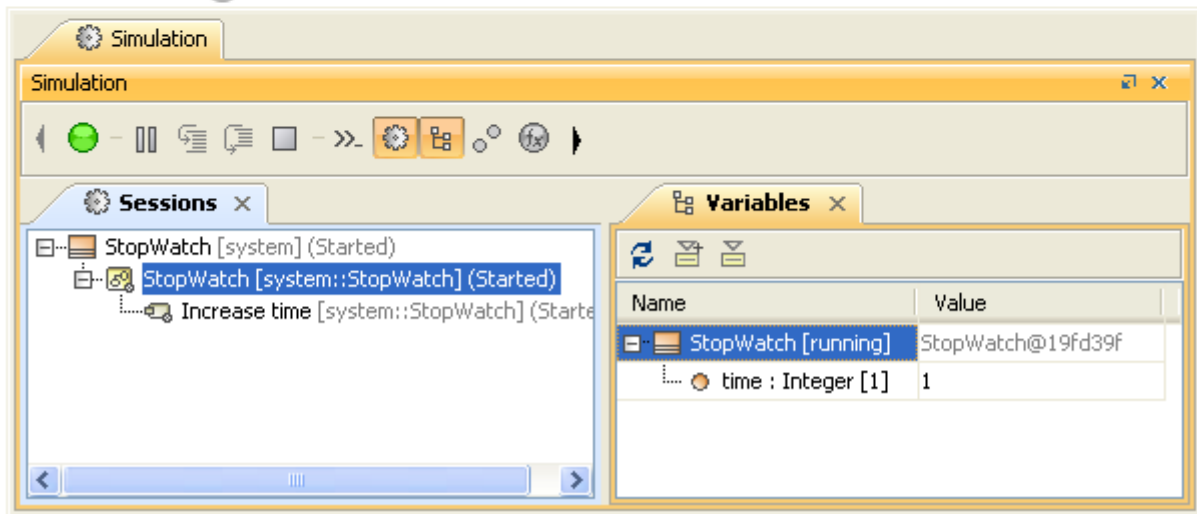
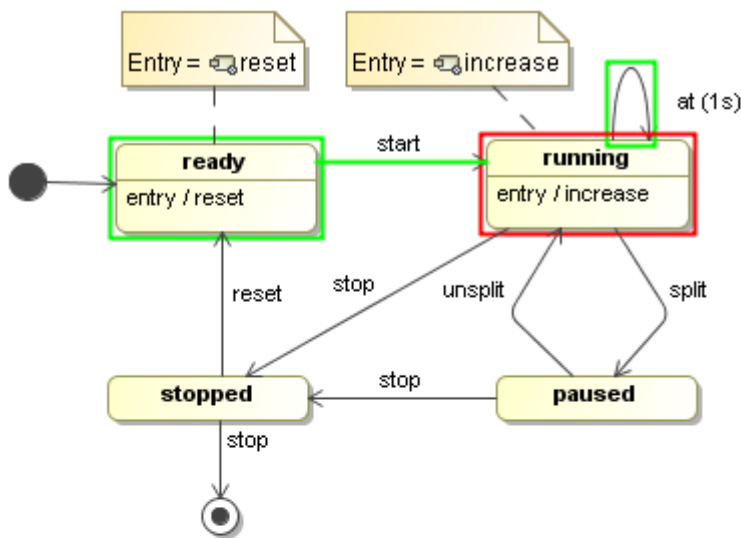


Figure 41 -- The Execution of Behavior on Entry of State (See Sample Project - Stopwatch.mdzip)

7.3 Running State Machine Execution

A state machine execution will be performed when the following elements are selected for the execution:

- State Machine
- State Machine diagram
- Class whose classifier behavior is defined by State Machine
- InstanceSpecification whose classifier is a Class that has a defined classifier behavior with State Machine

During a state machine execution, the state of a runtime object will be changed by a trigger. The trigger can be either a signal or time event. If it is a signal event trigger, the signal can be sent to a runtime object to trigger it from one state to another. To send the trigger signal, you have to select the runtime object, which is the target recipient for the signal, in the **Variables Pane**. All signals that can be received by a selected runtime object will be listed in the **Triggers** drop-down menu on the **Simulation Window** toolbar (Figure 42).

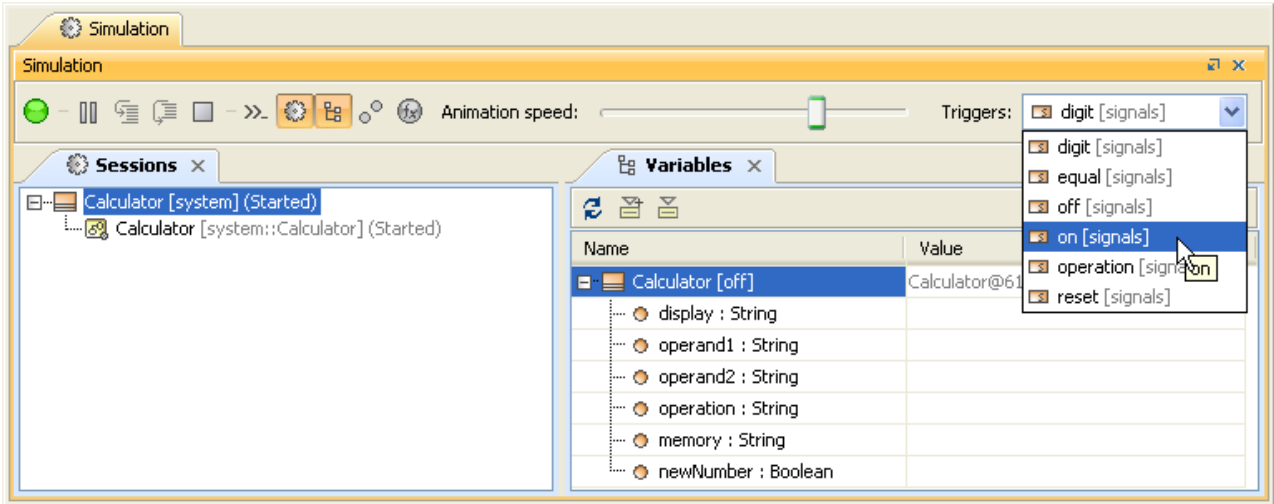


Figure 42 -- Triggers Drop-down Menu

A signal can be sent to a runtime object through a Mock-up User Interface. See more information about Mock-up in Section 3.4 UI Modeling Diagram Execution.

7.4 Sample Projects

The State Machine Simulation sample projects are available in the `<md.install.dir>/samples/simulation/tests` directory. The sample projects include:

- 7.4.1 test_regions.mdzip
- 7.4.2 test_timers.mdzip
- 7.4.3 test_guard.mdzip

7.4.1 test_regions.mdzip

This sample demonstrates the use of an orthogonal state with parallel regions, and entry or exit activities.

- An Entry activity will be executed right after a state has been activated before any other states in the inner regions.
- All of the initial states in all regions will be activated at the same time. It demonstrates multiple active states at the same time.
- The events list in Simulation Console contains the triggers of all outgoing transitions of all active states.
- If one of the parent's state outgoing transitions is triggered, an exit activity will be executed before the state is deactivated.

7.4.2 test_timers.mdzip

This sample demonstrates the implementation of timing events in State Machine.

- The transitions with specified time events will be automatically triggered after a specified amount of time (in seconds or milliseconds).
- Only relative time (delays) are supported.

7.4.3 test_guard.mdzip

This sample demonstrates the ability to specify and resolve guard conditions on the transitions.

- The properties of a context classifier can be used in the expressions as variable names.
- The real values of the variables will be resolved at runtime.
- If this is the case, they come from the slots of the instance of the context classifier (see the Instance diagram).

8. Activity Simulation

8.1 About Activity Execution Engine

Cameo Simulation Toolkit provides the Activity Execution Engine that allows you to perform an Activity Simulation (Execution) on Activity Diagrams or Activity Elements. Cameo Simulation Toolkit includes the implementation of OMG Semantics of a Foundational Subset for Executable UML Models (fUML), which is an executable subset of standard UML, that can be used to define the structural and behavioral semantics of systems. fUML defines a basic virtual machine for the Unified Modeling Language, and the specific abstractions supported, enabling compliant models to be transformed into various executable forms for verification, integration, and deployment.

Various UML activity diagram concepts are supported, including object and control flows, behavior and operation calls, sending and accepting signals and time events, pins, parameters, decisions, structured activity nodes, and many more.*

The Activity Execution Engine features include:

- fUML 1.0 specification support
- Any action languages in opaqueBehaviors, opaqueExpressions, decisions, guards, constraints (see 11.4 Using MATLAB® as a Mathematical Solver for more details)
- CallBehaviorAction with nested diagrams execution and animation
- SendSignalAction can be used to send signal to global event queue to be consumed by any other engine (for example, state machine)

NOTE

- Activities that will be executed must be owned in a Package or Class only. As a workaround, the CallBehavior actions, owned by the call behaviors in a package, will be used for the entry/do/exit behaviors in states.
- The guards on an ObjectFlow are not boolean expressions in fUML. They contain a value that should match with the runtime value that flows on the ObjectFlow during execution. You can change this mode to a regular UML (boolean expression) by changing the Environment Options-Simulation-Use fUML Decision Semantics value. The value is false (UML mode) by default.

8.2 Creating Model for Activity Execution

You can simulate a UML activity or a classifier whose classifier behavior is defined by an activity. This section will demonstrate how to create a simple, executable Activity model by using the following steps:

- (i) Create a class containing two properties typed by Integers.
- (ii) Create an activity to print the summation value of the two properties.
- (iii) Assign the activity as the classifier behavior of the created class.
- (iv) Create an opaque behavior to print the summation value of two input parameters of type **Integer**.
- (v) Write a script to print the summation of the given integer values that are referred to by the two input parameters.
- (vi) Complete the activity diagram of the class.
- (vii) Create a **ReadSelfAction** to read a runtime object that will be supplied to the input pins of both **readX** and **readY** actions.
- (viii) Create an **InstanceSpecification** and assign the values to the slots that correspond to the two created properties.

(i) To create a class containing two attributes typed by Integers:

1. Create a new UML project by clicking **File > New Project...** on the main menu. The **New Project** dialog will open (Figure 43).

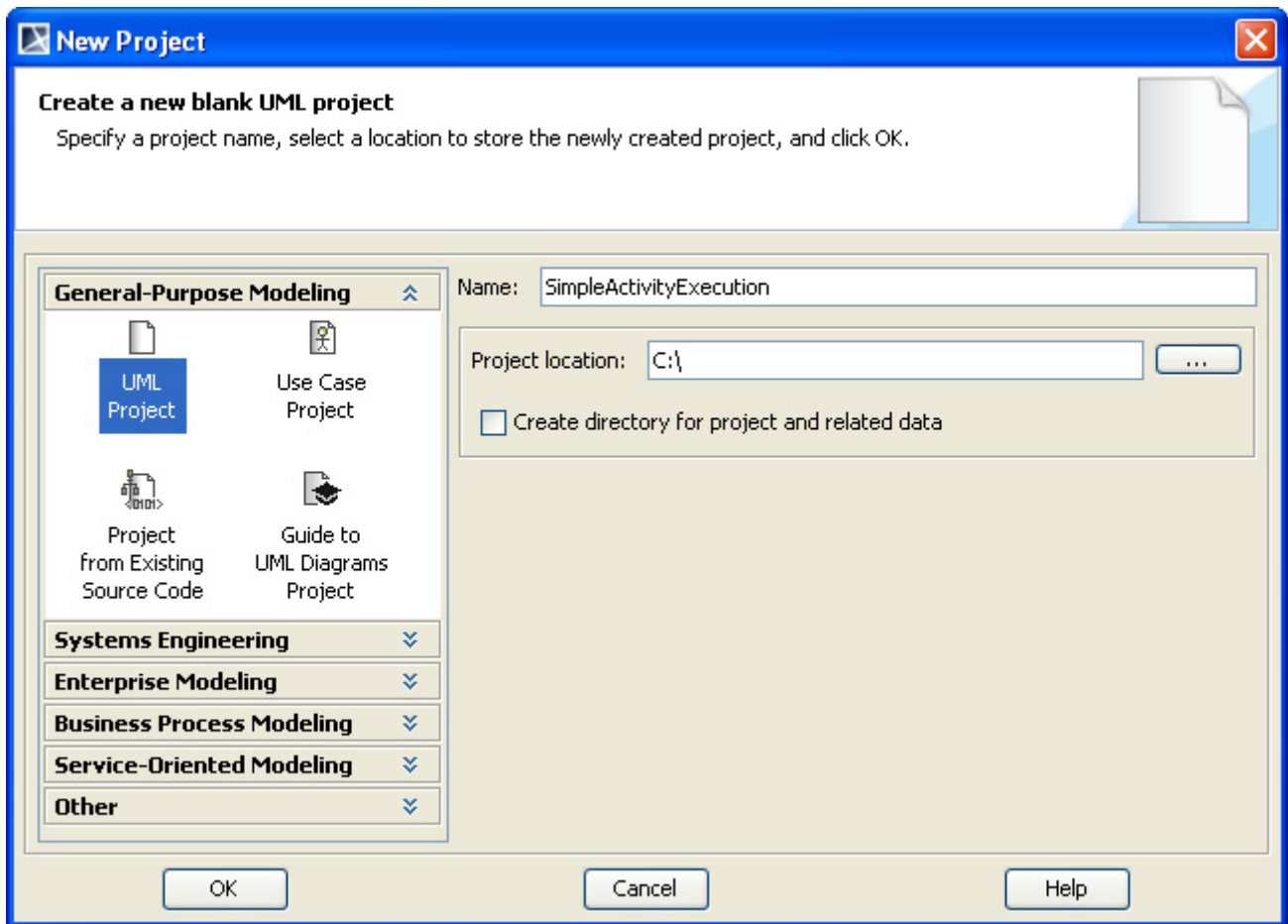


Figure 43 -- New Project dialog

2. Select **UML Project** from the **General-Purpose Modeling** group and specify the project's name, for example, "SimpleActivityExecution".
3. Specify the location where you want to save your project file, and then click **OK**.
4. Right-click the **Data** model in the containment browser and select **New Element > Class**. A new class element, which is the context of the activity, will be created in the containment browser. Name the created class, for example, "SumPrinter".
5. Add two properties: (i) **x** and (ii) **y** of type **Integer**.

- (i) Right-click the **SumPrinter** class and select **New Element > Property**. Type 'x' to name the property (Figure 44). Right click **x** and select **Specification** to open its **Specification** dialog. Select **Integer** as the property type (Figure 45).



Figure 44 -- Creating New Property 'x' for SumPrinter

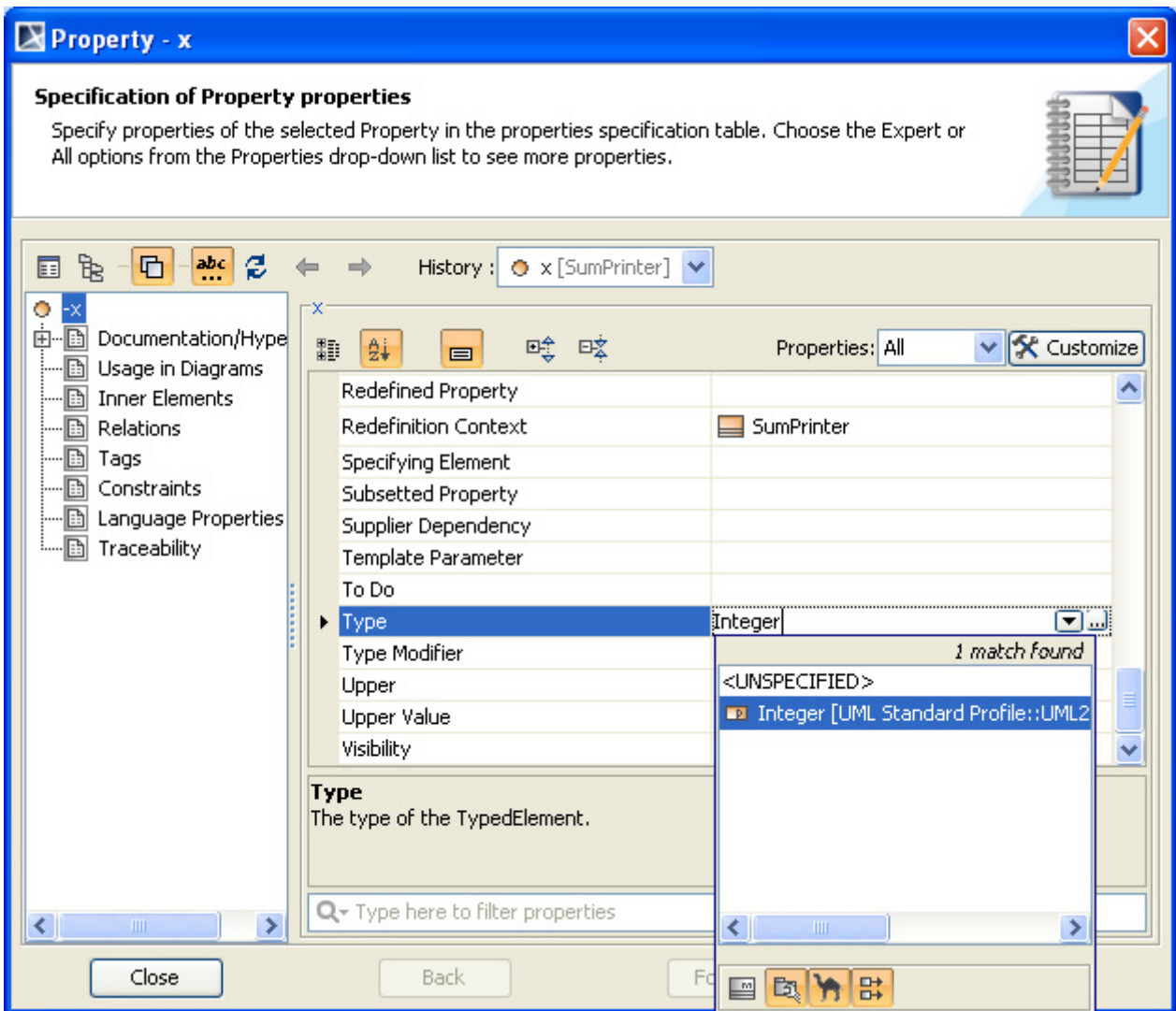


Figure 45 -- Selecting Property Type

(ii) Repeat the steps to create property **y** (Figure 46).



Figure 46 -- **SumPrinter** Class with Properties **X** and **Y** of **Integer** Type

Once the properties **x** and **y** have been created, define the behavior of the created class by specifying the classifier behavior of the **SumPrinter** class with a UML Activity element.

(ii) To create an activity to print the summation value of the two properties:

1. Right click the **SumPrinter** class in the containment browser and select **New Diagram > Activity Diagram** to create a new Activity under it.
2. Name the diagram "PrintSum".

Now that the activity has been created, assign it as the classifier behavior of SumPrinter.

(iii) To assign the activity as the classifier behavior of the created class:

1. Right-click the **SumPrinter** class in the containment browser and select **Specification** to open its **Specification** dialog (Figure 47).

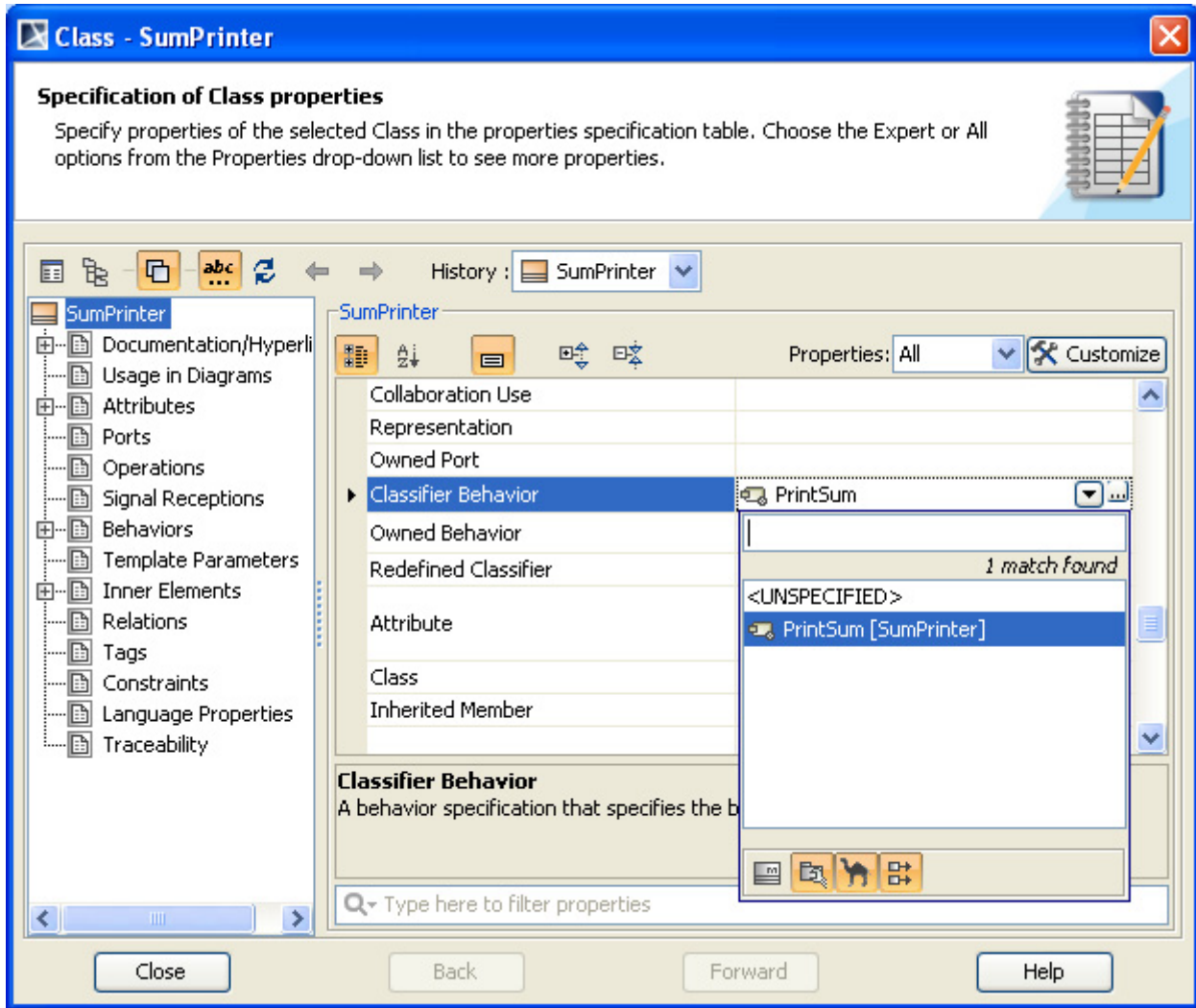


Figure 47 -- Assigning the Classifier Behavior of **SumPrinter**

2. Select **All** from the **Properties** drop-down menu to make sure that all of the properties are listed in the dialog.
3. Click **Classifier Behavior** and select the **PrintSum** activity.

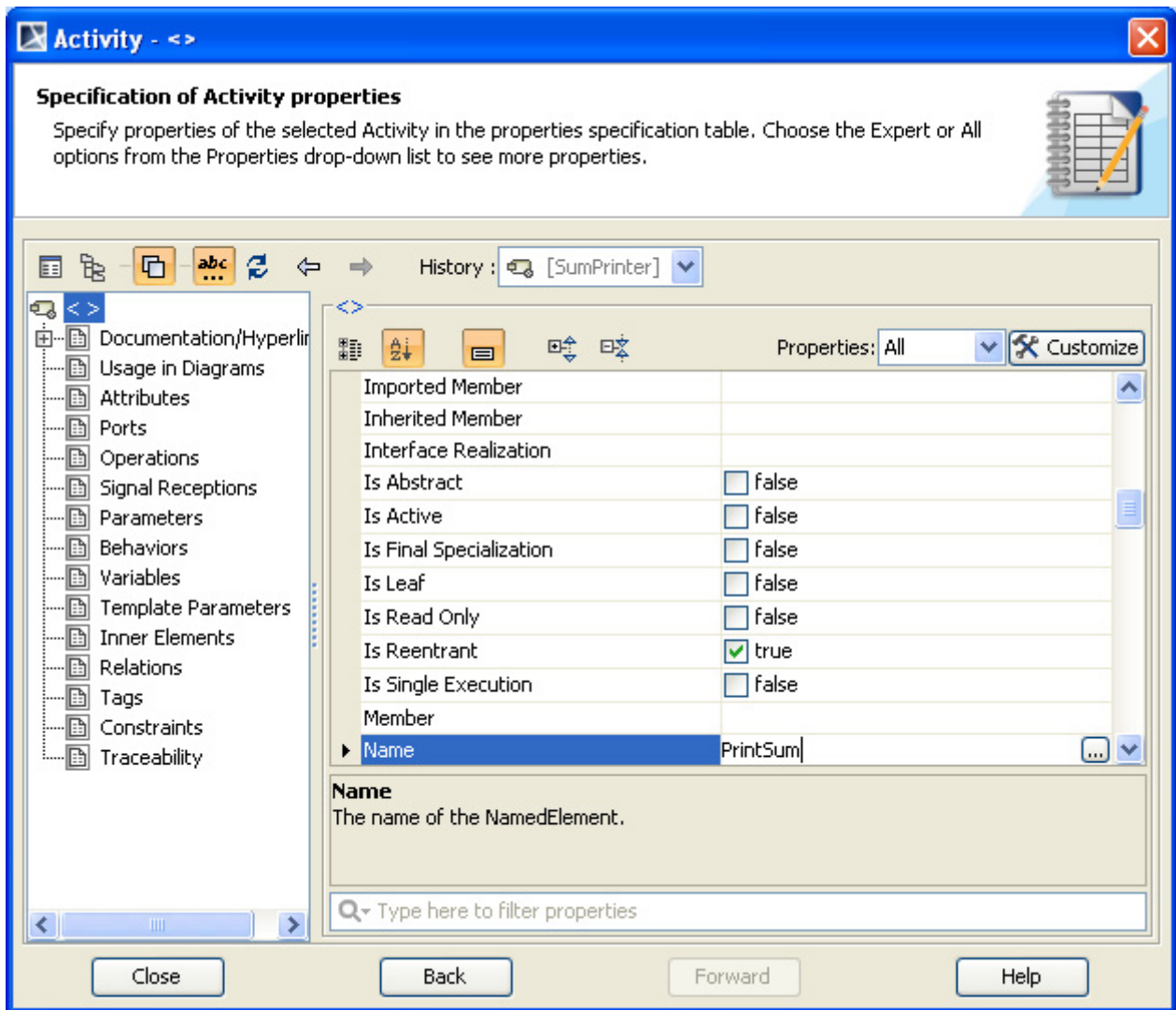


Figure 48 -- Specification Dialog of **PrintSum** Activity

(iv) To create an opaque behavior to print the summation value of the two input parameters of type **Integer**:

1. Right-click the **Data** model in the containment browser and select **New Element > Opaque Behavior**. A new opaque behavior will be created under the **Data** model.
2. Name it "PrintSumOfIntegers" (Figure 49).

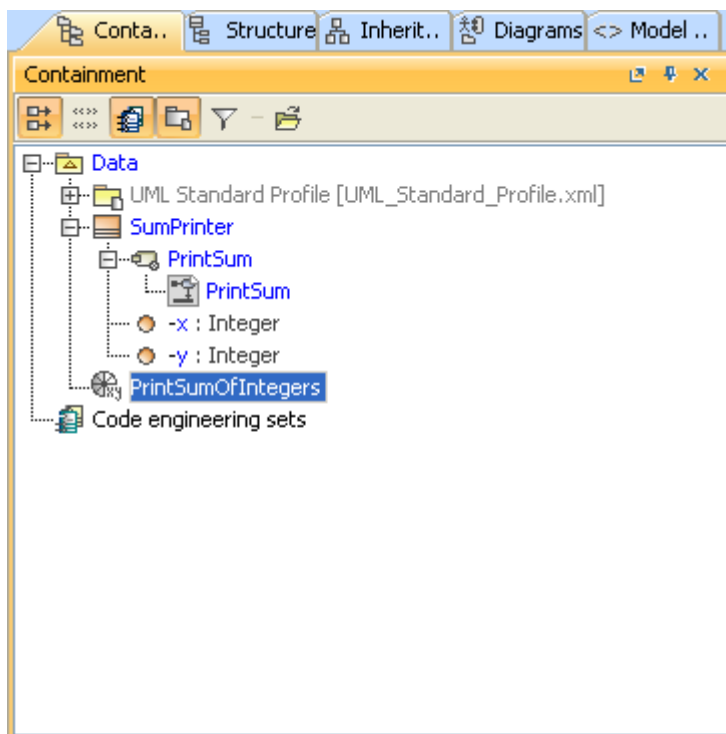


Figure 49 -- *PrintSumOfIntegers* Opaque Behavior in the Containment Browser

3. Add two input parameters of type **Integer**: (i) **a** and (ii) **b**.

- (i) Right-click the **PrintSumOfIntegers** opaque behavior and select **New Element > Parameter**. Name the created parameter 'a' and select **Integer** as the type of parameter **a** (Figure 50).

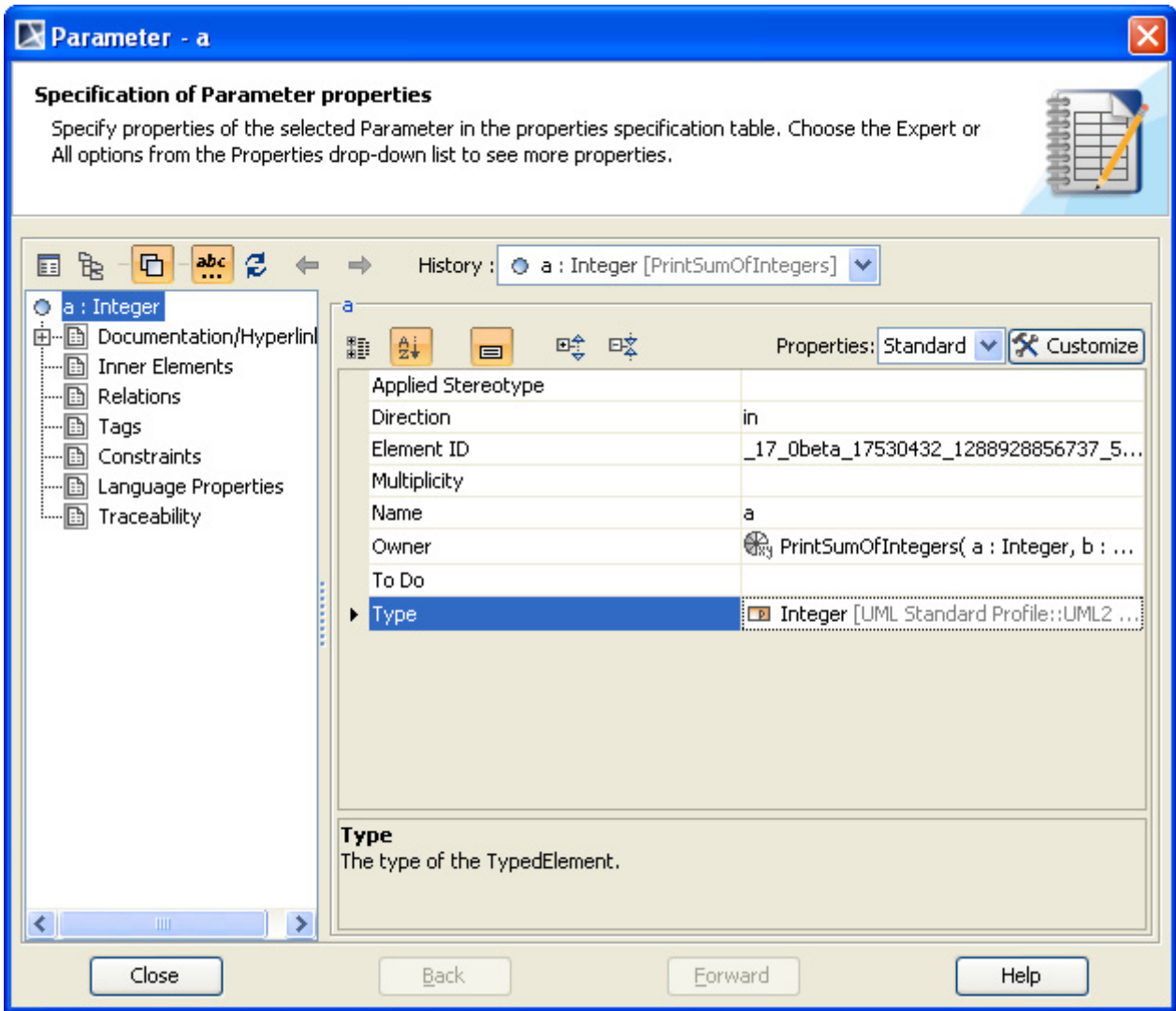


Figure 50 -- Specification Dialog of Parameter a

(ii) Repeat the steps to create parameter b (Figure 51).

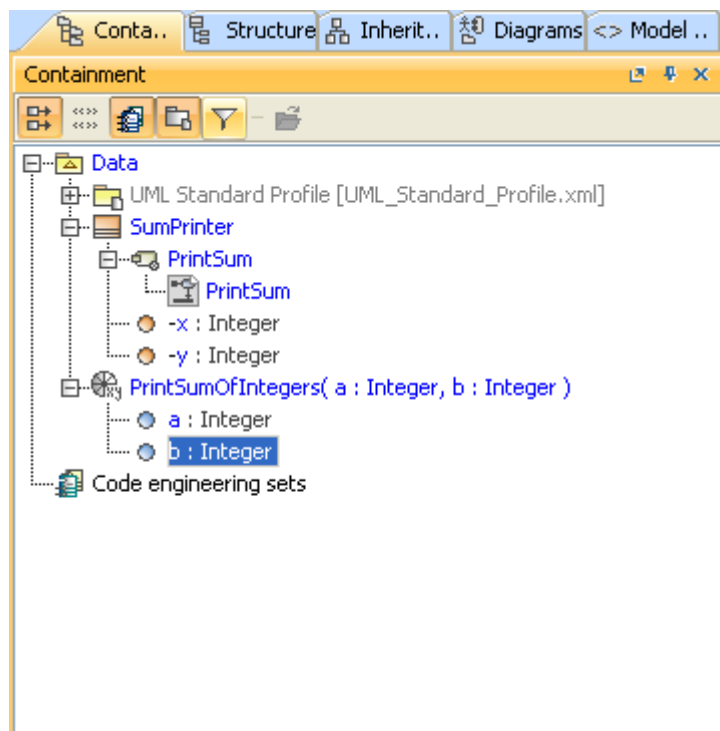


Figure 51 -- *PrintSumOfIntegers* Opaque Behavior Containing Parameters *a* and *b* in the Containment Browser

(v) To write a script to print the summation of the given integer values:

- Open the specification dialog of the **PrintSumOfIntegers** opaque behavior and write a script in the **Body** field (you can use any scripting language that is supported by MagicDraw's macro engine, for example, BeanShell, Groovy, JavaScript, Jython, or Jruby). In this example, JavaScript will be used to print the summation of the given integer values that are referred to by the parameters **a** and **b**; therefore, the script will be: "print(a+b)" (Figure 52).

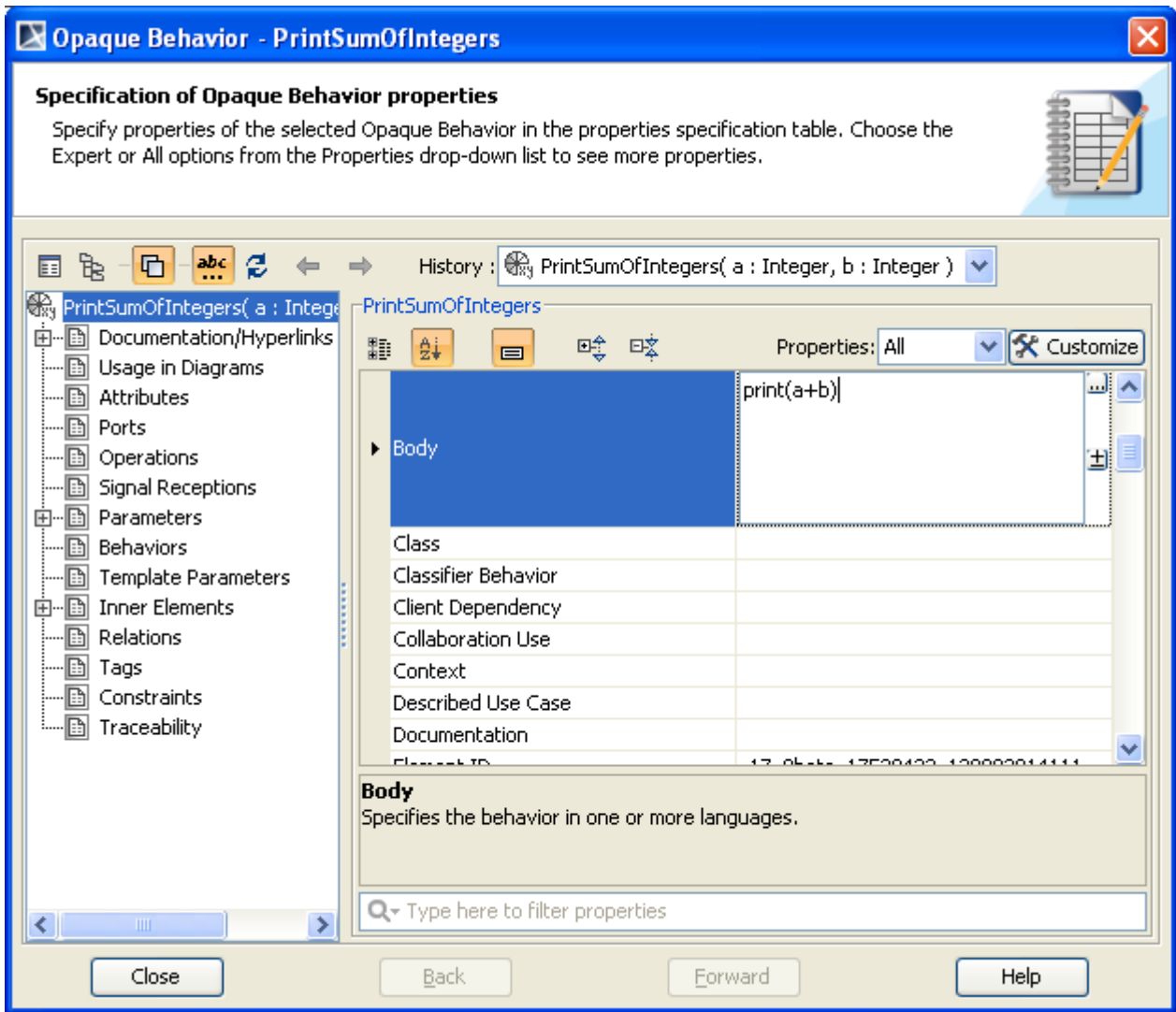
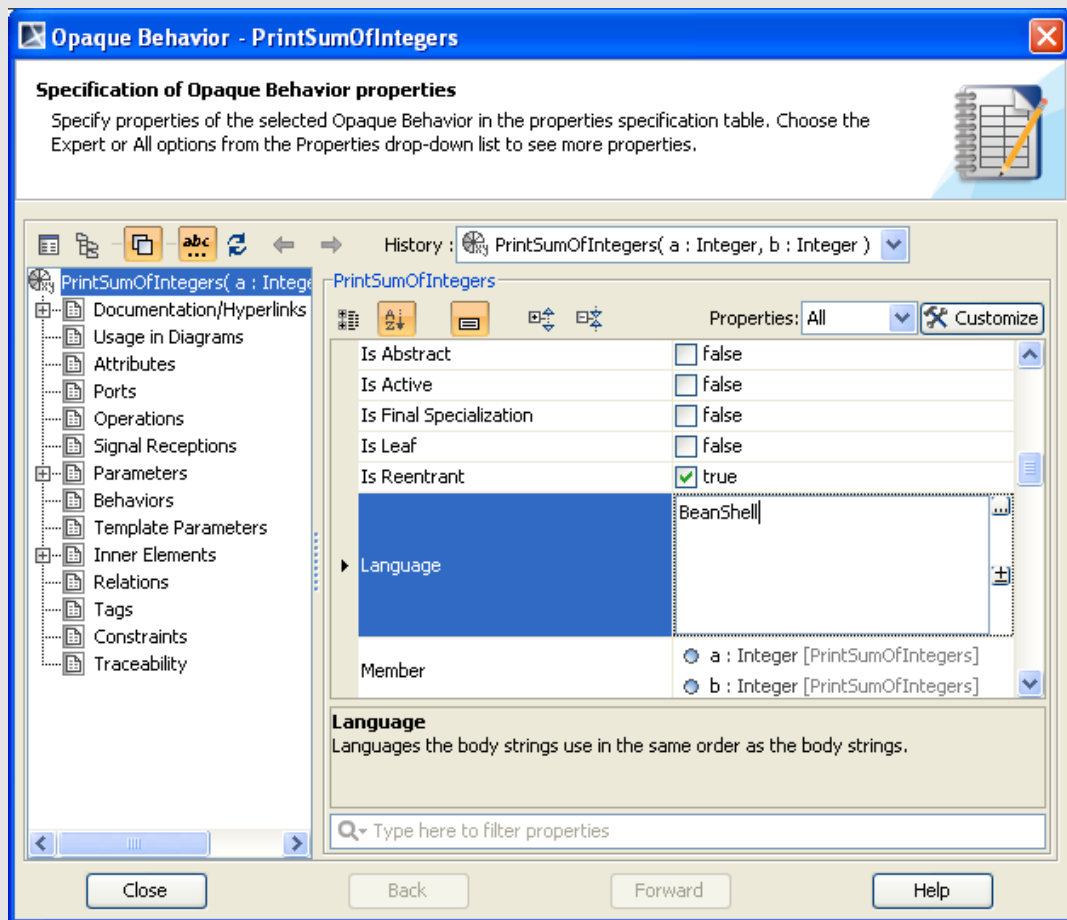


Figure 52 -- JavaScript for Printing the Summation of Integer Values

NOTE If you want use a script language other than JavaScript, specify it in the Language attribute.



The next steps will be to complete the **PrintSum** activity diagram of the **SumPrinter** class and add **ReadStructuralFeatureAction** so that the values of properties **x** and **y**, which are owned by the **SumPrinter** class, can be read. The values of **a** and **b** will later be passed on to the **PrintSumOfIntegers** opaque behavior as the values of input parameters **a** and **b** respectively.

(vi) To complete the activity diagram of the class:

1. Drag the **PrintSumOfIntegers** opaque behavior from containment browser to the **PrintSum** activity diagram. A new action of **PrintSumOfIntegers** will be created.
2. Name the action "print" (Figure 53).



Figure 53 -- Creating a Print Action by Dragging the PrintSumOfIntegers Opaque Behavior to Activity Diagram

3. Add Initial and Activity Final nodes to the activity diagram and connect them to the print action using a control flow (Figure 54).



Figure 54 -- Activity Diagram with Initial and Final Activity Nodes

4. Click **Action** and select the **Any Action...** diagram toolbar button on the **PrintSum** activity diagram.

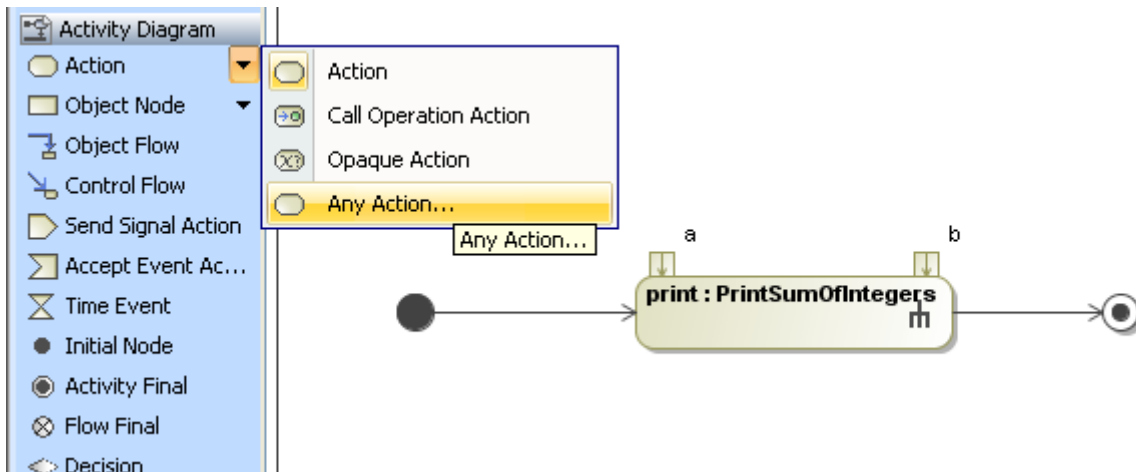


Figure 55 -- Selecting Any Action Diagram Toolbar Button

5. Select **ReadStructuralFeatureAction** in the **Select Action Metaclass** dialog and click **OK** (Figure 56).

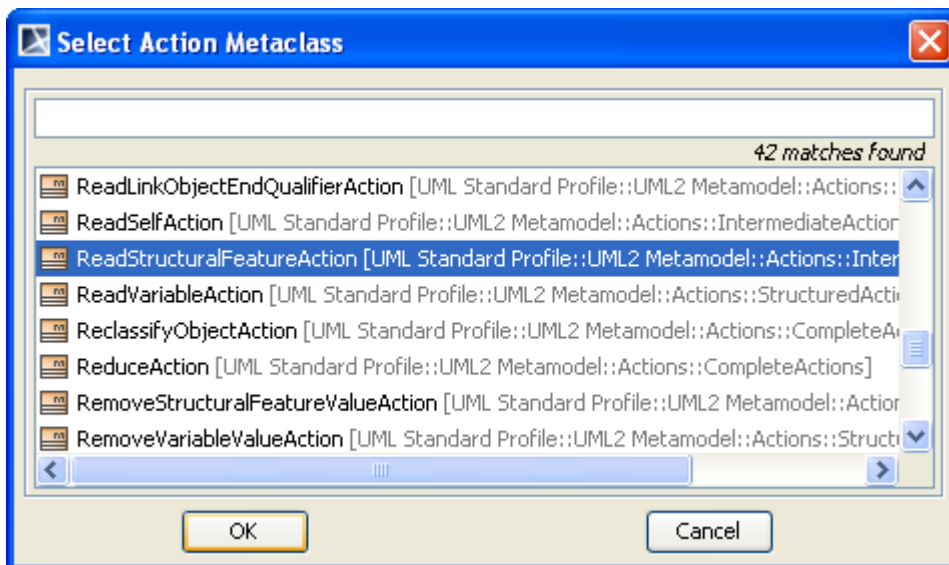


Figure 56 -- Selecting ReadStructuralFeatureAction in the Select Action Metaclass Dialog

- Click the **PrintSum** activity diagram to create the action and name it “readX” (Figure 57).

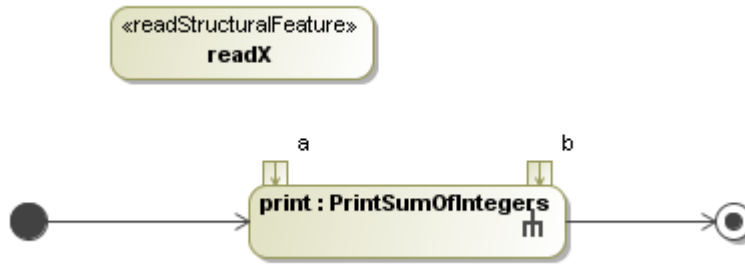


Figure 57 -- Activity Diagram with readX Action

- Open the **Specification** diagram of the **readX** action (Figure 58).

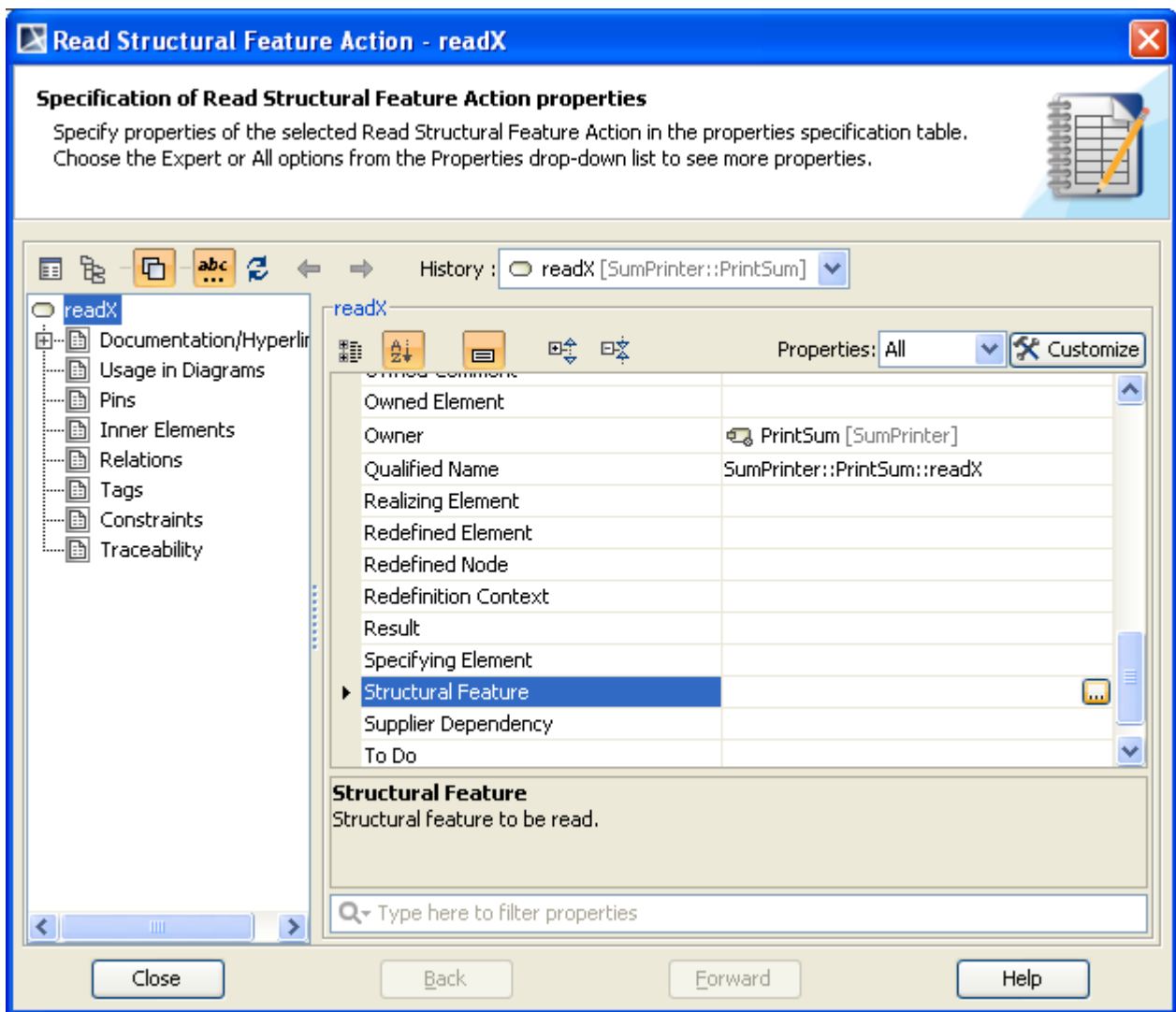


Figure 58 -- The Specification Dialog of readX ReadStructuralFeatureAction

- Click the **Structural Feature ...** button to open the **Select Property** dialog to select the structural feature (Figure 59).

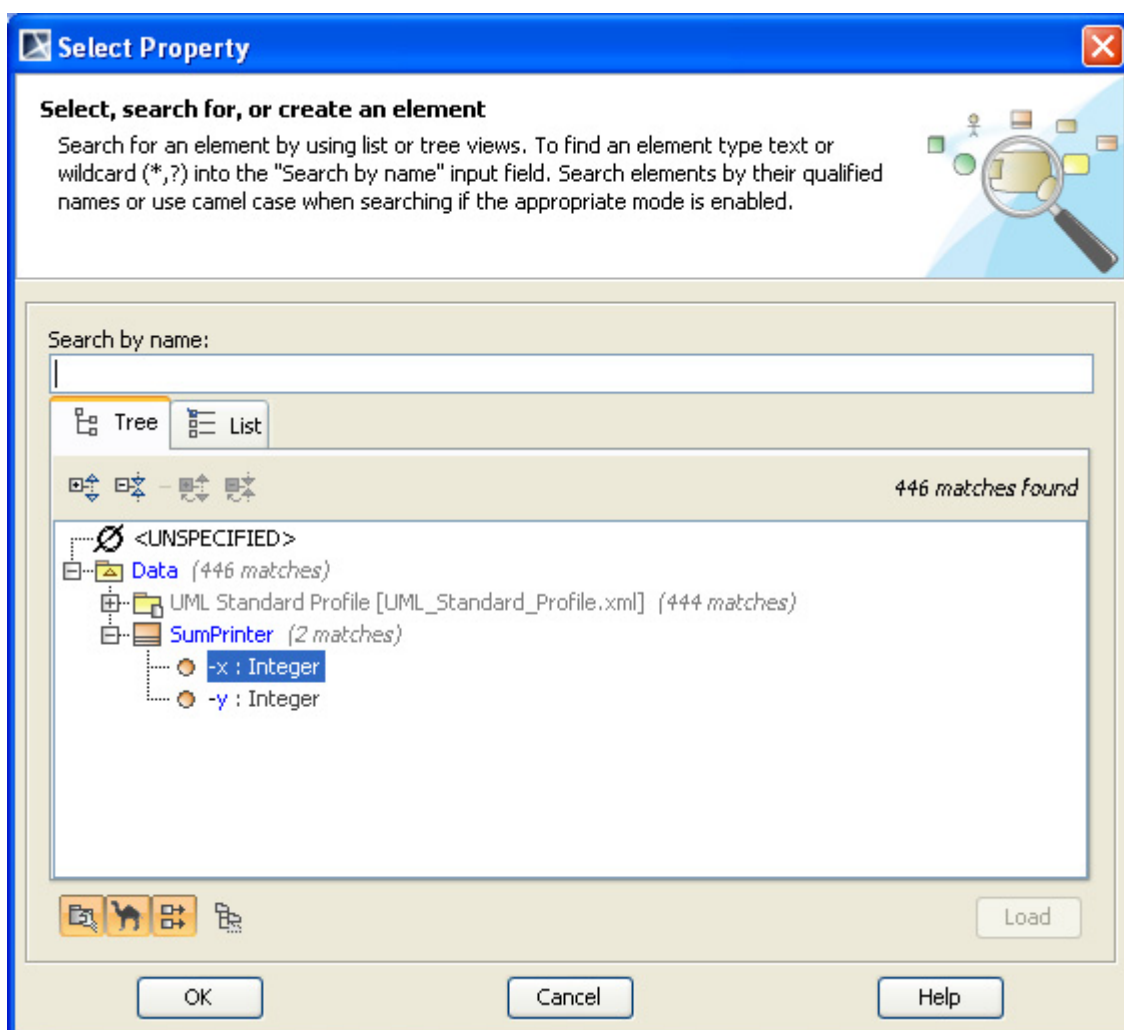


Figure 59 -- .Select Property dialog for select the property x

9. Select property **x** of the **SumPrinter** class and click **OK**. The **Select Property** dialog will close.

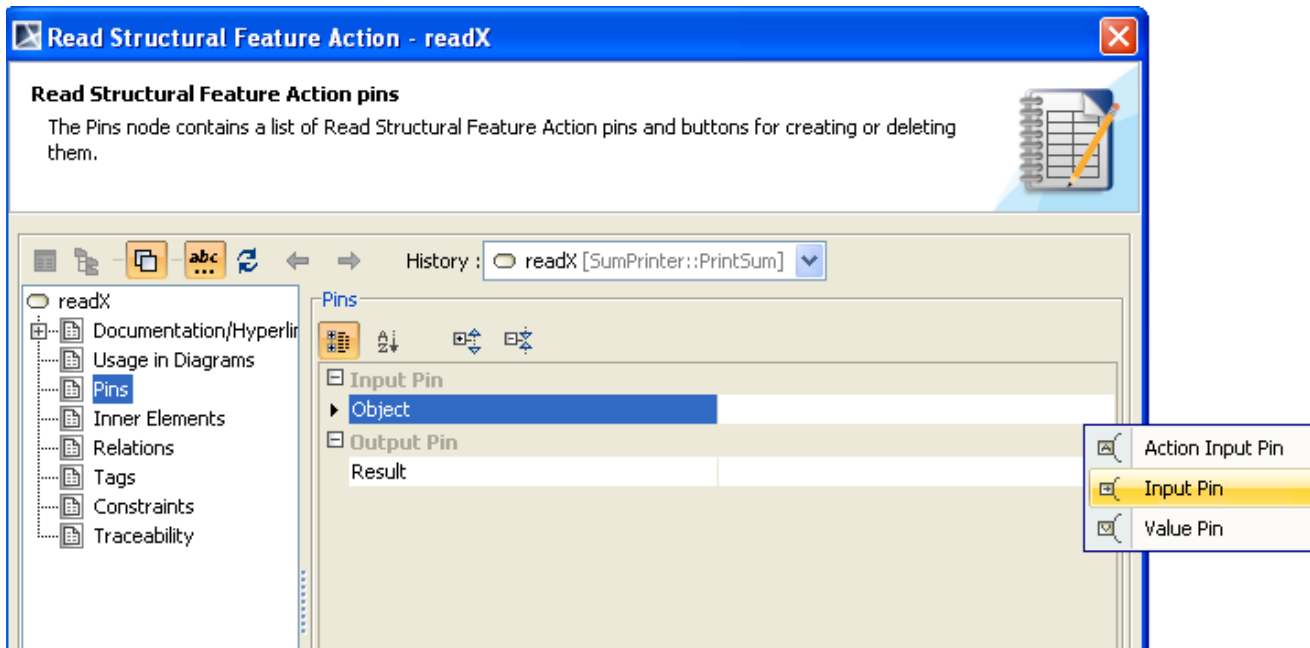


Figure 60 -- Add new input pin to readX action

10. Select **Pins** on the left-hand side pane of the **Specification** dialog. You need to create two pins for **ReadStructuralFeatureAction**: (i) the input pin to specify the runtime object of type **SumPrinter** whose runtime values that correspond to the properties **x** and **y** will be used for execution, and (ii) the output pin of the type **Integer** to specify the value read from the structural feature.
 - (i) Click the **Object** button and select **Input Pin** from the context menu (Figure 60). A new input pin will be added to the action. Name this pin "self" and select **SumPrinter** as its type, and then click the **Back** button (Figure 61).

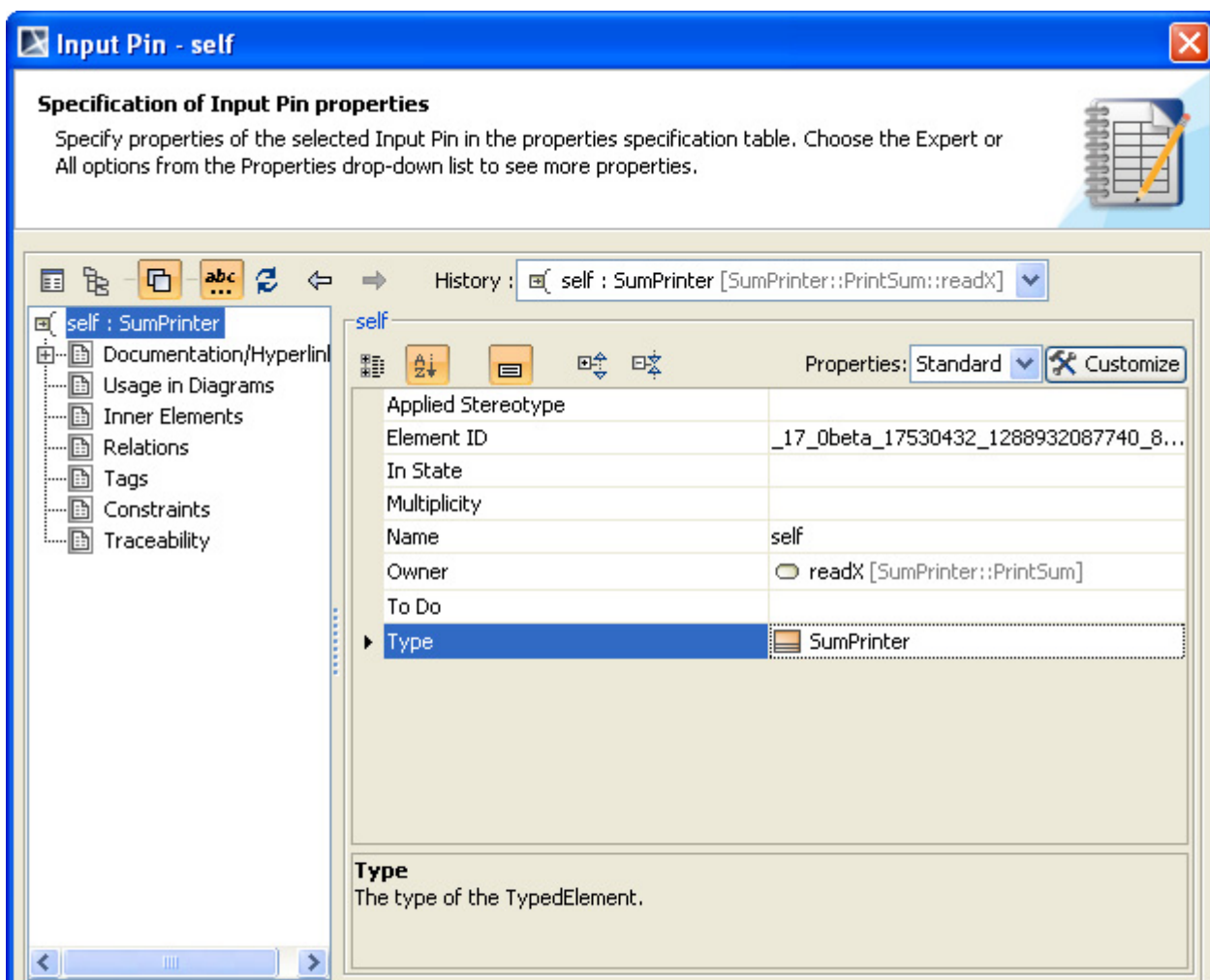


Figure 61 -- Naming the Input Pin and Selecting Its Type

- (ii) Click the **Result** button and select **Output Pin** from the context menu (Figure 62). Name this pin "a" and select **Integer** as its type, and then click the **Close** button (Figure 63).

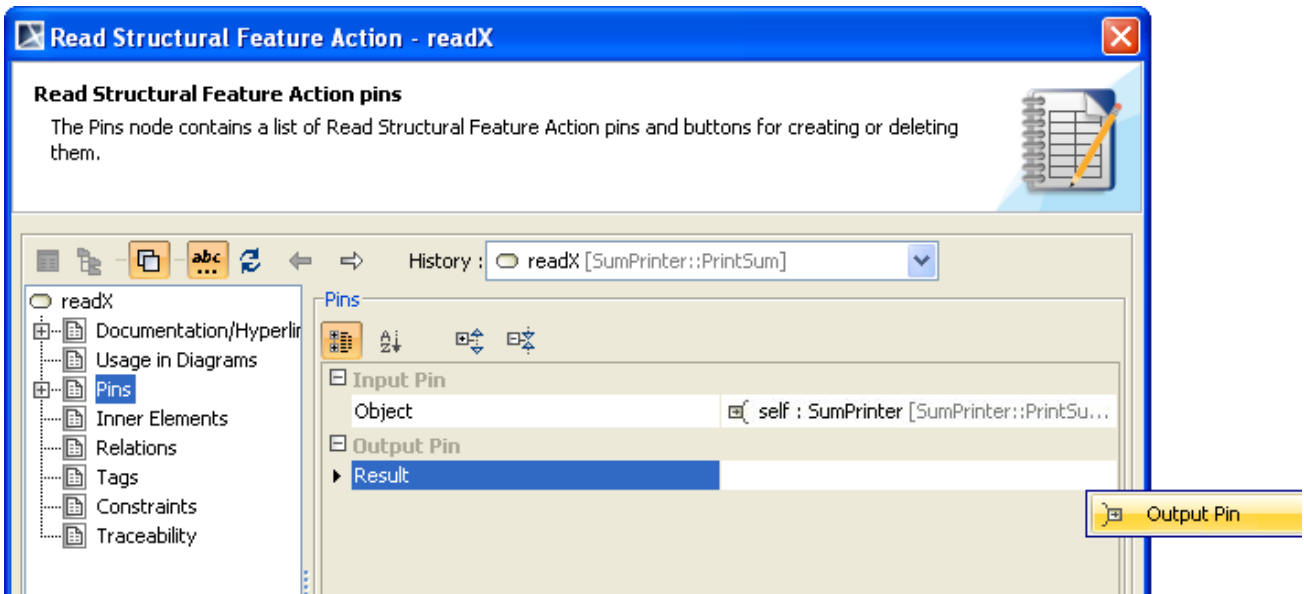


Figure 62 -- Adding Output Pin to readX Action

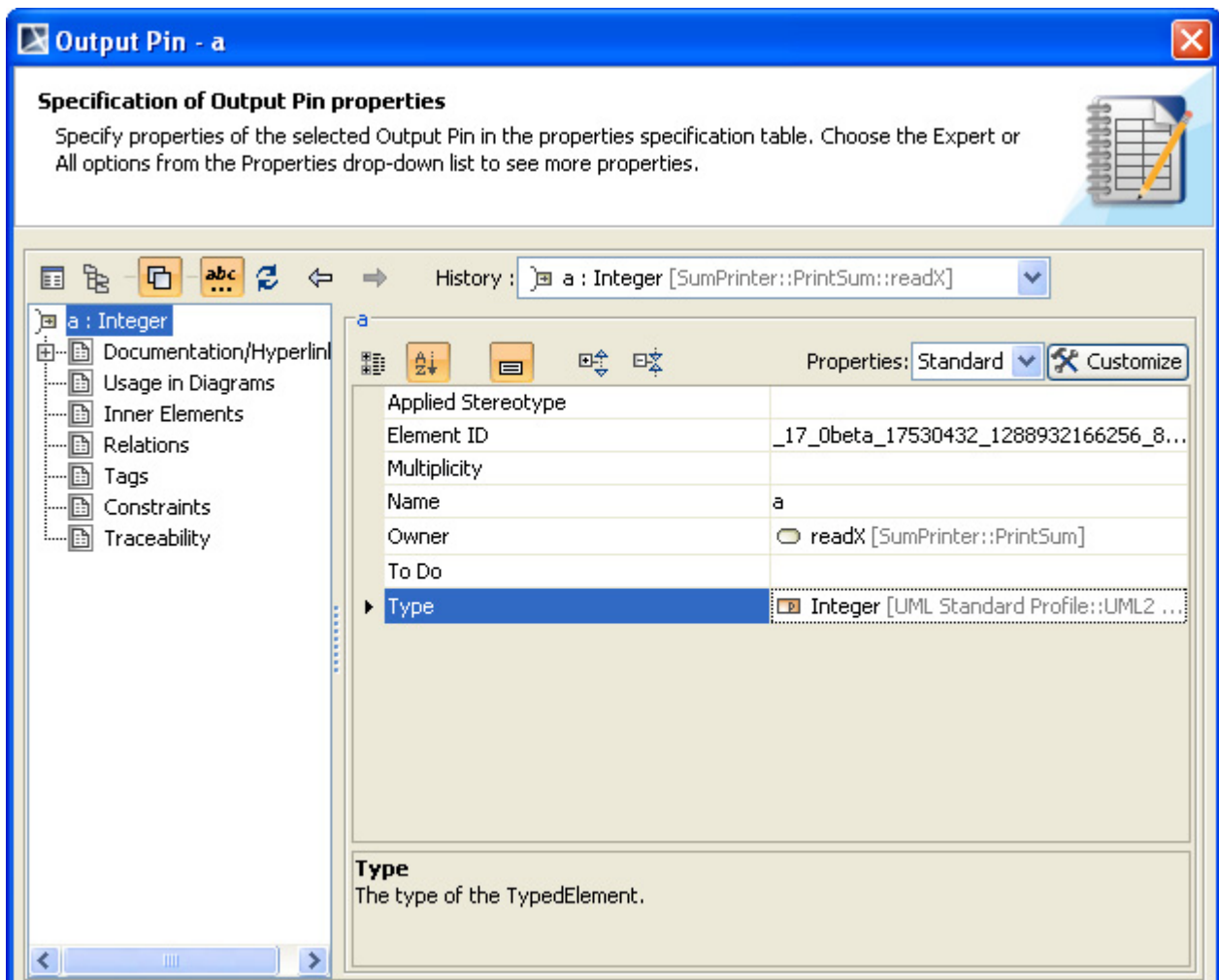


Figure 63 -- Naming the Output Pin and Selecting Its Type

11. Click the **readX** action on the activity diagram and select **Display Pins** on the smart manipulator (Figure 64). The **Select Pin** dialog will open (Figure 65).

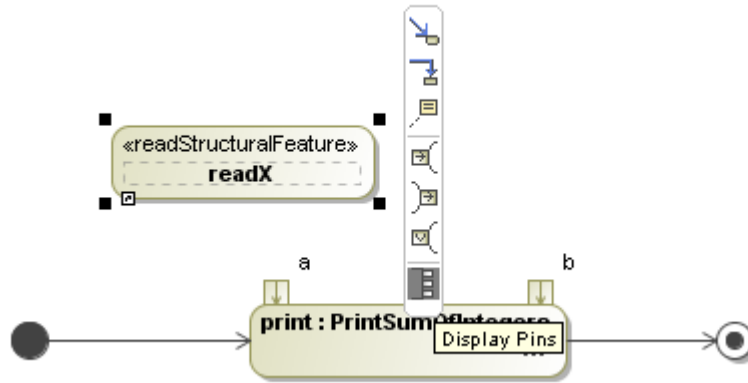


Figure 64 -- Pins of the Selected Action on Smart Manipulator

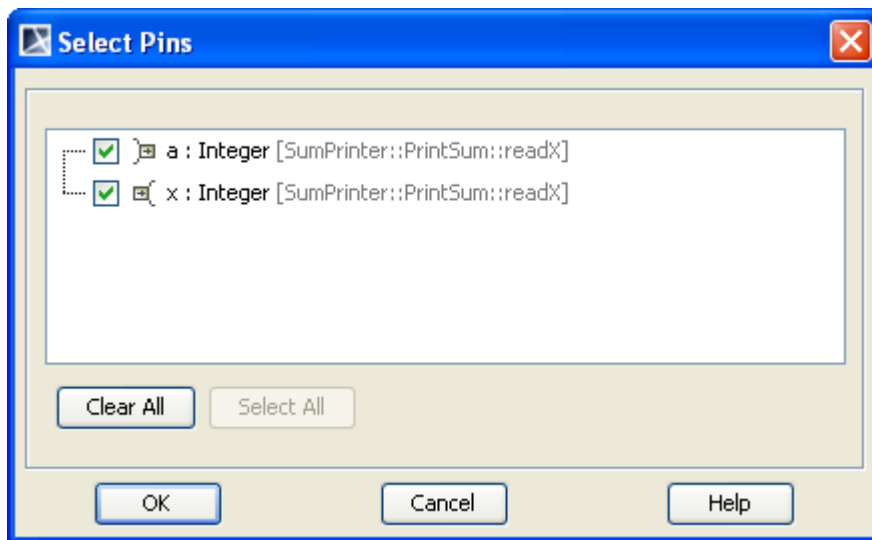


Figure 65 -- Selecting in the Select Pins Dialog

12. Select all pins and click **OK**. The **Select Pins** dialog will close.
13. Connect pin **a** of the **readX** action to pin **a** of the **print** action (Figure 66).

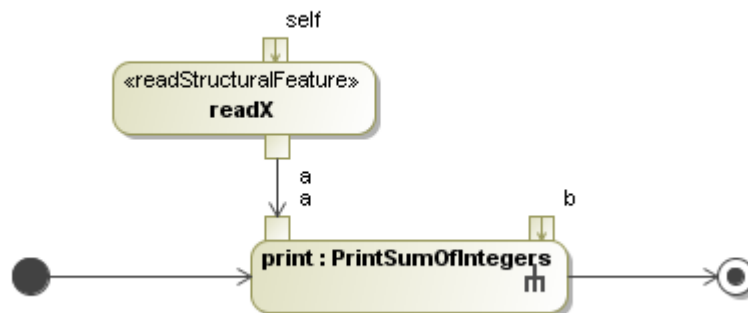


Figure 66 -- Activity Diagram Showing the Flow between **readX** and **print** Actions

14. Repeat steps 4 to 13 to create a **readY** action, which is the **ReadStructuralFeatureAction**, with the following arrangement:

- the name of the action is “readY”.
- the structural feature is ‘y’ attribute of the **SumPrinter** class.
- the name the output pin of **readY** is ‘b’.
- the output pin **b** of **readY** connects to pin **b** of the **print** action.

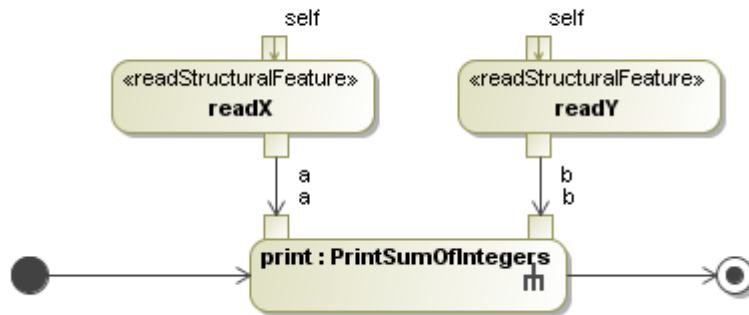


Figure 67 -- Activity Diagram with readX and readY Actions

(vii) To create a ReadSelfAction to read a runtime object that will be supplied to the input pins of **readX** and **readY** actions:

1. Right-click **Any Action** on the diagram toolbar of the Activity diagram. The **Select Action Metaclass** dialog will open (Figure 68).

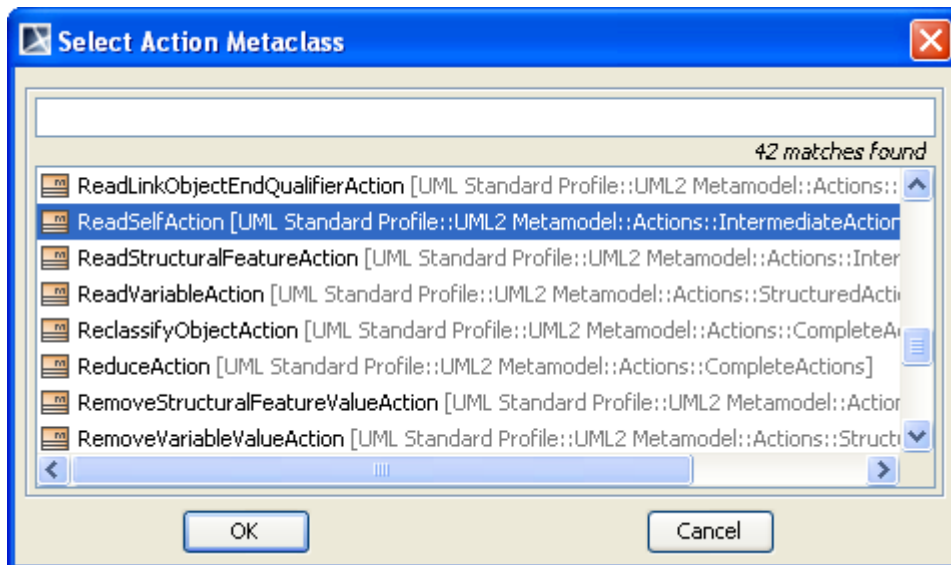


Figure 68 -- Select Action Metaclass Dialog

2. Select **ReadSelfAction** and click **OK**.
3. Click the **PrintSum** activity diagram to create an action and name it “readSelf” (Figure 69).

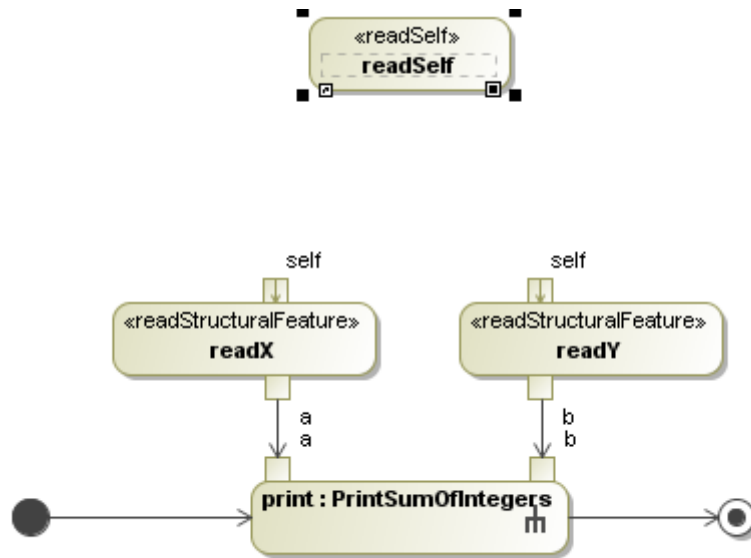


Figure 69 -- Activity Diagram with readSelf Action

4. Right-click the **readSelf** action to open its **Specification** dialog (Figure 70).

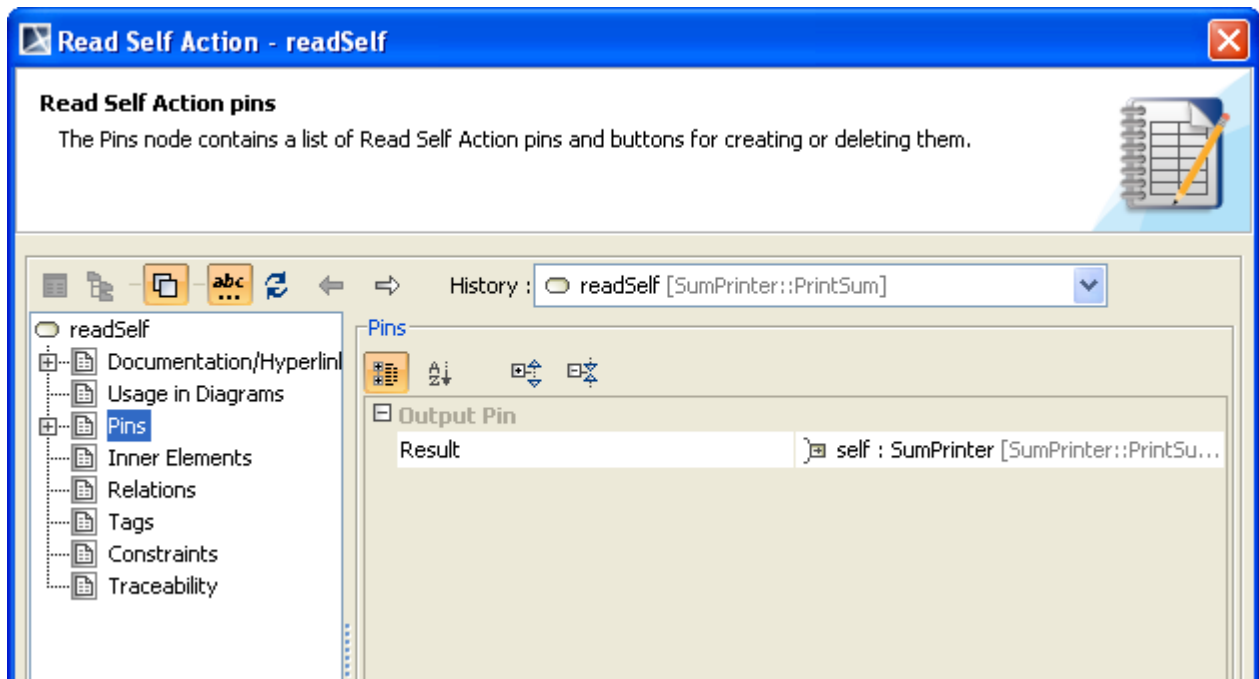


Figure 70 -- Adding Output Pin as a Result Pin of readSelf Action

5. Select **Pins** on the left-hand side pane of the dialog and add a new output pin named "self" of type **SumPrinter** to the **Result** row.
6. Go to the **PrintSum** activity diagram and show the output pin of the **readSelf** action using the smart manipulator button.
7. Create a **Fork Horizontal** and connect it to the pins of the actions on the diagrams using an object flow (Figure 71).

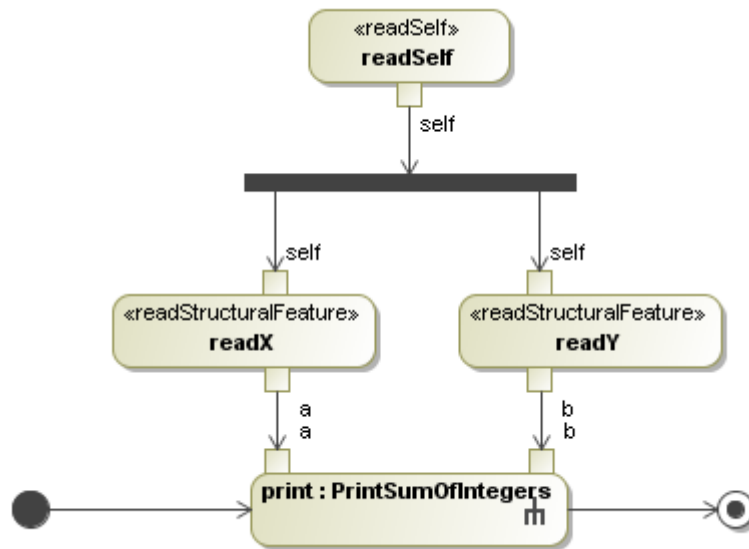


Figure 71 -- A Complete PrintSum Activity Diagram

The final step will be to create an **InstanceSpecification** whose classifier is the **SumPrinter** and assign the values to the slots that correspond to properties **x** and **y**. These values will be used during the simulation.

(viii) To create an **InstanceSpecification** whose classifier is the **SumPrinter** and assign the values to the slots that correspond to the properties **x** and **y**:

1. Right-click the **Data** model and select **New Element > InstanceSpecification**.
2. Name the created InstanceSpecification "instance" (Figure 72).

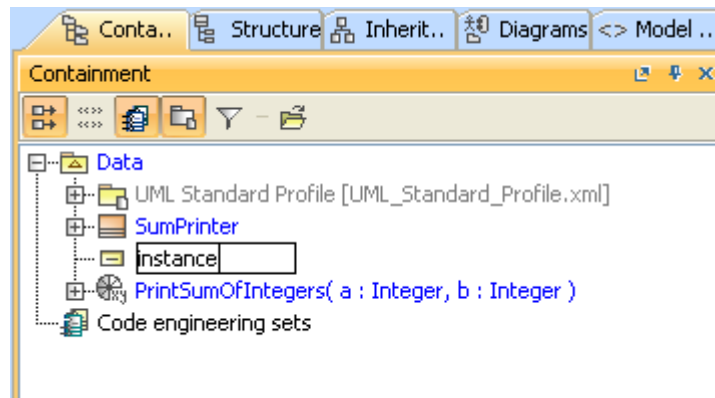


Figure 72 -- The Created InstanceSpecification in the Containment Browser

3. Open the **Specification** dialog of **instance**.
4. Click the **Classifier** field button. The **Select Element** dialog will open (Figure 73).

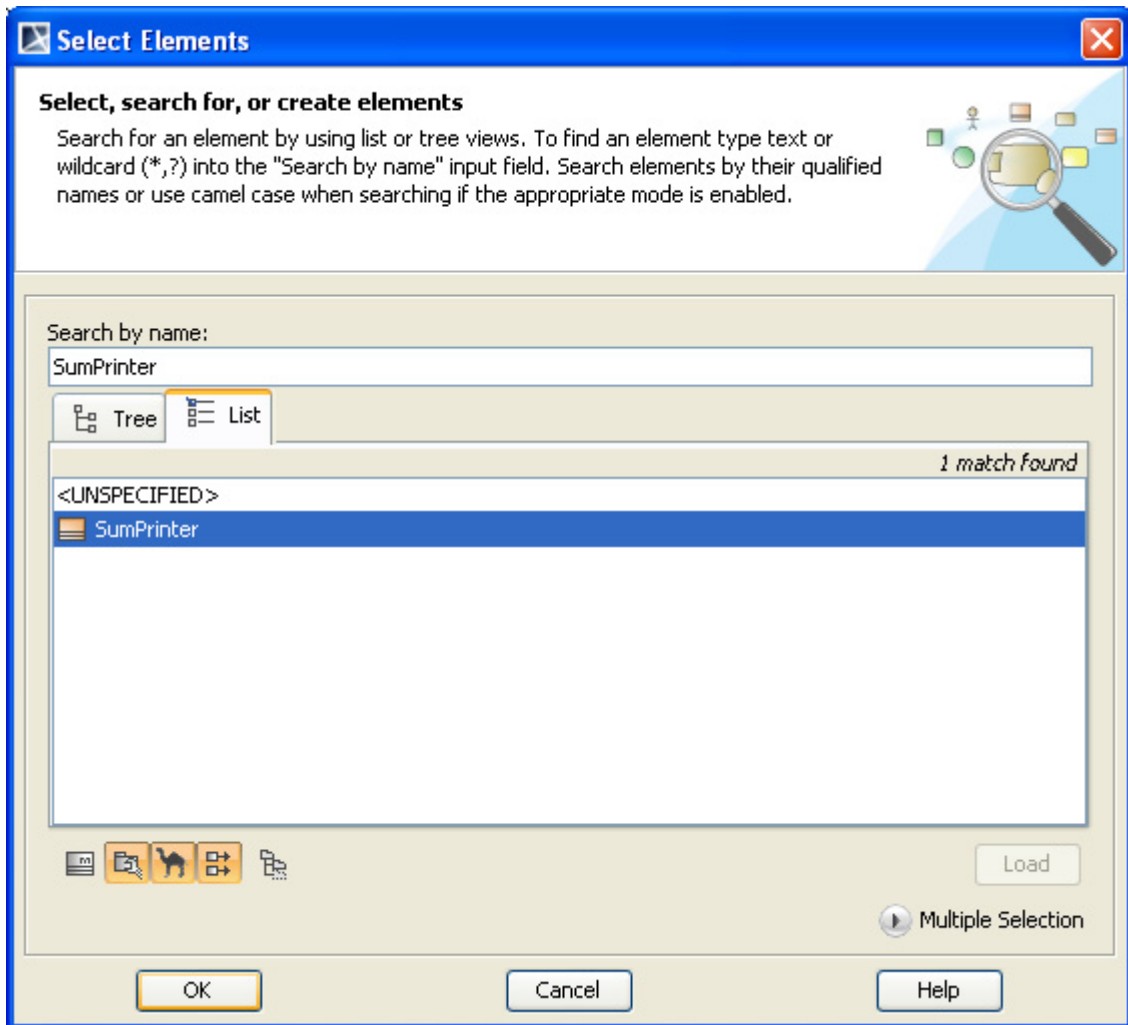


Figure 73 -- The Select Element Dialog of the Selected Classifier

5. Edit the classifier by selecting the **SumPrinter** class and click **OK**.
6. Click **Slots** on the left-hand side pane of the **Specification** dialog and select **x:Integer** (Figure 74).
7. Click the **Create Value** button to create a new value of the slot (Figure 74). The **Value** box will open (Figure 75).

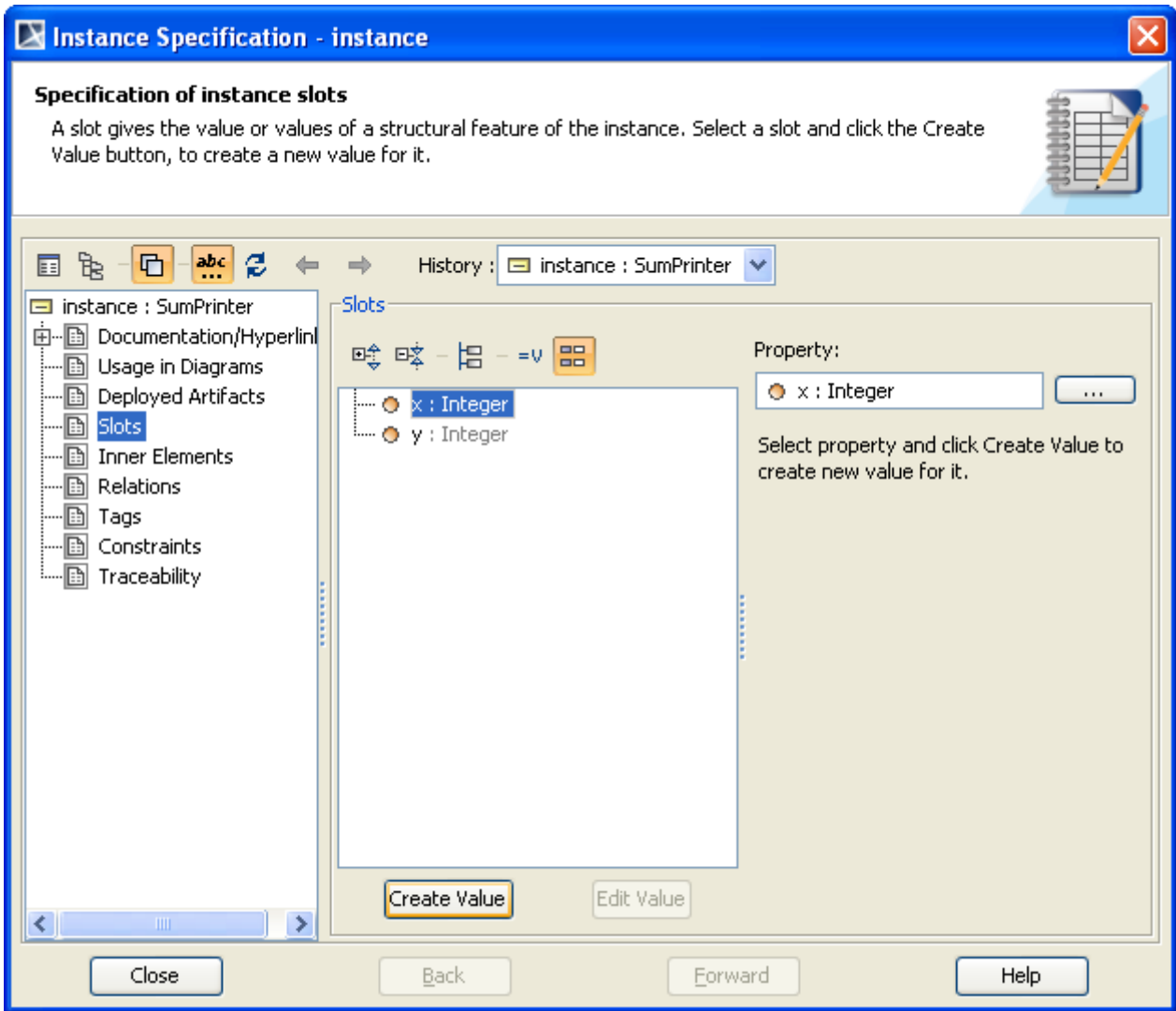


Figure 74 -- Creating Slot Value of x

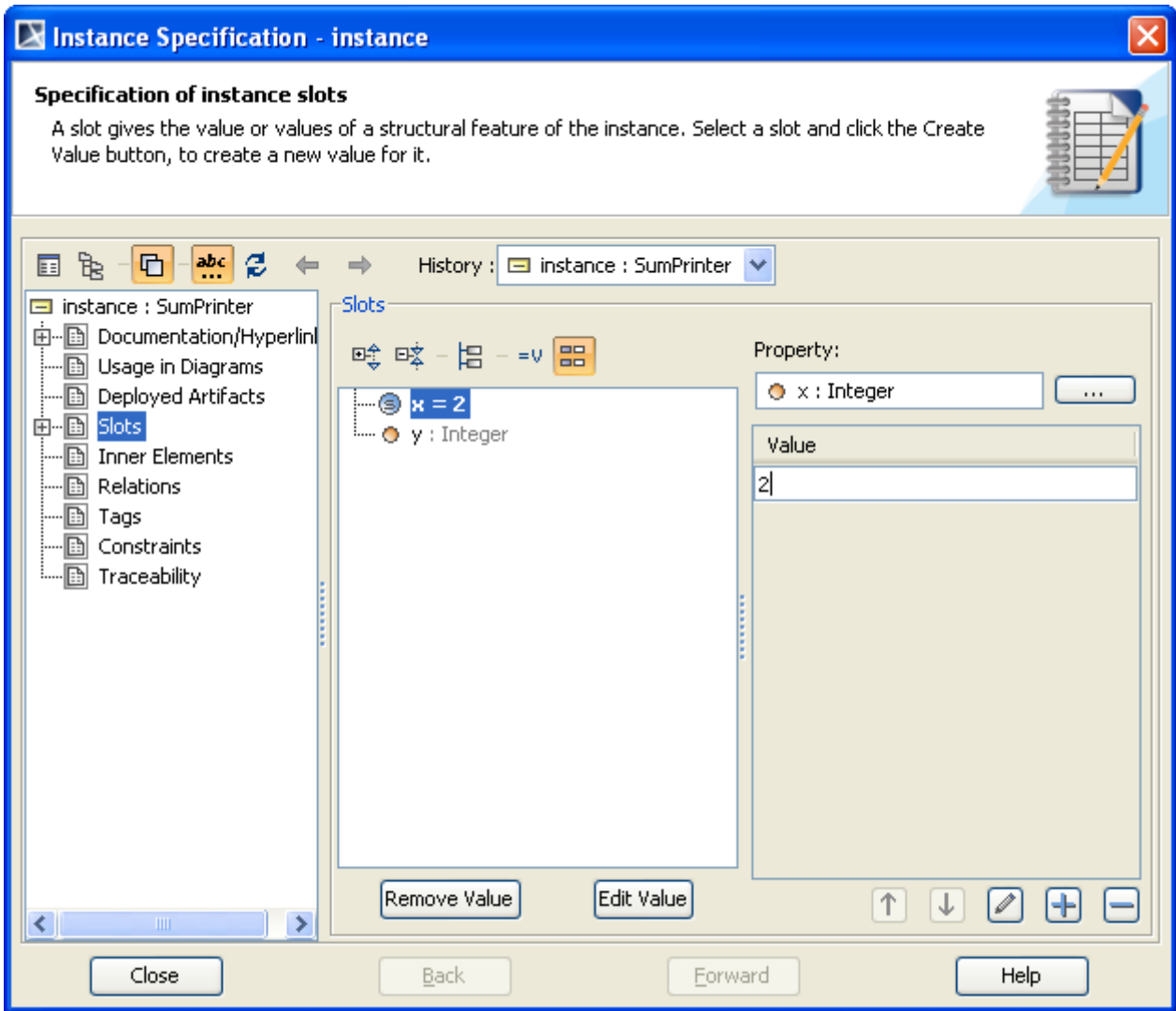


Figure 75 -- Assigning Value to Property x Slot

8. Type, for example, 2 as the value of the property x slot.
9. Repeat the same steps to assign 8 as the value of the property y slot (Figure 76).

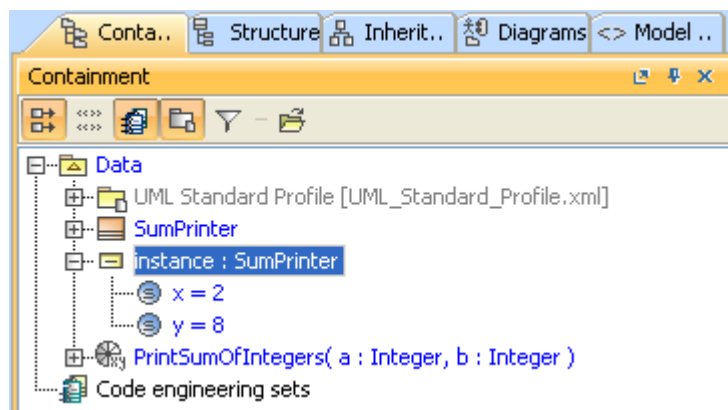


Figure 76 -- The created InstanceSpecification with Slot Values in the Containment Browser

The model is now ready to be executed.

8.3 Executing Activity

You can add some breakpoints to the model created in 8.2 Creating Model for Activity Execution before executing it. This section will demonstrate how to suspend the execution at some specific points by using breakpoints. You can add a breakpoint to an element using either the diagram or browser context menu.

The following example will show you how to add breakpoints to pin **a** and **b** of the **print** action. Once the model execution has reached these pins, the simulation will be suspended.

To add a breakpoint to an element and execute the model:

1. Right-click an element and select **Simulation > Add Breakpoint(s)** (Figure 77). The breakpoints will be shown in the **Breakpoints** pane of the **Simulation Window** (Figure 78).

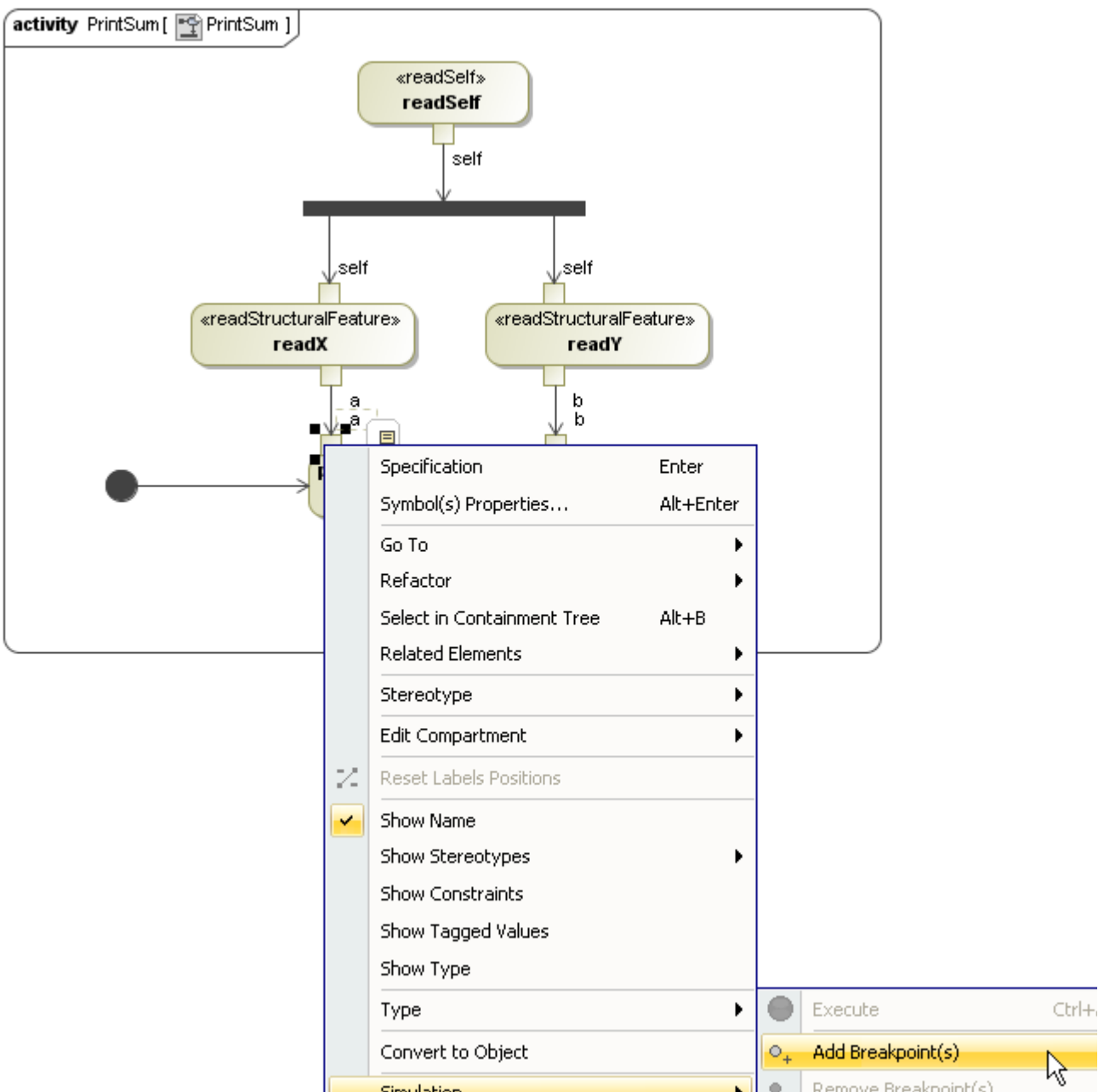


Figure 77 -- Adding Breakpoints to Pin **a** of **print** Action

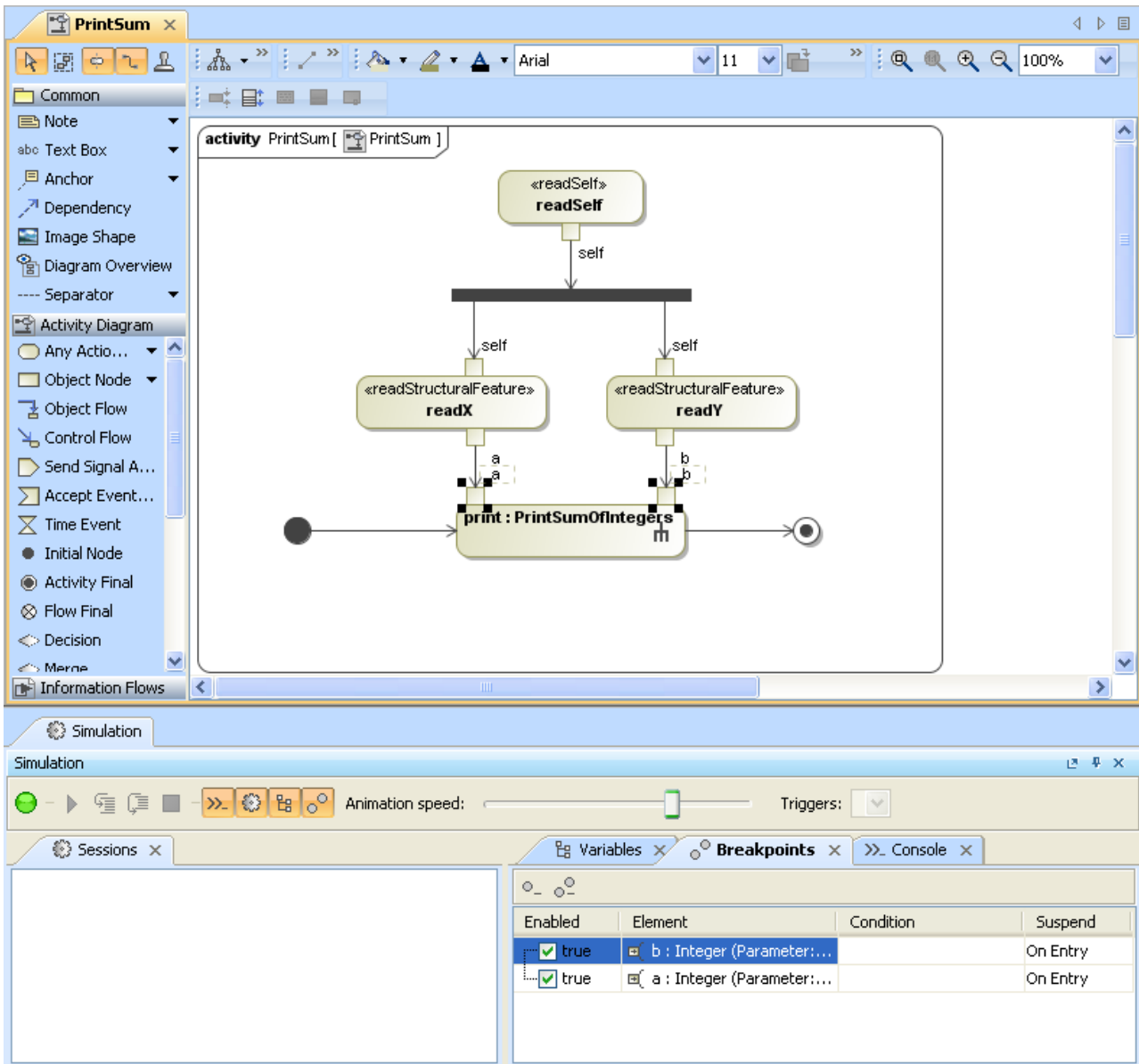


Figure 78 -- Breakpoints Pane in Simulation Window

You can open the **Simulation Window** by clicking **Window > Simulation** on the main menu (Figure 79).

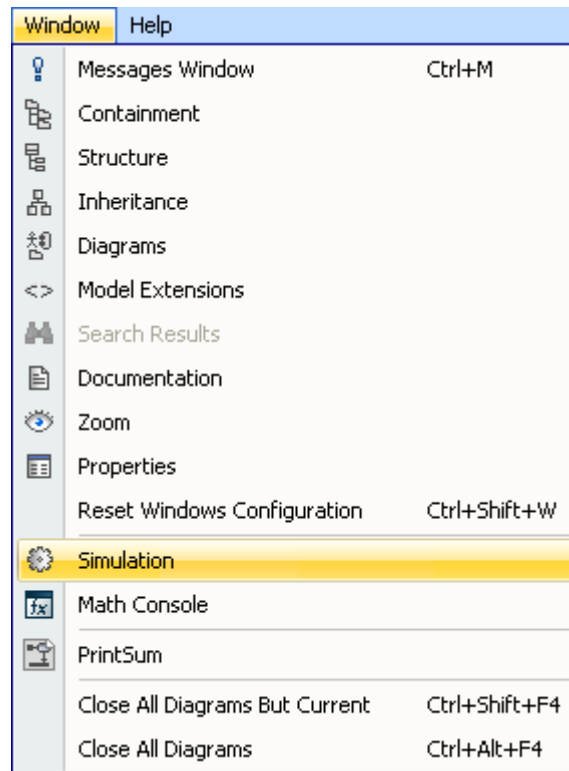


Figure 79 -- Opening Simulation Window from the Main Menu

2. Right-click **instance** in the containment browser and select **Simulation > Execution** (Figure 80) to execute the model from **instance**, which is the InstanceSpecification of the **Sum-Printer** classifier.

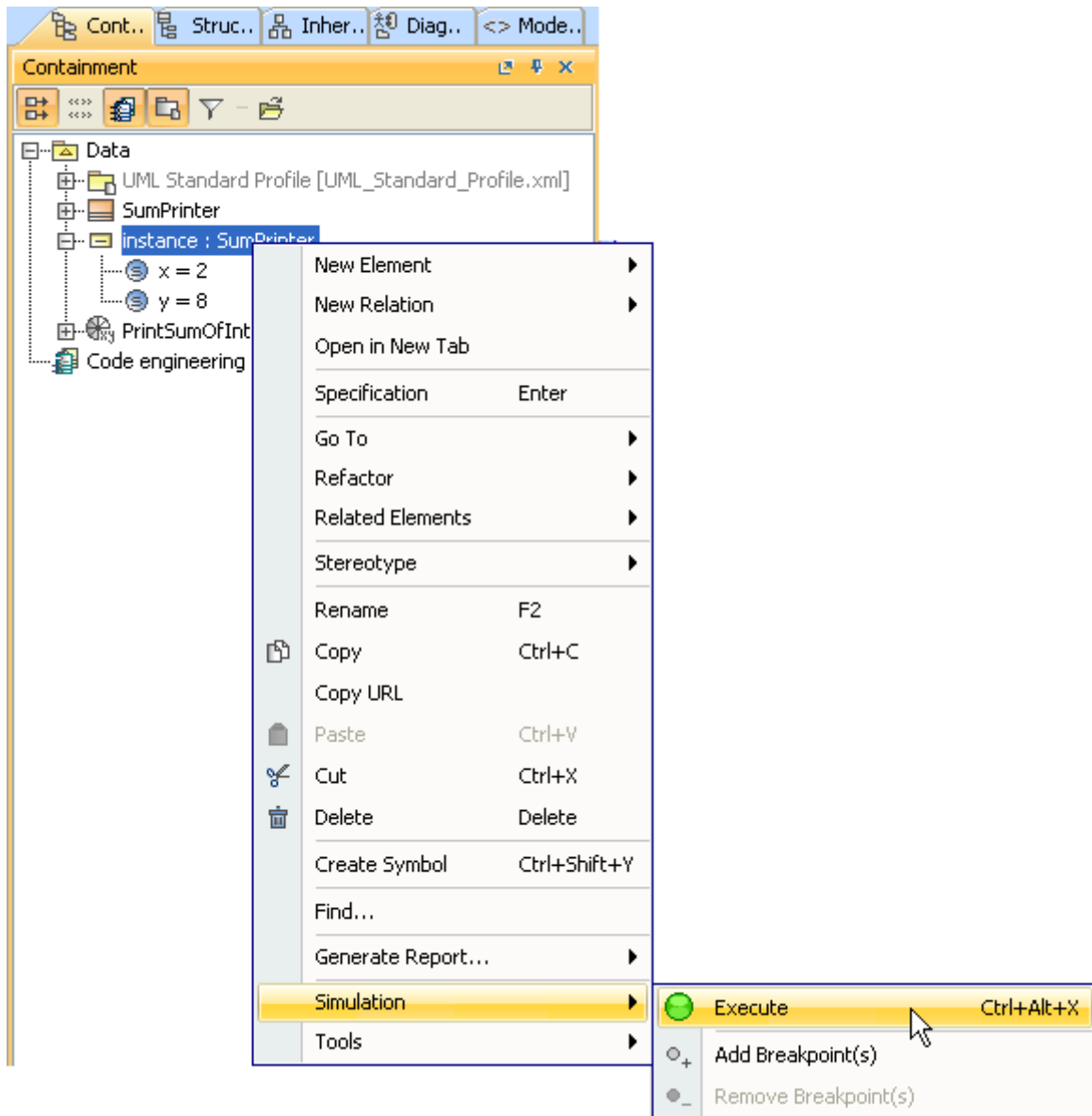


Figure 80 -- Executing InstanceSpecification

3. A new simulation session will be created and displayed in the **Sessions Pane** of the **Simulation Window** (Figure 81). The symbol of elements that have breakpoints attached will be highlighted in yellow (default) (Figure 82).

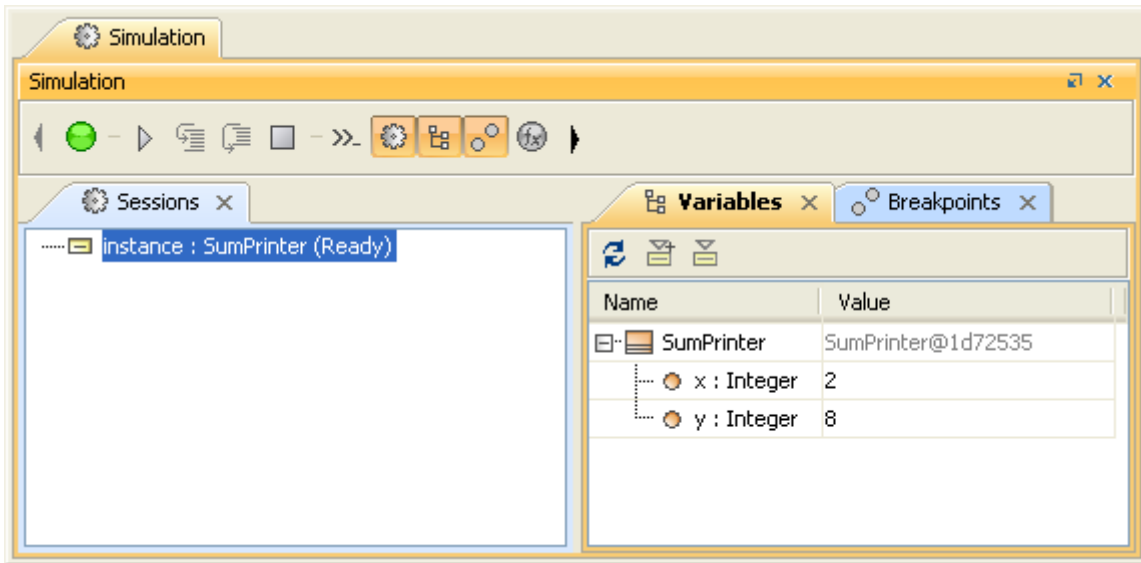


Figure 81 -- Simulation Session in the Sessions Pane of Simulation Window

4. Click the **Run Execution** button on the **Simulation Window** toolbar. Cameo Simulation Toolkit will animate the execution on the **PrintSum** activity diagram. The execution will be suspended when pin **a** or **b** of the print action is activated. You can hover your mouse pointer over the active element to see its runtime value.

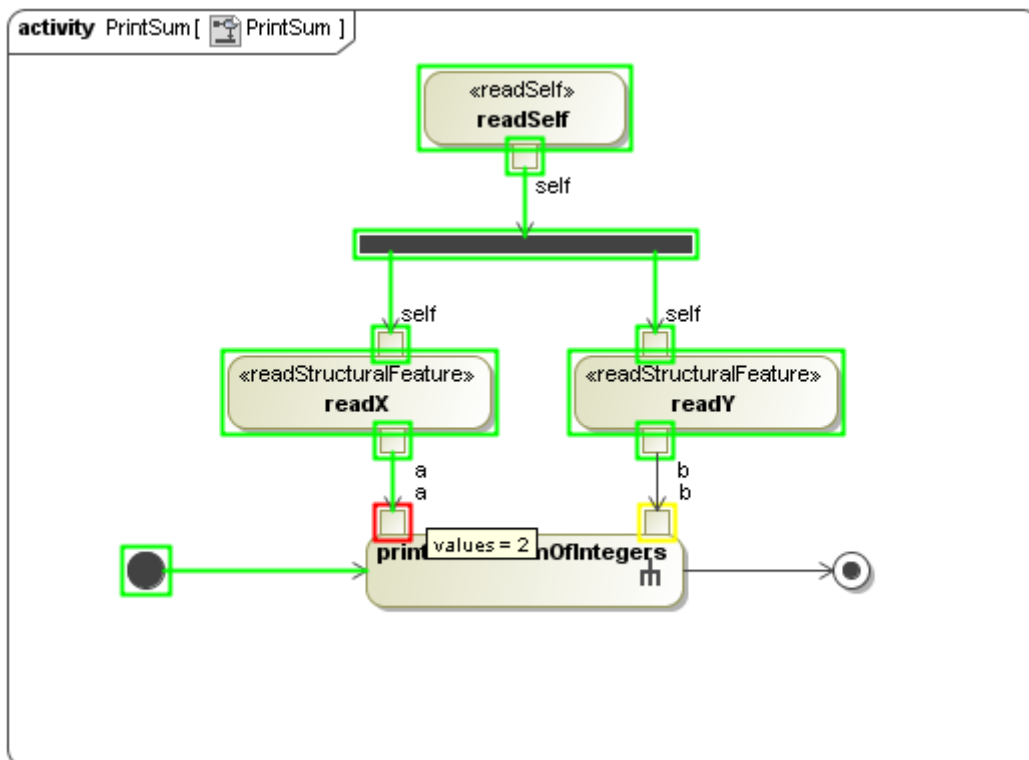


Figure 82 -- The Execution is Suspended when Pin a is Activated

5. Click the **Resume Execution** button on the **Simulation Window** toolbar to continue the execution (Figure 80).

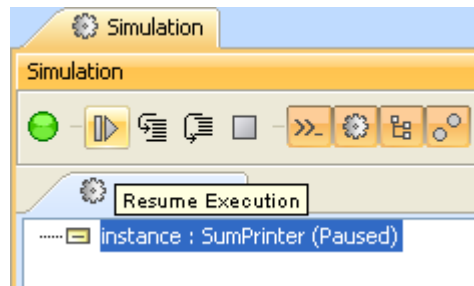


Figure 83 -- Resume Execution Button in Simulation Window

- The execution will be suspended again when pin b is activated. Click **Resume Execution** to continue the execution. In the **Simulation Console** of the **Simulation Window**, you can see the printed value of 10, which is the summation between 2 and 8.

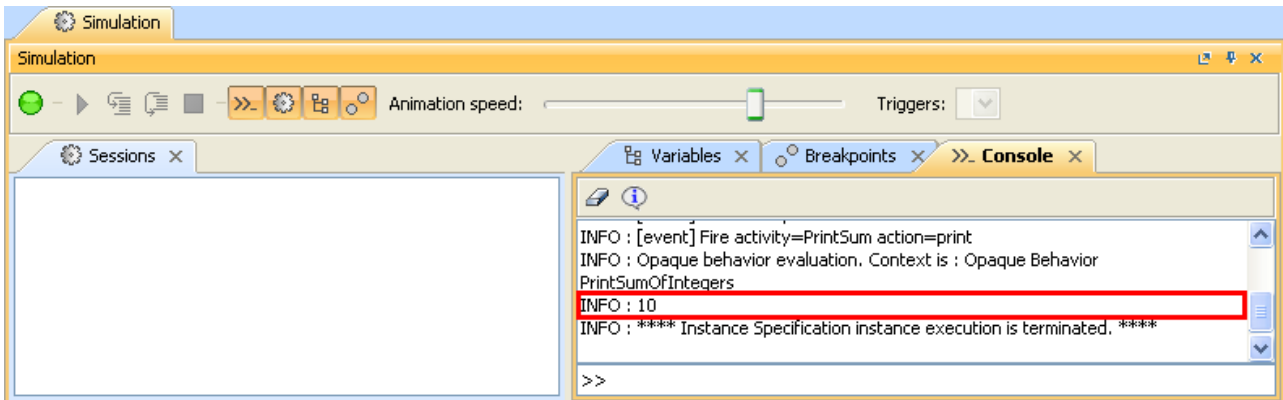


Figure 84 -- Simulation Console of Simulation Window Showing the Printed Result of Summation

NOTE If you do not want to display the animation (silent execution), you can create **Execution Configuration** to customize the execution by selecting **instance** as the **executionTarget** and set **silent** to **true**. See Section 2.2 for more information.

9. Parametrics Simulation

9.1 About Parametrics Engine

Cameo Simulation Toolkit comes with the Parametrics Engine plugin to enable you to calculate the mathematical model of a system. The system on which the parametric simulation can be performed, must be modeled by SysML. Therefore, it must be defined as a SysML block that contains constraint properties as its owned attributes or nested properties.

The Parametric Engine will use the Mathematical Engine to solve the mathematical and logical expressions that are defined as the constraints of Constraint Blocks. The default Mathematical Engine, which comes with Cameo Simulation Toolkit, is **Math Console**.

NOTE

- The SysML profile is required for a parametric execution.
- A Parametrics simulation evaluates the expressions in one direction by specifying inputs to get outputs, for example, for the expression $z = x + y$, the values of x and y must be given to evaluate z .
- Binding Connectors (the connectors applied with the «BindingConnector» stereotype) must be used to connect Value Properties to Constraint Parameters, Value Properties to Value Properties, or Constraint Parameters to Parameters.

9.2 Adapting Model for Parametric Execution

9.2.1 Understanding the Flow of Parametric Execution

The Parametric engine in the current version of Cameo Simulation Toolkit can solve expressions in a one-way direction only. The variables that are defined on the left-hand side of an expression will be considered as the output parameters, whereas the variables on the right-hand side as the input parameters. The values of the input constraint parameters must be specified in order to evaluate the values of the output constraint parameters.

When you start the parametric execution on a SysML block, the Parametric Engine will execute the constraint and nested constraint properties of the block. The order of the execution of the constraint properties will depend on the expressions. If an input constraint parameter of the constraint property is connected to an output constraint parameter of another constraint property, the constraint property that requires the input values will be executed after the one that provides the values to the output constraint parameters.

In the SysML Parametric Diagram of the *test_parametrics.mdzip* sample (Figure 85), which is located in the `<md.install.dir>/samples/simulation/Parametrics/` directory, you can see that the order of the execution of constraint properties will be: s1, s2 and s3 respectively.

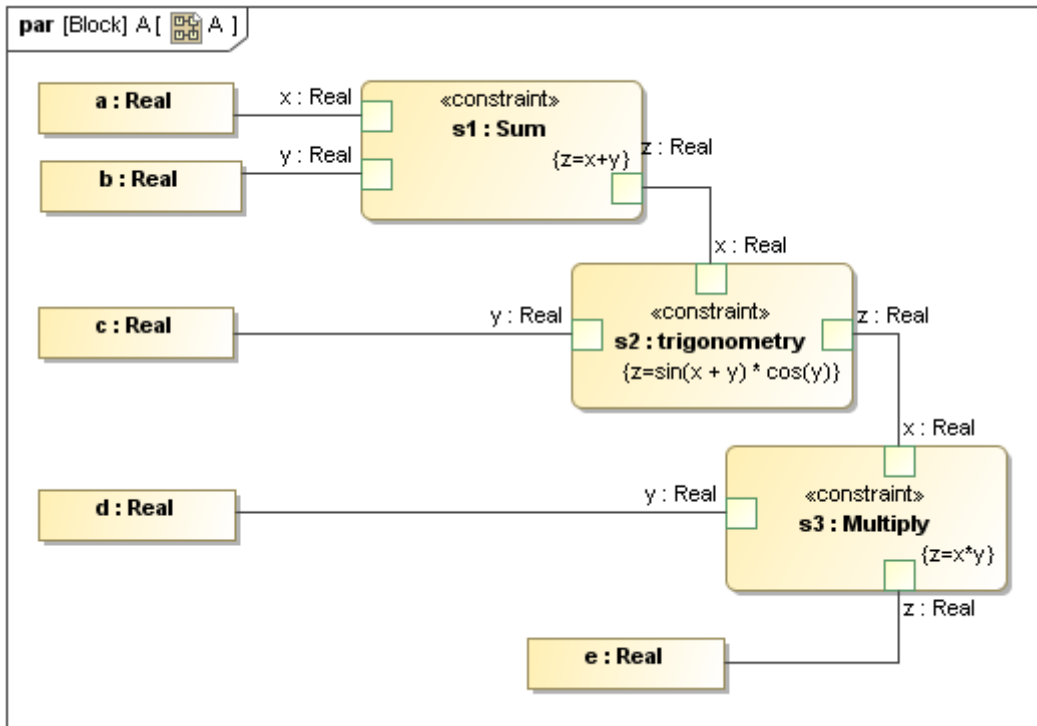


Figure 85 -- SysML Parametric Diagram in the test_parametrics.mdzip Sample

9.2.2 Typing Value Properties by Boolean, Integer, Real, Complex, or Their Subtypes

SysML provides the QUDV library to create different value types. You can use these value types to type the value properties that are defined in a SysML model, which can be used for the parametric execution. However, they must be inherited from the basic SysML value types which are Boolean, Integer, Real, and Complex. You can see the example in the CylinderPipe.mdzip sample.

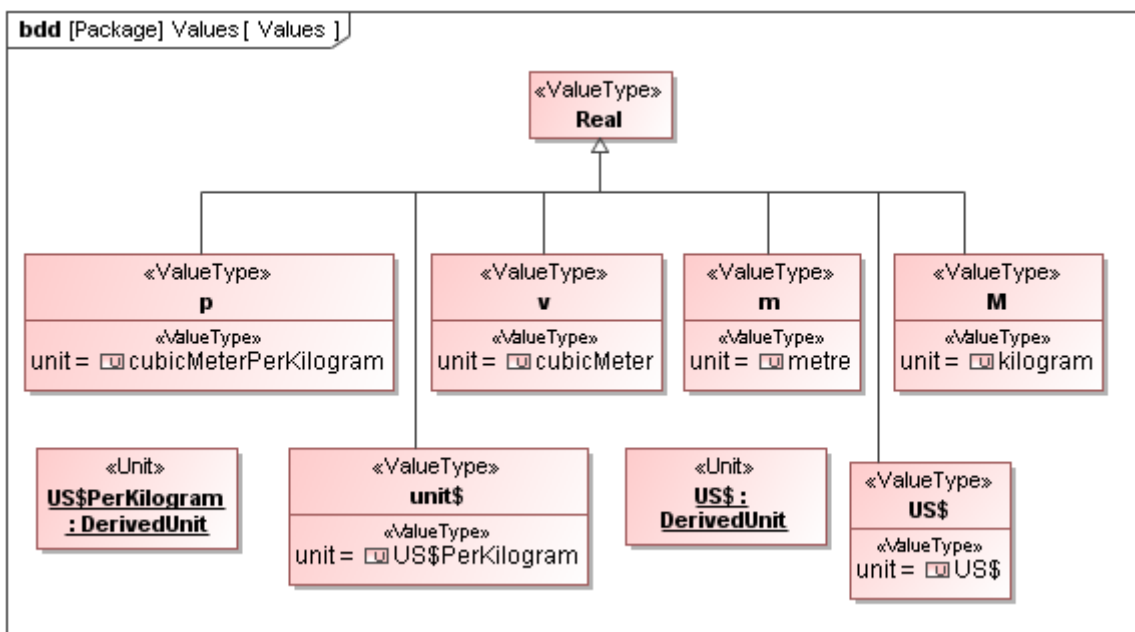


Figure 86 -- Value Types Inherited from Real

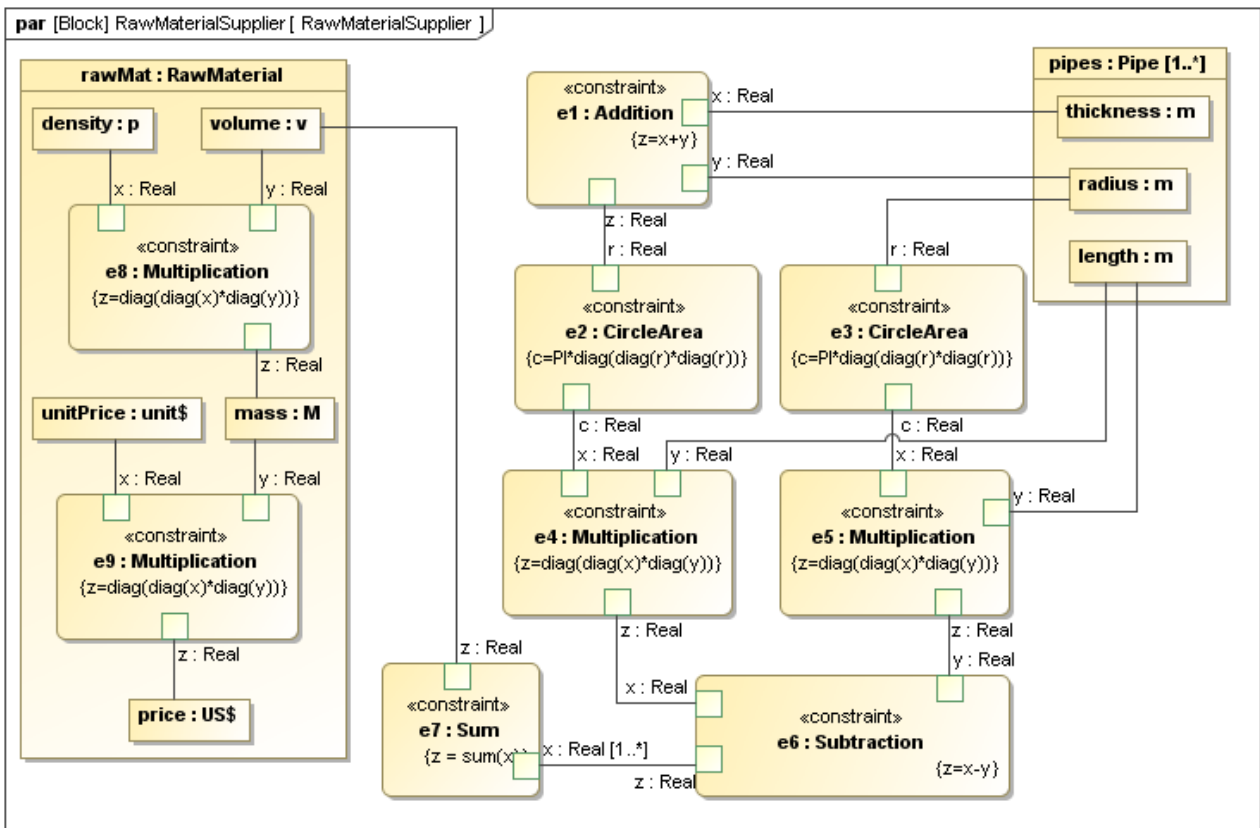


Figure 87 -- SysML Model Using the Value Types Defined in Figure 86

9.2.3 Using Binding Connectors

SysML provides a binding connector to connect elements whose values are bound together. The Parametric engine uses the binding connector to distinguish between the connector that represents a physical connection and the one that bounds the values. Therefore, you can use a binding connector to connect a value property to a constraint parameter, and a constraint parameter to another constraint parameter. You cannot use it to connect a value property to another value property if neither of them is connected to a constraint parameter, because it cannot specify the flow of the parametric execution direction. To do this, you need to first create a constraint block to assign the operation, and then insert a binding connector.

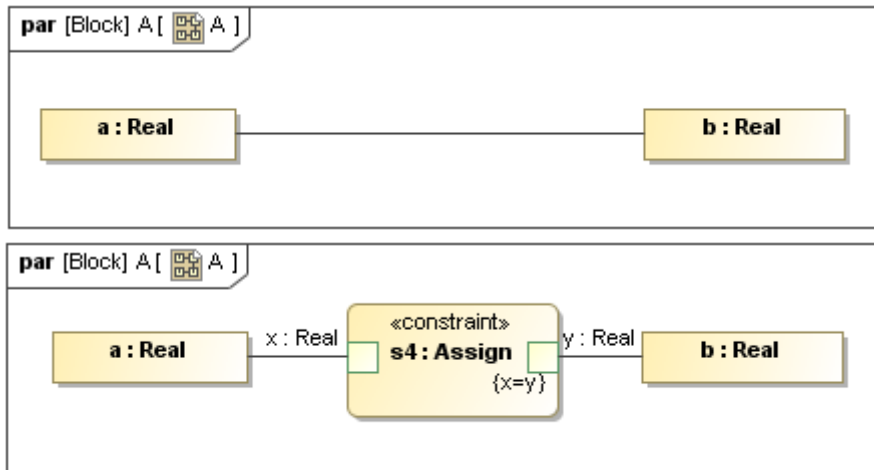


Figure 88 -- Using a Constraint Property Instead of Directly Binding Value Properties using Binding Connector

9.2.4 Creating InstanceSpecification with Initial Values

An InstanceSpecification of the SysML block is required to start the parametric execution on a SysML model. The initial values, which will be used for simulation, must be specified as the slot values of the InstanceSpecification. You also need to specify the values of the value properties that are connected to the input constraint parameters, otherwise, the default values will be used. The default value of the value properties whose type is Number or its subtypes, is zero. The default value of the value properties, which are boolean, is false.

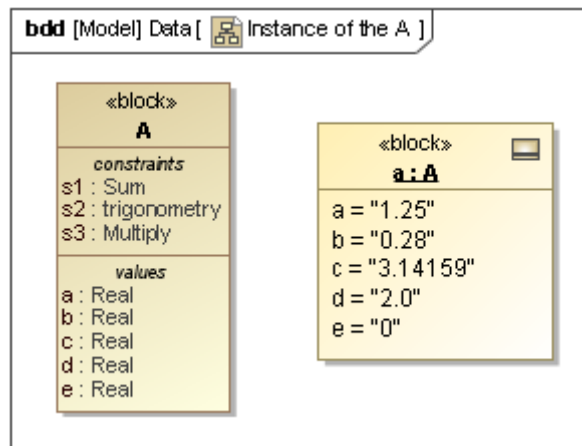


Figure 89 -- InstanceSpecification with Initial Values in the test_parametrics.mdzip Sample

You can also use the InstanceSpecification of a SysML block to store the values resulting from the parametric execution if the execution configuration is used, by defining the resultInstance of the «ExecutionConfig» stereotype in the InstanceSpecification (Figure 90).

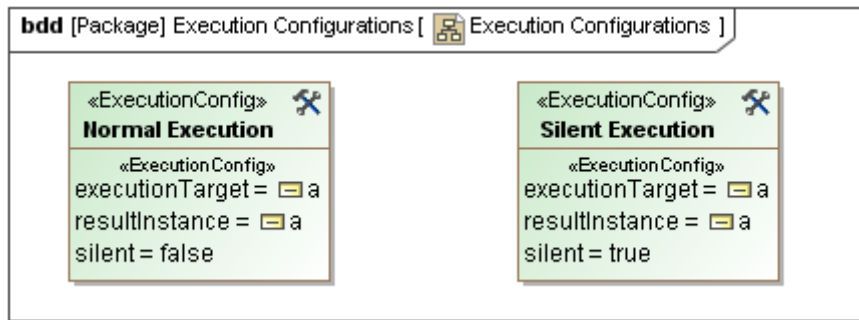


Figure 90 -- Using InstanceSpecification to Save the Result Values from Simulation

NOTE You need to create InstanceValues for the slots which correspond to the part properties, reference properties, or shared properties (InstanceValue), even though they contain empty slot values of the value properties. Otherwise, you cannot save the result values to the InstanceSpecification. You need to create the slots before saving the result values.

9.2.5 Working with Multiple Values

According to the multiplicity of property elements, a runtime object cannot contain multiple runtime values that correspond to a property. If the property is bound to constraint parameters, which are the input of an expression, then a list of values will be passed on to the mathematical engine to solve the expression. The mathematical and logical expressions, which are defined in the constraint blocks, must support the use of multiple values.

9.2.5.1 Modifying Expressions to Support Multiple Values

Since a matrix column will be constructed from a list of input values in the Built-in Math Solver, the mathematical expression must be written in a form that supports matrix variables.

If you refer to the Multiply constraint block in the test_parametrics sample, you will see that the mathematical expression of the constraint block is: $z = x * y$. If four values are passed on to the mathematical engine for each x or y parameter, then two column matrices (4x1 matrices) will be constructed and used to solve the expression. However, the column matrices cannot solve the expression because the matrix dimensions do not agree (the number of column of x must be equal to the number of row of y). To solve this, you need to rewrite the expression. You need to change the column matrices to diagonal matrices before the multiplication operation starts by changing the expression to: $z = \text{diag}(\text{diag}(x) * \text{diag}(y))$.

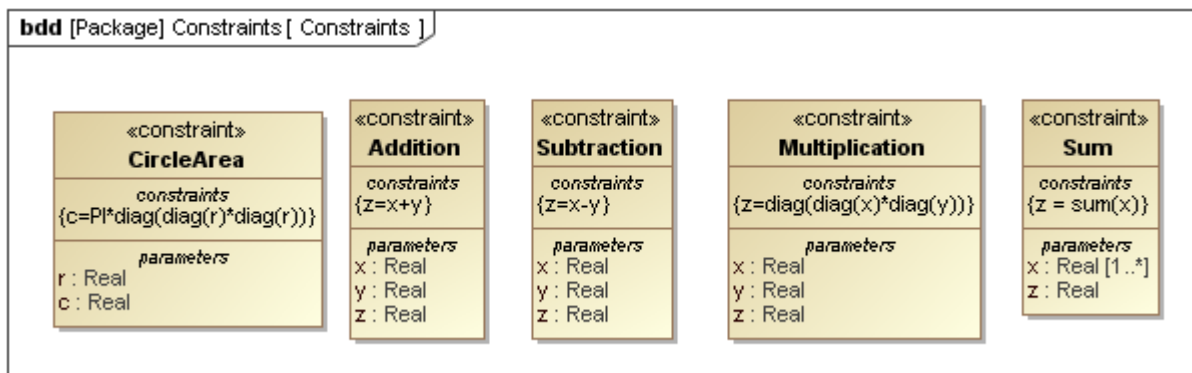


Figure 91 -- Constraint Blocks Expressions Used with Multiple Values in CylinderPipe.mdzip Sample

9.2.5.2 Constructing Values List from Complex Aggregation Structure

If you have an InstanceSpecification of the SysML block that contains multiple slot values and if the slot values are the InstanceValues whose InstanceSpecifications also contain multiple slot values and so on, up to the slot values, which correspond to the value properties that are connected to the constraint parameters. You need to pass all of these values on to the mathematical engine.

If this is the case, the Parametric engine will first collect all of the values that correspond to the value properties that are connected to the constraint parameters, and then create a list of values and pass it on to the Mathematical engine. The order of the values will depend on the order of the slot values. To ensure that the values order will remain the same, you need to specify the **IsOrder** attribute of the Property elements, which has a non-singular multiplicity, to **true**. In the SysML Parametric diagram of the CylinderPipe.mdzip sample (Figure 87), and the InstanceSpecification of the SysML block “RawMaterialSupplier” in Figure 92, the list of the values for the mathematical engine to solve the expression are as follows:

- length = {1.0, 2.25, 12}
- radius = {0.1, 0.25, 0.25}
- tickness = {0.002, 0.002, 0.005}

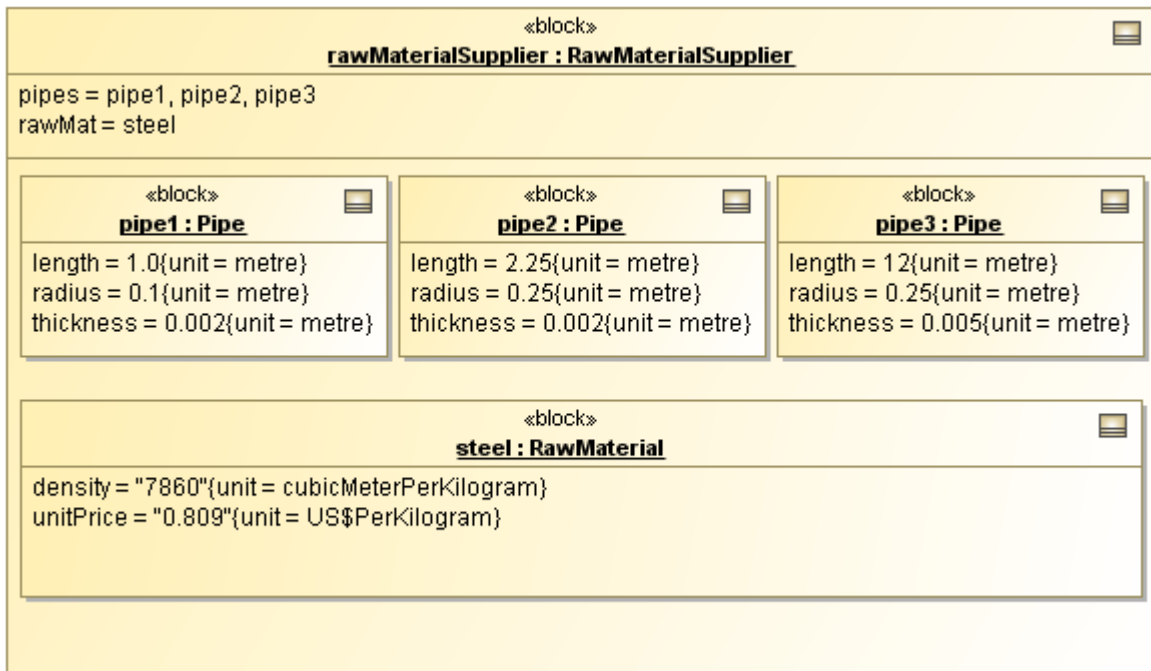



Figure 92 -- InstanceSpecification of Complex Aggregation Structure

9.3 Running Parametric Simulation

This section will use the *test_parametrics.mdzip* sample, located in the `<md.install.dir>/samples/simulation/Parametrics/` directory, to demonstrate how to run a Parametrics simulation.

To run a Parametrics simulation:

1. Start the Parametric Simulation Engine (you can select Block A, InstanceSpecification a:A, or the Execution Configuration class symbol (on the Execution Configurations Block Definition Diagram). Either:
 - (i) right-click the element symbol and select **Simulation > Execute** in context menu
 - or

(ii) select the element symbol and click the **Execute** button  in the **Simulation Window** toolbar. (If you click the **Execute** button without selecting any element and the active diagram is a SysML Parametric diagram, then the classifier, which is the context of the active SysML Parametric diagram, will be used as the element to be executed.)

NOTE In the case that the element to be executed is a Classifier, the InstanceSpecification must be specified. The slot values defined in that particular InstanceSpecification will be used as inputs for the simulation, and placeholders as outputs. If there is only one matching InstanceSpecification found in the project, then it will automatically be used for the simulation. Otherwise, the **Select Element** dialog will open for you to select an InstanceSpecification (Figure 93).

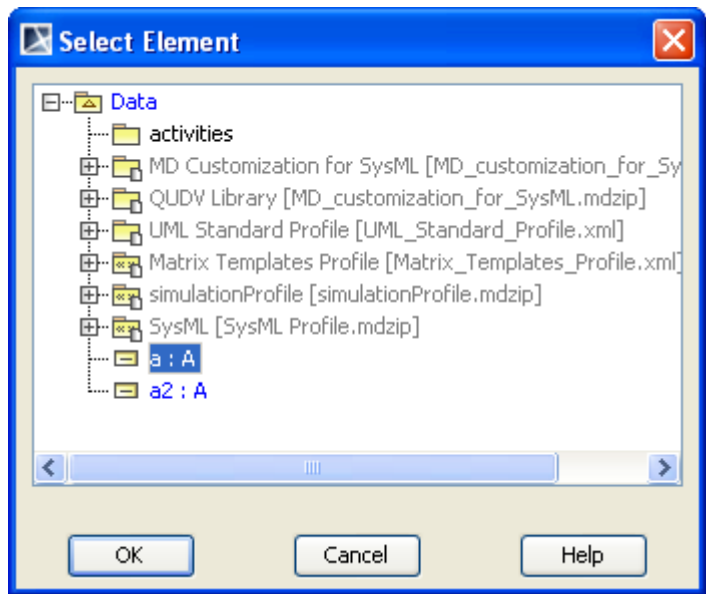


Figure 93 -- Selecting InstanceSpecification in the Select Element Dialog

2. Once the Parametric Simulation Engine has been started, the runtime structure of the executed classifier will be shown in the **Variables Pane** (Figure 94). You can modify the values in the value column of the **Variables Pane**.

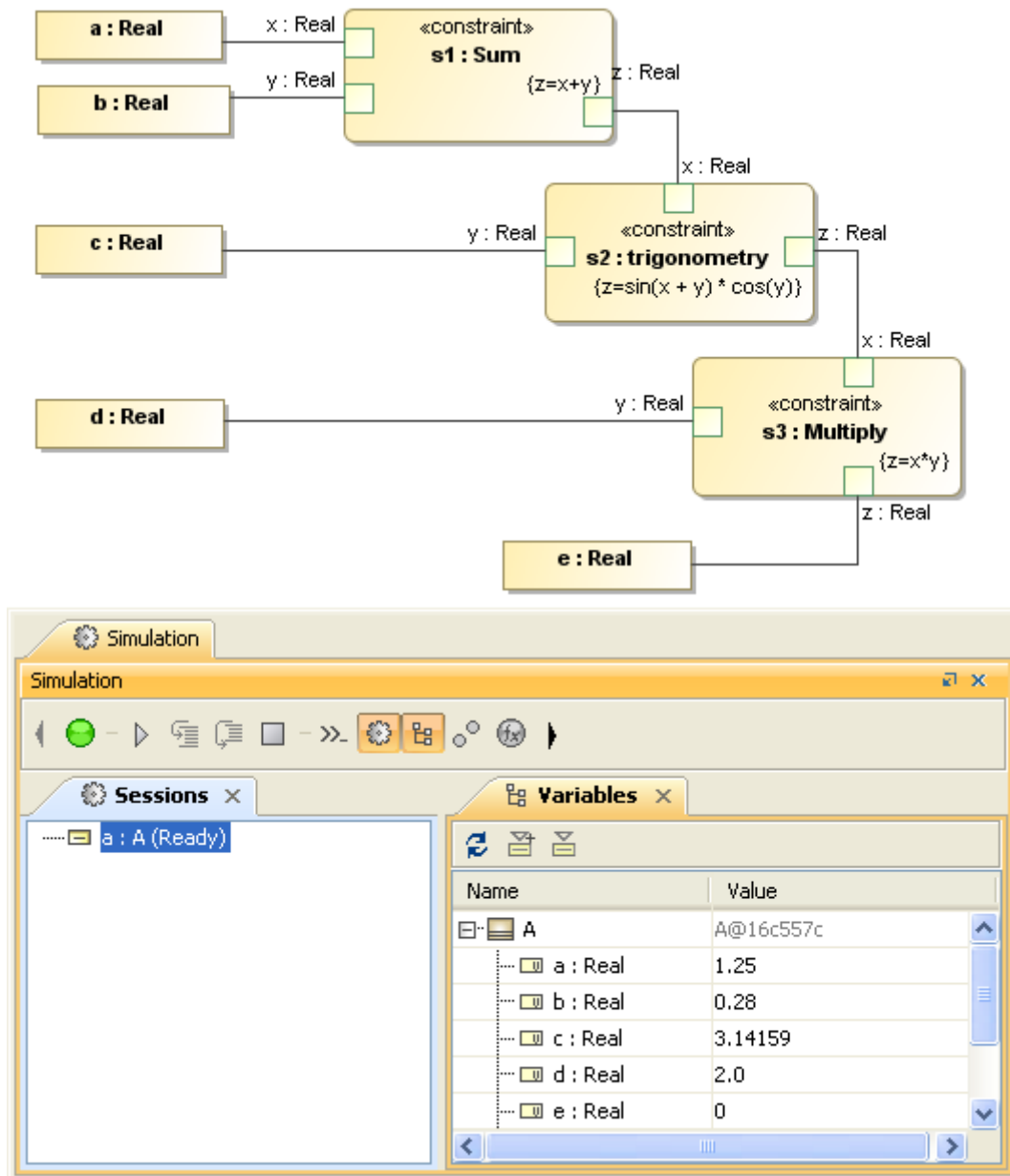



Figure 94 -- Initializing Parametric Simulation Engine with a Selected Classifier

- Click the **Run** button  to start the simulation. The Parametric Simulation Engine will simulate your Parametric model (with animation on your diagram), and input the calculation result into the corresponding slot of the selected InstanceSpecification automatically (Figure 95).

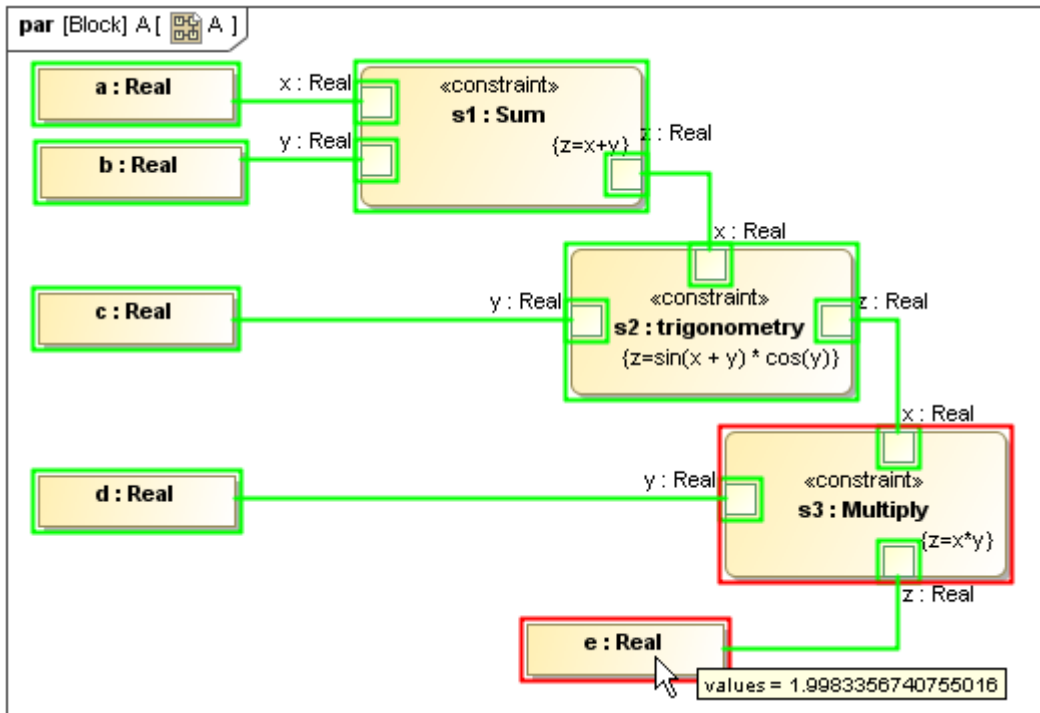


Figure 95 -- Simulation with Parametric Simulation Engine

9.4 Retrieving Simulated Values

You can save the simulated values in an InstanceSpecification, which is specified in the resultInstance of the «ExecutionConfig» stereotype. Therefore, you can save the parametric simulation results to the InstanceSpecification only when the ExecutionConfig is selected for the execution.

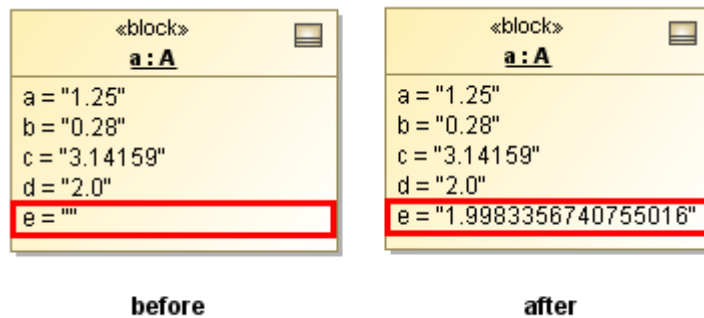


Figure 96 -- Slot Value before and after Simulation with Parametric Simulation Engine

9.5 Executing Parametric Simulation from Activity

The Parametric engine provides an API for a parametric execution with the runtime object of a classifier. The runtime object of the classifier will be passed to the API as an argument and the engine will execute the given object. With this API, you can use a scripting language for the parametric execution, for example:

```
com.nomagic.magicdraw.simulation.parametrics.ParametersEngine.executeObject(Object object);
```

An argument object is the runtime object of a classifier to be executed. You can obtain this particular runtime object by using some UML actions such as ReadSelfAction, ReadStructuralFeatureValueAction, and ValueSpecificationAction, or by using the Cameo Simulation Toolkit Open API. Figure 97 shows the Parametric activity diagram in the CylinderPipe.mdzip sample. The action:ExecuteParametric is used to run the parametric execution. The runtime object, which will be executed, is obtained from the value specification action rawMaterialSupplier.

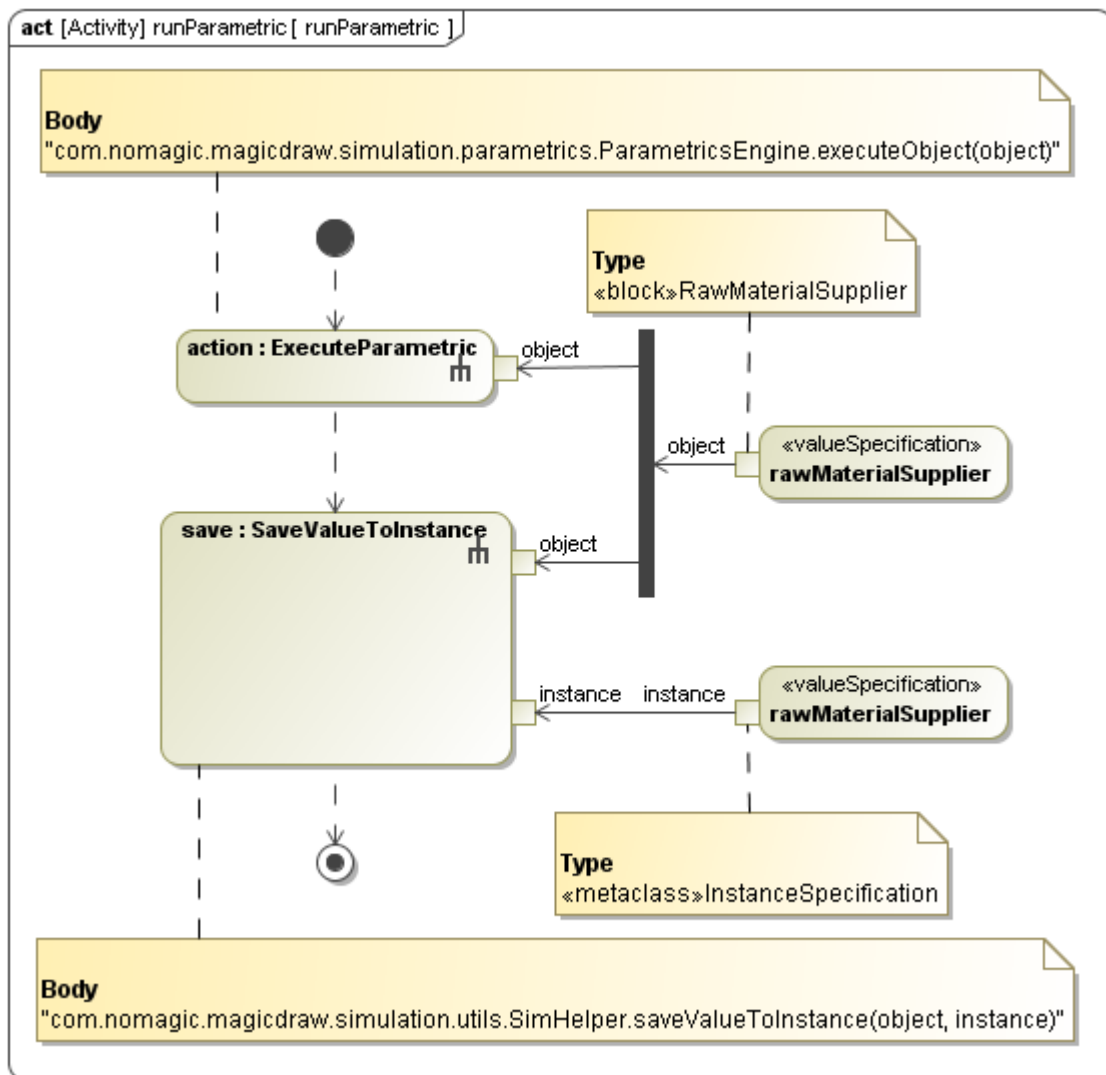


Figure 97 -- Using Action to Execute Parametric

9.6 Sample Projects

The Parametric Simulation sample projects are available in the `<md.install.dir>/samples/simulation/Parametrics` directory. SysML Parametric diagrams and InstanceSpecifications, which can be used for the simulation as described in Step 1. of Section 9.2.5, are as follows:

- (i) The *test_parametrics.mdzip* sample demonstrates a simple mathematical model of block **A**.
- (ii) The *CylinderPipe.mdzip* sample demonstrates how to deal with multiple values. It shows the calculation for the cost of raw materials that will be used to manufacture the cylinder pipes. It also demonstrates the use of `OpaqueBehaviorAction` to execute the parametric.
- (iii) The *ActParIntegrate.mdzip* sample demonstrates the use of `OpaqueBehavior` to execute the parametric.
- (iv) *TradeTransformModel.mdzip*
- (v) *Financial.mdzip*
- (vi) *SCARA manipulator.mdzip* demonstrate the use of Parametric Simulation to evaluate the position of end-effector of the SCARA manipulator from the given angles of actuators.

NOTE

All of the sample projects of the Parametric Simulation Engine include the Execution Configurations package that contains two `ExecutionConfig` elements for normal and silent execution. You can select this `ExecutionConfig` class to start the Parametric Simulation Engine.

10. Interaction Between Engines

You can use all of the Simulation engines at the same time.

- `SendSignalAction` can send a signal to a trigger transition in an active State Machine (using Activity Simulation to control State Machine Simulation). *Stopwatch* is an example of such collaboration.
- Activation of the State can invoke entry/do/exit Activities. *testDoActivity.mdzip* and *test_regions.mdzip* are examples of such collaboration.

10.1 Stopwatch Sample

You can execute and then control the Stopwatch sample (**StopWatch.mdzip**), located in the `<md.install.dir>/samples/simulation/StopWatch` directory, by either (10.1.1) manually handling the `StopWatch` or (10.1.2) Activity Diagram.

10.1.1 Manual Execution

To execute and control the Stopwatch sample manually:

1. Open **StopWatch.mdzip**.
2. Right-click the **stopwatch_config** `ExecutionConfig` (in the **Config** Simulation Configuration Diagram) and then select **Simulation > Execute** in the context menu.
3. In the **Simulation Window**, press the **Run Execution** button to start the execution.
4. The Stopwatch mockup panel will then open. Also, the **StopWatch** is started with the “ready” state.
5. Either (i) click the **start** button in the mockup panel, or (ii) select the context (**StopWatch [ready]**) in the Variables Pane, and then select the **start** signal in the **Triggers:** combo box in

the **Simulation Window** to initiate the timer. The following buttons / signals can be used in different states:

- The **stop** button / signal will stop the timer if the current state is either **running** or **paused**.
- The **reset** button / signal will reset the timer to 0 if the current state is **stopped**.
- The **split** button / signal will stop displaying the elapsed time, but the timer still runs in background, if the current state is **running**.
- The **unsplit** button will redisplay the elapsed time if the current state is **paused**.

| | |
|-------------|---|
| NOTE | You need to close all of the current project control windows before switching to another project, or close the project or MagicDraw to ensure that the tool is fully functioning. |
|-------------|---|

10.1.2 Controlling Execution with Activity Diagram

If you do not want to trigger events manually, you can model the events instead. Activity diagrams allow you to model the SendSignalActions sequence and send Signals to any target Objects. Cameo Simulation Toolkit allows you to execute this particular activity and sends the signals to other active Engines. Therefore, whenever you start a State Machine Simulation, the transitions will be automatically triggered.

To execute the Stopwatch sample and control it by an Activity diagram:

1. Open **StopWatch.mdzip**.
2. Right-click the **Stopwatch Testcase** ExecutionConfig (in the **testcase** Simulation Configuration Diagram) and then select **Simulation > Execute** in the context menu.
3. In the **Simulation Window**, press the **Run Execution** button to start the execution.
4. The **Testcase scenario** Activity diagram will then be executed. Once the context created by the **Create Object** createObject is passed to the startObjectBehavior element, the Stopwatch mockup panel will then open, and a state machine execution will start.
5. You will see how the Activity diagram is executed; each SendSignalAction will be highlighted in red, the transition will be triggered, and the StopWatch system will start, pause, or stop according to the signals sent by each SendSignalAction.

11. Mathematical Engine

In order to perform a Parametrics Simulation on a SysML Parametrics Diagram, you will need a Mathematical Engine to evaluate the mathematical and logical expressions defined in the Constraints of Constraint Blocks which type the Constraint Properties on the diagram.

11.1 Math Console

Math Console in Cameo Simulation Toolkit is used to communicate with the Mathematical engine. Cameo Simulation Toolkit is designed to be used with various mathematical engines such as MATLAB^{®1}, OpenModelica², etc. You can create a new mathematical engine as a MagicDraw plugin and register with Cameo Simulation Toolkit.

1. MATLAB[®] is a registered trademark of The MathWorks, Inc.
2. Currently not supported.

The current Cameo Simulation Toolkit version comes with a Built-in Math Solver.

To use a selected mathematical engine in Cameo Simulation Toolkit:

1. Click **Options > Environment** on the main menu to open the **Environment Options** dialog.
2. Select **Simulation** in the left-hand side pane and select a mathematical engine from the **Mathematical Engine** field (Figure 98).

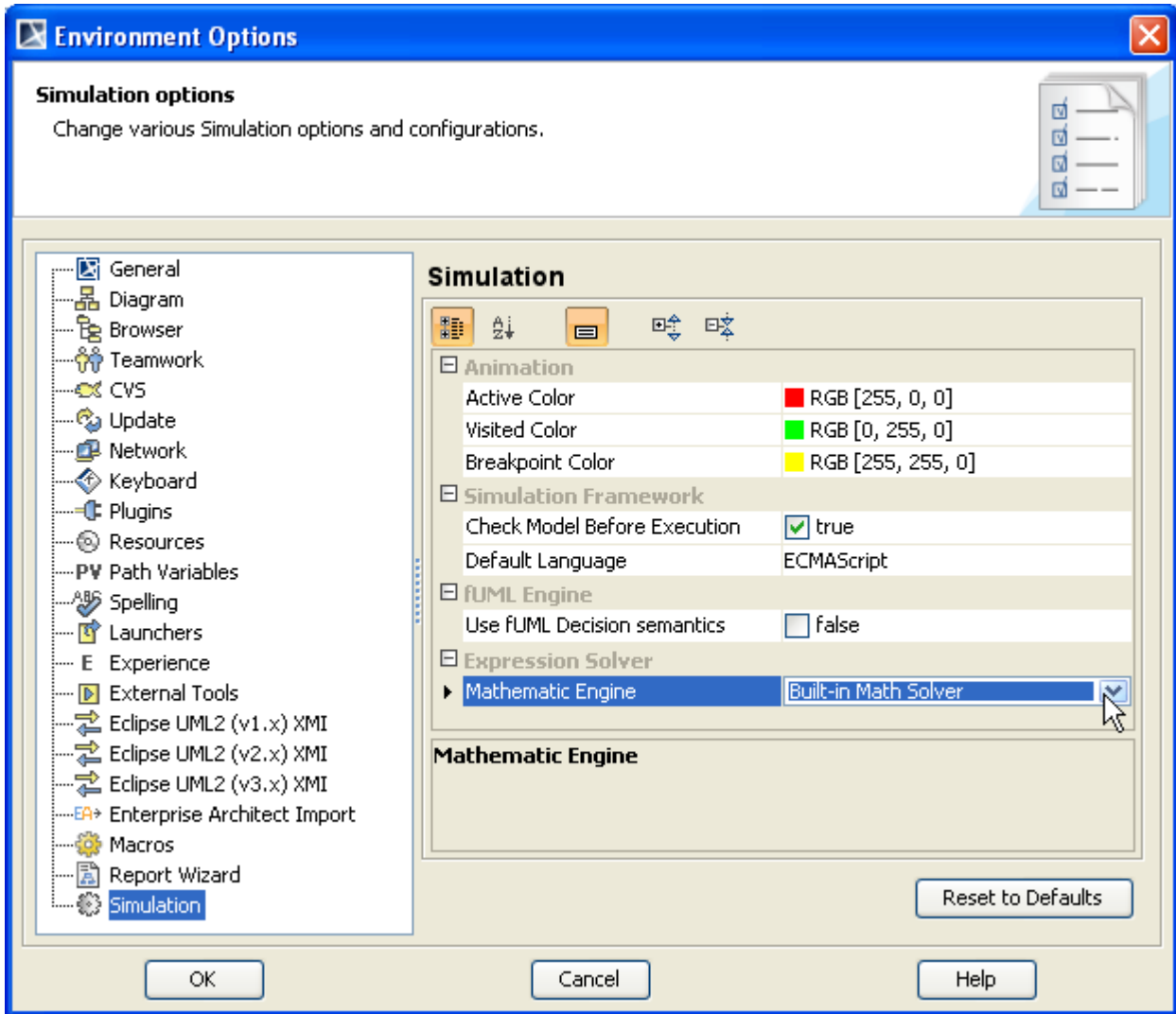


Figure 98 -- Selecting a Mathematical Engine in the Environment Options Dialog

You can start or stop the mathematical engine by clicking the **Start Math Engine** or **Stop Math Engine** button (Figure 99).

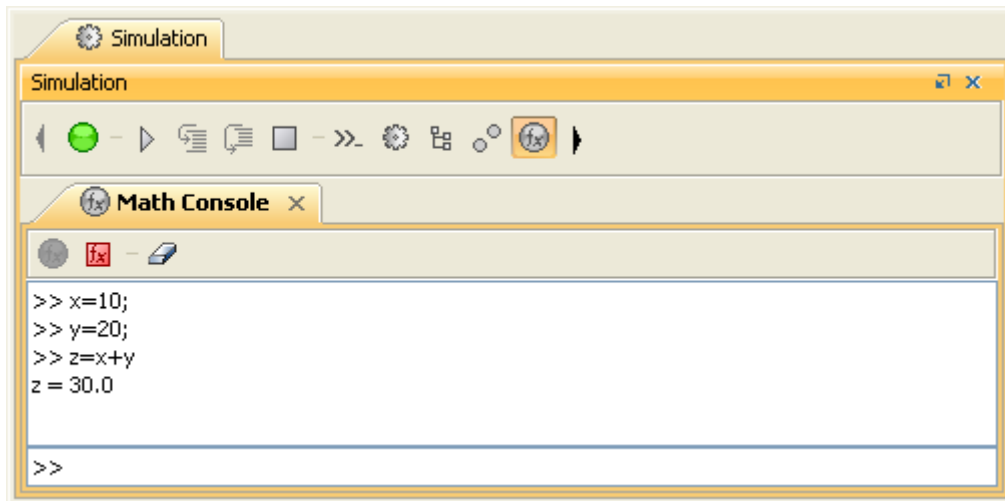





Figure 99 -- Math Console to Communicate with Mathematical Engine

| Button | Name | Function |
|---|-------------------|---|
|  | Start Math Engine | To start the Mathematical engine selected in the Environment Options dialog. |
|  | Stop Math Engine | To stop the Mathematical engine. |
|  | Clear | To clear all text displayed in the Math Console . |

11.2 Exchanging Values between Cameo Simulation Toolkit and Mathematical Engine

11.2.1 Exchanging values between Slot and Mathematic Environment

Cameo Simulation Toolkit allows you to exchange values between slots and the Mathematical engine through the diagram context menu on slot (Figure 100).

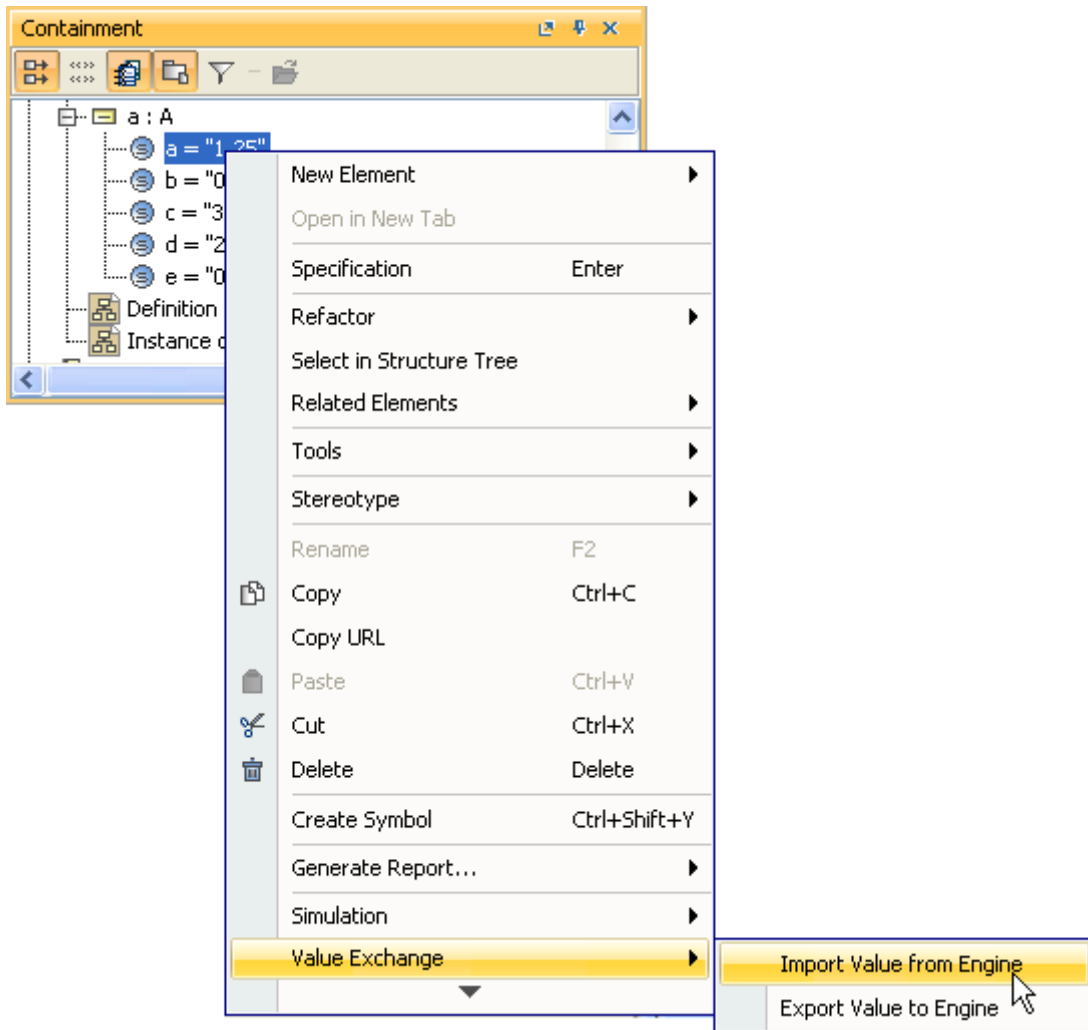


Figure 100 -- Context menu to Exchange Values with Mathematical Engine

To import a value from a mathematical engine to a slot:

1. Right-click the slot to which you will export a value and select **Import Value from Engine**. The **Value Exchange** dialog will be open (Figure 101).

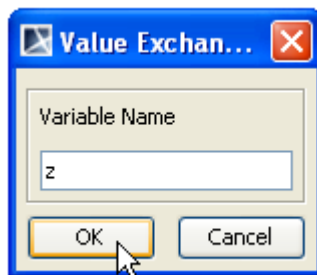


Figure 101 -- Value Exchange Dialog

2. Specify the variable name whose value you want to import and click **OK**.

To export a value from a slot to a Mathematical engine:

1. Right-click the slot whose value you want to export and select **Export Value to Engine**. The **Value Exchange** dialog will open.
2. Specify the variable name to which you will export the value and click **OK**.

11.2.2 Export Runtime Value to Mathematical Engine

During model execution, the runtime values which are shown in the Variable Pane could be exported to the Mathematical engine using context menu on the selected row as shown in Figure 102. Therefore, you can analyze these exported runtime values with Mathematica engine's functions, e.g. plot.

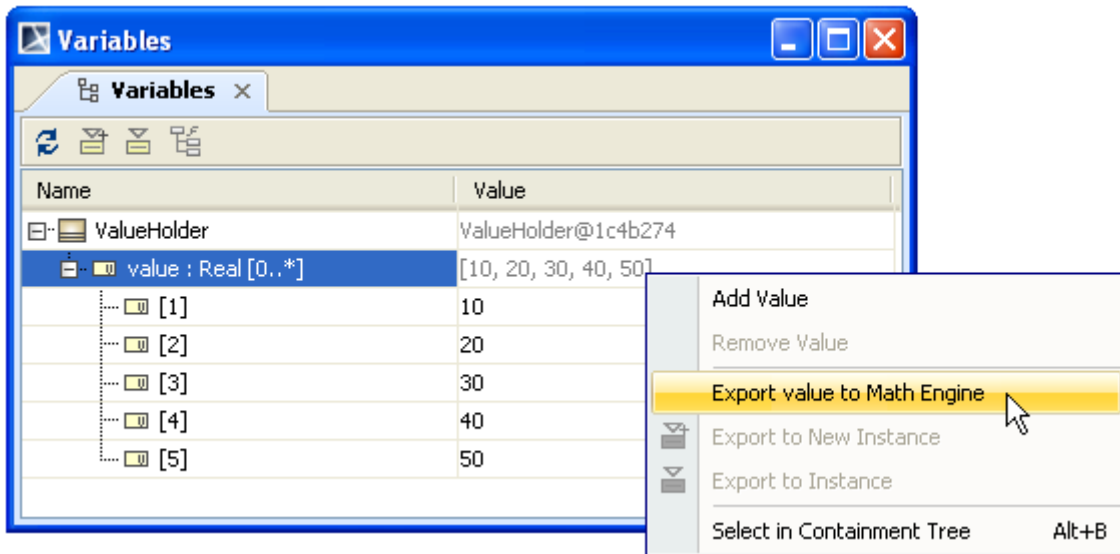


Figure 102 -- Show the context menu for exporting the runtime value to Mathematical engine.

To export the selected runtime value(s) to a Mathematical engine

1. Right-click the row that contains runtime value to be exported and select **Export value to Math Engine**. The **Value Exchanger** will open.
2. Specify the variable name to which you will export the value and click **OK**.

11.3 Built-in Math Solver

Built-in Math Solver is the default mathematical engine that comes with Cameo Simulation Toolkit. This engine can solve simple mathematical and logical expressions. You can use Built-in Math Solver to:

- evaluate the mathematical and logical expressions defined in the Constraints of Constraint Blocks for Parametrics Simulation on a SysML Parametrics Diagram.
- evaluate the mathematical and logical expressions in Math Console.

11.3.1 Using Built-in Math Solver in Math Console

You can type generic mathematical equations directly in the Math Console, for example:

x = 10;

y = 20;

z = x+y

z = 30 (the calculation result) will be displayed in the Math Console.

NOTE

The calculation results for the expressions that end with a semicolon (;) will be set to the corresponding variable in the Built-in Math Solver environment. It will not be displayed in the Math Console.

Or, if you type, for example, in the Math Console:

a = true;

b = false;

c = a & b;

The calculation result (**false**) will be assigned to the variable **c**, but it will not be displayed in the Math Console.

If an expression does not contain any assignment operator, the calculation result will be set to the variable '**ans**', for example:

x = 10;

20 + x

ans = 30 will be displayed in the Math Console.

You can calculate multiple expressions at the same time by typing a semicolon (;) at the end of each expression, for example:

x = 10; y = 20; z = x+y; a = z / x

a = 3 will be displayed in the Math Console.

11.3.2 Variables

The variables (operands) that can be used in Built-in Math Solver must conform to the following naming conventions:

- The characters in a variable name must be **a-z**, **A-Z**, or **0-9**.
- The first character must **not** be a **number**.
- Variable name must **not** be **Constants** ("**E**" or "**PI**")
- Variable names must **not** be **Functions** ("**sqrt**", "**sin**", "**cos**").
- Variable names must **not** be **Operators** ("**+**", "**-**", "*****", "**/**").

11.3.3 Values

Valid values that can be used in an expression are: (11.3.3.1) Real Number, (11.3.3.2) Complex Number, (11.3.3.3) Boolean, and (11.3.3.4) Matrix.

11.3.3.1 Real Number

x = 3.14159

y = 2

11.3.3.2 Complex Number

c = 3 + 4i

$$d = 1.25 + 0.25i$$

NOTE

The 'i' character in an expression can be parsed as the imaginary unit or character of a variable name. If 'i' is placed after a number and the next character is neither an alphabet nor number, it will be parsed as an imaginary unit (otherwise parsed as a variable). For example:

- $ca = 1i$ 'i' is parsed as an imaginary unit.
- $cb = i$ 'i' is parsed as a variable.
- $cx = 3.25i$ 'i' is parsed as an imaginary unit.
- $cy = 4i4$ 'i' is parsed as the first character of a variable name 'i4'

11.3.3.3 Boolean

$a = \text{true}$

$b = \text{false}$

11.3.3.4 Matrix

$U = [1.0, 2.0, 3.0; 4.0, 5.0, 6.0; 7.0, 8.0, 9.0]$

$A = [\text{true}; \text{false}; \text{false}; \text{true}]$

NOTE

- You can add a matrix to the built-in math solver by the following syntax (a semi-colon is used as row separator and comma or space is used as a comma separator). From the expression in 11.3.3.4 above:

$U = [1.0, 2.0, 3.0; 4.0, 5.0, 6.0; 7.0, 8.0, 9.0]$

$$U = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix}$$

$A = [\text{true}; \text{false}; \text{false}; \text{true}]$

$$A = \begin{bmatrix} \text{true} \\ \text{false} \\ \text{false} \\ \text{true} \end{bmatrix}$$

- You can refer to a matrix element by specifying the row index and column index in round brackets after a matrix name, for example (from U above):

$U(1, 1)$ is 1.0

$U(2, 3)$ is 6.0

- You can also refer to a matrix element by specifying only one index in round brackets after a matrix name. If this is the case, the matrix will be considered as a column-major order matrix. The elements on the given column-major order index will be returned. For example (from U above):

$U(2)$ is 4.0

$U(6)$ is 8.0

11.3.4 Constants

| Constant | Value |
|-----------|---|
| E | The real value that is closer than any other to e , the base of the natural logarithms. |
| PI | The real value that is closer than any other to pi , the ratio of the circumference of a circle to its diameter. |

11.3.5 Operators

| | |
|-------------|--|
| NOTE | <ul style="list-style-type: none"> • x and y represent numerical values or variables. • m n and p represent integer values or variables. • a and b represent boolean values or variables. • U and V represent matrices of numerical values. • A and B represent matrices of boolean values. |
|-------------|--|

11.3.5.1 Arithmetic Operators

| Operator | Operator Name | Syntax |
|----------|----------------|---|
| + | Addition | $x+y$ $U + V$ (U and V are mxn matrices) |
| - | Subtraction | $x-y$ $U - V$ (U and V are mxn matrices) |
| * | Multiplication | $x*y$ $U*V$ (U is mxn matrix and V is nxp matrix) |
| / | Division | x/y |
| % | Modulus | $m%n$ $U \% V$ (U and V are mxn matrices of integer values) <i>This operator operates element-wise on matrices.</i> |
| ! | Factorial | $m!$ |
| ^ | Power | x^y |

b) Assignment Operators

| Operator | Operator Name | Syntax |
|----------|---------------|-------------------------|
| = | Assignment | $x=y$ $a=b$ $U=V$ |

11.3.5.2 Comparison Operators

| Operator | Operator Name | Syntax |
|----------|------------------|----------------------------------|
| > | Greater | $x > y$ $U > V$ |
| < | Less | $x < y$ $U < V$ |
| >= | Greater or Equal | $x \geq y$ $U \geq V$ |
| <= | Less of Equal | $x \leq y$ $U \leq V$ |
| == | Equality | $x == y$ $a == b$ $U == V$ |
| != | Inequality | $x != y$ $a != b$ $U != V$ |

| | |
|-------------|---|
| NOTE | All comparison operators operate element-wise on matrices, for example: $A = [1; 2; 3]$ $B = [3; 2; 1]$ Then $A > B$ is [false false true]; |
|-------------|---|

11.3.5.3 d) Boolean Operators

| Operator | Operator Name | Syntax |
|----------|--------------------|------------------------------|
| ! | NOT | $!a$ $!A$ |
| & | AND | $a \& b$ $A \& B$ |
| | OR | $a b$ $A B$ |
| ^ | XOR (exclusive OR) | $a \wedge b$ $A \wedge B$ |

| | |
|-------------|--|
| NOTE | All boolean operators operate element-wise on matrices, for example: $A = [\text{true}; \text{true}; \text{false}; \text{false}];$ $B = [\text{true}; \text{false}; \text{true}; \text{false}];$ Then $A \& B$ is [true; false; false; false]; |
|-------------|--|

11.3.6 Functions

| | |
|-------------|--|
| NOTE | <ul style="list-style-type: none"> • x and y represent real values or variables. • c and d represent complex values or variables. • m and n represent integer values or variables. • U represent matrix of values. <p>For the function that operates element-wise on matrices, a matrix can be passed to the function as its argument, for example:</p> <pre style="margin-left: 40px;">X = [1, -2, 3; -4 5 -6; 7 -8 9]; Y = abs(X) result: Y = [1 2 3; 4 5 6; 7 8 9]</pre> |
|-------------|--|

| Function | Syntax | Function |
|--------------|----------------------|---|
| abs | abs(x) abs(c) | To return the absolute value of x or the complex modulus of c. <i>This function operates element-wise on matrices.</i> |
| acos | acos(x) acos(c) | To return the arc cosine of an angle, in the range of 0.0 through pi. All angles are in radians. <i>This function operates element-wise on matrices.</i> |
| acosd | acosd(x) acosd(c) | To return the inverted cosine of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| acosh | acosh(x) acosh(c) | To return the inverted hyperbolic cosine of a given value. <i>This function operates element-wise on matrices.</i> |
| acot | acot(x) acot(c) | To return the inverted cotangent of a given value. <i>This function operates element-wise on matrices.</i> |
| acotd | acotd(x) acotd(c) | To return the inverted cotangent of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| acoth | acoth(x) acoth(c) | To return the inverted hyperbolic cotangent of a given value. <i>This function operates element-wise on matrices.</i> |
| acsc | acsc(x) acsc(c) | To return the inverted cosecant of a given value. <i>This function operates element-wise on matrices.</i> |
| acscd | acscd(x) acscd(c) | To return the inverted cosecant of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| acsch | acsch(x) acsch(c) | To return the inverted hyperbolic cosecant of a given value. <i>This function operates element-wise on matrices.</i> |
| asec | asec(x) asec(c) | To return the inverted secant of a given value. <i>This function operates element-wise on matrices.</i> |

| Function | Syntax | Function |
|--------------|----------------------------|---|
| asecd | asecd(x) asecd(c) | To return the inverted secant of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| asech | asech(x) asech(c) | To return the inverted hyperbolic secant of a given value. <i>This function operates element-wise on matrices.</i> |
| asin | asin(x) asin(c) | To return the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$. <i>This function operates element-wise on matrices.</i> |
| asind | asind(x) asind(c) | To return the inverted sine of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| asinh | asinh(x) asinh(c) | To return the inverted hyperbolic sine of a given value. <i>This function operates element-wise on matrices.</i> |
| atan | atan(x) atan(c) | Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$. <i>This function operates element-wise on matrices.</i> |
| atan2 | atan2(x, y) atan2(U, V) | To return the arc tangent of an angle, in the range of $-\pi$ through π . atan2(U, V) returns a matrix the same size as U and V containing the element-by-element, inverse tangent of the real parts of U and V. |
| atand | atand(x) atand(c) | To return the inverted tangent of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| atanh | atanh(x) atanh(c) | To return the inverted hyperbolic tangent of a given value. <i>This function operates element-wise on matrices.</i> |
| ceil | ceil(x) | To return the smallest (closest to negative infinity) value that is not less than the value of x and is equal to a mathematical integer. <i>This function operates element-wise on matrices.</i> |
| conj | conj(c) | To return the conjugated value of c. <i>This function operates element-wise on matrices.</i> |
| cos | cos(x) cos(c) | To return the trigonometric cosine of an angle. <i>This function operates element-wise on matrices.</i> |
| cosd | cosd(x) cosd(c) | To return the cosine of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| cosh | cosh(x) cosh(c) | To return the hyperbolic cosine of a given value. <i>This function operates element-wise on matrices.</i> |
| cot | cot(x) cot(c) | To return the cotangent of a given value. <i>This function operates element-wise on matrices.</i> |
| cotd | cotd(x) cotd(c) | To return the cotangent of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |

| Function | Syntax | Function |
|----------------------|-----------------------|---|
| coth | coth(x) coth(c) | To return the hyperbolic cotangent of a given value. <i>This function operates element-wise on matrices.</i> |
| count | count(U) | To return the number of elements of a given matrix |
| csc | csc(x) csc(c) | To return the cosecant of a given value. <i>This function operates element-wise on matrices.</i> |
| cscd | cscd(x) cscd(c) | To return the cosecant of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| csch | csch(x) csch(c) | To return the hyperbolic cosecant of a given value. <i>This function operates element-wise on matrices.</i> |
| diag | diag(U) diag(U, m) | To return a diagonal matrix and diagonals of a matrix. If U is a row matrix or a column matrix of n elements, this function will return a square matrix of order n+abs(m), with the elements of U on the kth diagonal. <ul style="list-style-type: none"> • k = 0 represents the main diagonal. • k > 0 above the main diagonal. • k < 0 below the main diagonal. If U is a square matrix, this function will return a column matrix formed by the elements of the kth diagonal of U. |
| exp | exp(x) exp(c) | To return the Euler's number e raised to the power of a or c. <i>This function operates element-wise on matrices.</i> |
| eye | eye(m) | To return the identity matrix of dimension mxm. |
| factorial | factorial(m) | To return the factorial of m value. |
| floor | floor(x) floor(X) | To return the largest (closest to positive infinity) value that is not greater than the value of x and is equal to a mathematical integer. <i>This function operates element-wise on matrices.</i> |
| IEEEremainder | IEEEremainder(x, y) | To compute the remainder operation in two arguments as prescribed by the IEEE 754 standard. |
| imag | imag(c) | To return the real value of the imaginary part of a given complex number. <i>This function operates element-wise on matrices.</i> |
| invert | invert(U) | To return the invert matrix or pseudo invert matrix of a given matrix. <ul style="list-style-type: none"> • If the given matrix is a square matrix, the invert matrix of U will be returned using the LU factorization. • If the given matrix is not a square matrix, the pseudo inverted matrix will be returned using the QR factorization. |
| linsolve | linsolve(U, V) | X = linsolve(U,V) solves the linear system U*X = V using the LU factorization with partial pivoting when U is a square matrix |
| ln | ln(x) ln(c) | To return the natural logarithm (base e) of a given value. <i>This function operates element-wise on matrices.</i> |
| log | log(x) log(c) | To return the natural logarithm (base e) of a given value. <i>This function operates element-wise on matrices.</i> |

| Function | Syntax | Function |
|---------------|---|--|
| log10 | log10(x) log10(c) | To return the logarithm base 10 of a given value. <i>This function operates element-wise on matrices.</i> |
| log2 | log2(x) log2(c) | To return the logarithm base 2 of a given value. <i>This function operates element-wise on matrices.</i> |
| max | max(x, y) max(c, d) max(U) max(U, V) | To return the greater of two given values. <ul style="list-style-type: none"> max(U) returns the largest element of a given matrix. max(U, V) returns a matrix the same size as U and V with the largest elements taken from U or V. The dimensions of U and V must be the same. |
| mean | mean(U) | To return mean or average value of a given matrix. <ul style="list-style-type: none"> U is row or column matrix: mean(U) returns the mean value of all elements in the given matrix. U is 2-D matrix: mean(U) returns row matrix that contains the mean value of each column of the given matrix. |
| median | median(U) | To return median value of a given matrix. <ul style="list-style-type: none"> U is row or column matrix: median(U) returns the median value of all elements in the given matrix. U is 2-D matrix: median(U) returns row matrix that contains the median value of each column of the given matrix. |
| min | min(x, y) min(c, d) min(U) min(U, V) | To return the smaller of two given values. <ul style="list-style-type: none"> min(U) returns the smallest element of a given matrix. min(U, V) returns a matrix the same size as U and V with the smallest elements taken from U or V. The dimensions of U and V must be the same. |
| ones | ones(m, n) | To return the mxn matrix of all ones. |
| pow | pow(x, y) pow(U, c) pow(c, d) | To return the value of the first argument raised to the power of the second argument. <i>This function operates element-wise on a given matrix U.</i> |
| random | random() | To return a real value with a positive sign, greater than or equal to 0.0 but less than 1.0. |
| real | real(c) | To return the real value of the real part of a given complex number. <i>This function operates element-wise on matrices.</i> |
| rint | rint(x) | To return the value that is closest in value to an argument and is equal to a mathematical integer. <i>This function operates element-wise on matrices.</i> |
| round | round(x) | To return the closest value to an argument and is equal to a mathematical integer. <i>This function operates element-wise on matrices.</i> |
| sec | sec(x) sec(c) | To return the secant of a given value. <i>This function operates element-wise on matrices.</i> |
| secd | secd(x) secd(c) | To return the secant of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |
| sech | sech(x) sech(c) | To return the hyperbolic secant of a given value. <i>This function operates element-wise on matrices.</i> |

| Function | Syntax | Function |
|-------------|-------------------------------|--|
| sin | sin(x) sin(c) | To return the trigonometric sine of an angle. <i>This function operates element-wise on matrices.</i> |
| sind | sind(x) sind(c) | To return the sine of a given value, expressed in degree <i>This function operates element-wise on matrices.</i> |
| sinh | sinh(x) sinh(c) | To return the hyperbolic sine of a given value. <i>This function operates element-wise on matrices.</i> |
| size | size(U) size(U, m) | To return the size of a given matrix. If only matrix is passed to the function as an argument, the returned value is the matrix of size 1x2. The first element is the number of row and the second element is the number of column. If the second parameter (m) is specified, this function will return the size of the m th dimension of a given matrix as a scalar value. The second argument can be 1 or 2 (1 for row size, 2 for column size). For example: <div style="text-align: center;"> U = [1, 2, 3; 4, 5, 6]; size(U) is [2, 3] size(U, 1) is 2 size(U, 2) is 3 </div> |
| sort | sort(U) sort(U, 'descend') | To sort the elements of a given matrix in ascending or descending order. If the second argument is specified with 'ascend' or 'descend', the elements will be ascending sorted or descending sorted respectively. If this function is called without second argument, the elements are sorted in ascending order. <ul style="list-style-type: none"> • U is row or column matrix: sort(U) and sort(U, ascend) sort all elements in the given matrix. • U is 2-D matrix: std(U) and std(U,flag) sort elements in each column of the given matrix. |
| sqrt | sqrt(x) sqrt(c) | To return the correctly rounded positive square root of a double value. <i>This function operates element-wise on matrices.</i> |
| std | std(U) std(U, flag) | To return standard deviation of a given matrix. The 'flag' argument can be 0 or 1. It specifies the method for calculating the standard deviation. If flag = 0, the standard deviation is normalized by N-1. If flag = 1, the standard deviation is normalized by N. Where N is number of data. The value of flag will be 0 by default. <ul style="list-style-type: none"> • U is row or column matrix: std(U) and std(U, flag) returns the standard deviation of all elements in the given matrix. • U is 2-D matrix: std(U) and std(U,flag) returns row matrix that contains the standard deviation of each column of the given matrix. |
| sum | sum(U) | To return the summation of all elements in matrix U. |
| tan | tan(x) tan(c) | To return the trigonometric tangent of an angle. <i>This function operates element-wise on matrices.</i> |
| tand | tand(x) tand(c) | To return the tangent of a given value, expressed in degree. <i>This function operates element-wise on matrices.</i> |

| Function | Syntax | Function |
|------------------|------------------------------|--|
| tanh | tanh(x) tanh(c) | Returns hyperbolic tangent of the given value. <i>This function operates element-wise on matrices.</i> |
| toDegrees | toDegrees(x) toDegrees(c) | To convert an angle measured in radians to an approximately equivalent angle measured in degrees. <i>This function operates element-wise on matrices.</i> |
| toRadians | toRadians(x) toRadians(c) | To convert an angle measured in degrees to an approximately equivalent angle measured in radians. <i>This function operates element-wise on matrices.</i> |
| transpose | transpose(U) | To return the transposition of the given matrix |
| zeros | zeros(m, n) | To return the mxn matrix of all zeros. |

11.3.7 Built-in Math Solver API for User-Defined Functions

Built-in Math Solver provides the API for the user to create the user-defined functions. These functions can be used in mathematical or logical expressions of constraint elements. To create the user-defined function, you have to create a new MagicDraw plugin. Then, create a Java class that implements the UserDefinedFunction interface. Finally, register the created class to the built-in math solver.

You can see the **MagicDraw OpenAPI UserGuide.pdf** in the `<md.install.dir>/manual` directory for the information about how to create a new MagicDraw plugin.

11.3.7.1 Understanding UserDefinedFunction interface

Cameo Simulation Toolkit provides the Java interface, which is UserDefinedFunction interface, for creating the user-defined functions in built-in math solver. It has 3 methods that must be implemented in the Java class.

String getName()

This method returns name of the user-defined function. It will be used for calling to the user-defined function in mathematical expression.

boolean isValidInputParameters(List<Value> parameters)

This method will be called by built-in math solver for validating the input parameters before perform function operation. The '*parameters*' are the input parameters which are passed to the user-defined function. If all of them are valid, this method returns **true**. Otherwise, **false** is returned.

Value performFunction(List<Value> parameters)

This method will be called by built-in math solver for perform the user-defined function operation. The '*parameters*' are the input parameters which are passed to the user-defined function. The implemented code for calculation the result value from the given input parameters should be placed in this method.

For example, the user-defined function for polynomial value evaluation from the given polynomial coefficient and the value that polynomial will be evaluated.

So, we create a new Java class which is named with "PolyvalFunctionDescriptor". It must implements the UserDefinedFunction interface.

```

public class PolyvalFunctionDescriptor implements UserDefinedFunction {
    public static final String name = "polyval";
    @Override
    public String getName() {
        // Return name of the function
        return PolyvalFunctionDescriptor.name;
    }
    @Override
    public boolean isValidInputParameters(List<Value> parameters) {
        // This function requires two input parameters
        if(parameters.size() == 2) {
            // The first parameter must be the value node that contains a matrix of complex values.
            if((parameters.get(0) instanceof Value) && (((Value)parameters.get(0)).isMatrix())) {
                // This matrix must be row matrix or column matrix
                ComplexMatrix A = ((Value)parameters.get(0)).getMatrix();
                if((A.getRowCount() == 1) || (A.getColumnCount() == 1)) {
                    // The second parameter must be the value node that contains a complex value.
                    if((parameters.get(1) instanceof Value) && (((Value)parameters.get(1)).isComplex())) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
    @Override
    public Value performFunction(List<Value> parameters) throws Exception {
        // Get the polynomial coefficient matrix
        ComplexMatrix A = ((Value)parameters.get(0)).getMatrix();
        // and get the value x
        Complex x = ((Value)parameters.get(1)).getComplex();
        // Obtain the order of polynomial n (the number of elements of p is n+1).
        Therefore,
        int n = A.getElementCount() - 1;
        // Create complex value for storing result of calculation
        Complex result = new Complex(0.0, 0.0);
        for(int i=0; i<=n; i++) {
            // Get i-th order coefficient.
            Complex ai = A.getElement(n - i);
            // Get the value of ai*x^i
            Complex tmp = ComplexMathHelper.multiply(ai, ComplexMathHelper.pow(x, (double)i));
            // Add to result.
            result = ComplexMathHelper.add(result, tmp);
        }
        // Create a value node that contains the result of calculation.
        return new Value(result);
    }
}

```

11.3.7.2 Register user-defined function to Built-in Math Solver using SimpleMathEngine class

The SimpleMathEngine class represents the built-in math solver. The Java class which implements the UserDefinedFunction interface must be registered to this class when the created plugin is initialized.

For example, if the class UDFSsamplePlugin is inherited from Plugin then:

```
package com.nomagic.magicdraw.simulation.udfsample;
import com.nomagic.magicdraw.plugins.Plugin;
import com.nomagic.magicdraw.simulation.expsolver.mathengine.SimpleMathEngine;

public class UDFSsamplePlugin extends Plugin {
    @Override
    public void init() {
        SimpleMathEngine.registerUserDefinedFunction(new PolyvalFunctionDescriptor());
    }

    @Override
    public boolean close() {
        SimpleMathEngine.unregisterUserDefinedFunction(PolyvalFunctionDescriptor.name);
        return true;
    }

    @Override
    public boolean isSupported() {
        return true;
    }
}
```

11.4 Using MATLAB® as a Mathematical Solver

Cameo Simulation Toolkit can use MATLAB®, which is installed on the local machine, for solving mathematical expressions.

| | |
|-------------|---|
| NOTE | The current version of Cameo Simulation Toolkit can integrate with MATLAB® only on Microsoft Windows and Mac OS 10.6 (Snow Leopard) platform. This feature cannot be used on Linux. |
|-------------|---|

11.4.1 Setting up system for calling MATLAB® from Cameo Simulation Toolkit

Microsoft Windows 32-bit and 64-bit

1. Install MATLAB® to Microsoft Windows.
2. Register MATLAB® component to Microsoft Windows by calling “**matlab /regserver**” in command prompt window.
 - Press Win + R to open Run dialog
 - Type “cmd” in open combo box and click OK button to open command prompt window.

- Type “matlab /regserver” and enter to register MATLAB® component to window.

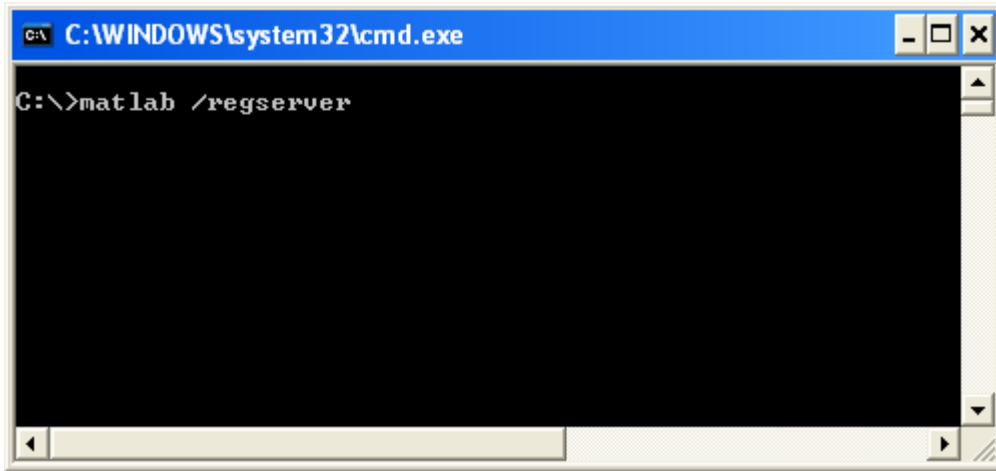


Figure 103 -- Register MATLAB® Component using Command Prompt Window.

3. Add path of MATLAB® bin and bin/win32 (or bin/win64 for Microsoft Windows 64-bit) folder to the **Path** environment variable.
 - Double-click **System** in **Control Panel** to open the **System Properties** dialog. Select **Advanced** tab.

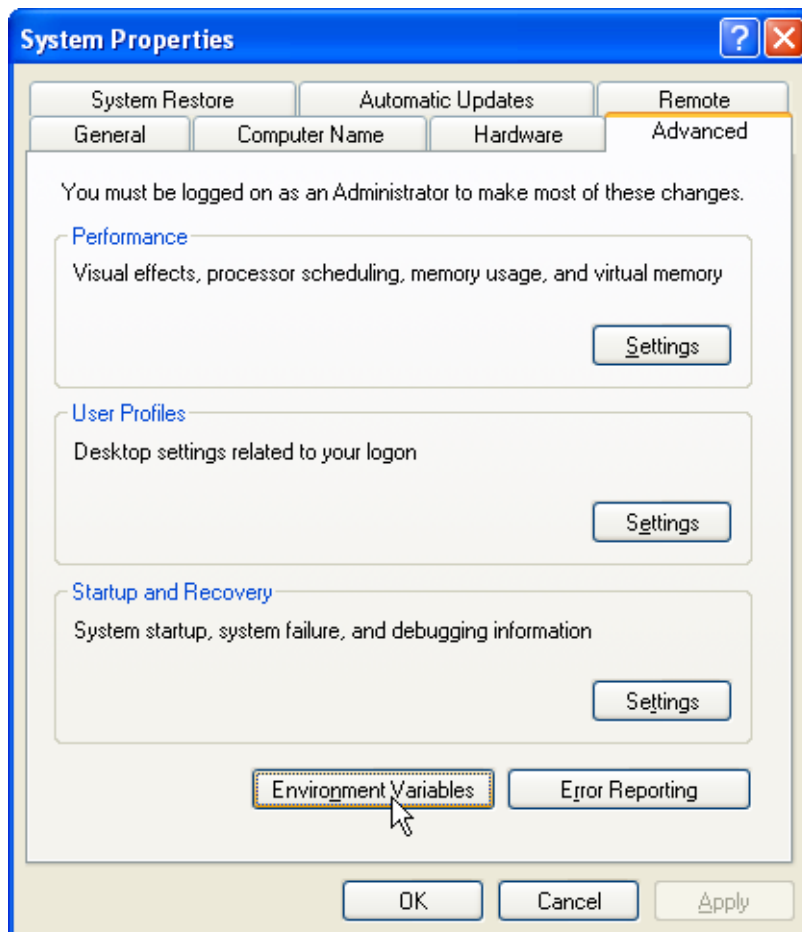


Figure 104 -- System Properties Dialog for Setting Environment Variables

- Click **Environment Variables** button to open the **Environment Variables** dialog.

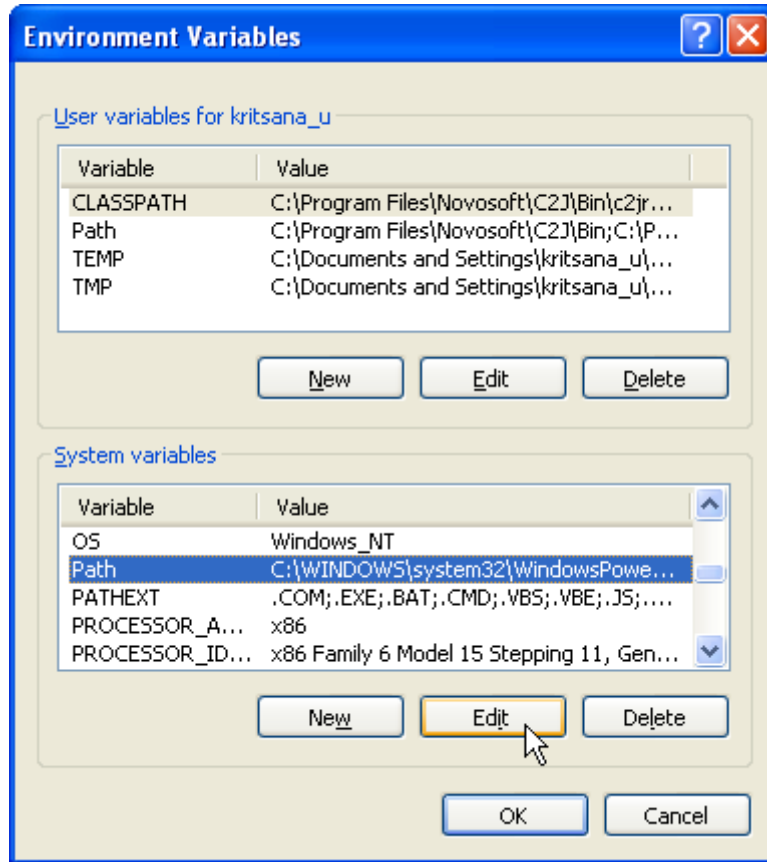


Figure 105 -- Environment Variables Dialog for Setting System Path Variable.

- Select **Path** in System variables group and click **Edit** button to open the Edit System Variable dialog.
- Insert path to MATLAB® bin and bin/win32 folder (or bin/win64 for Microsoft Windows 64-bit) at the begin of Variable value, e.g. "C:\Program Files\MATLAB\R2010b\bin;C:\Program Files\MATLAB\R2010b\bin\win32;".

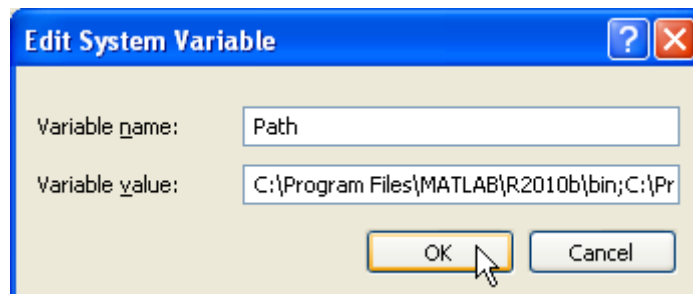


Figure 106 -- Edit System Variable

- Click OK to all opened dialog.

4. Restart Microsoft Windows.

Mac OS 10.6 (Snow Leopard)

1. Install MATLAB® to Mac OS 10.6
2. Make Finder to show all files by calling the following command in Terminal:
 - \$ defaults write com.apple.finder AppleShowAllFiles TRUE
 - \$ killall Finder
3. Add **DYLD_LIBRARY_PATH** variable to Mac OS
 - Create empty text file with name **launchd.conf** in **/etc** folder
 - Open it with text editor, e.g. TextEdit, and put the following text (all text must be in a single line):

```
setenv DYLD_LIBRARY_PATH /Applications/MATLAB_R2010b.app/bin/maci64: /Applications/  
MATLAB_R2010b.app/runtime/maci64
```

- Save text file with the name **launchd.conf** to desktop.
 - Move the **launchd.conf** to **/etc** folder
4. Create link to MATLAB® executable file in **/usr/bin** if it does not exist. By using Terminal, call the following commands:
 - \$ cd /usr/bin
 - \$ ln -s /Applications/MATLAB_R2010b.app/bin/matlab matlab
 5. Reset Finder by calling the following commands in Terminal:
 - \$ defaults write com.apple.finder AppleShowAllFiles FALSE
 - \$ killall Finder
 6. Restart Mac OS.

11.4.2 Selecting MATLAB® as Mathematical Solver for Cameo Simulation Toolkit

You can switch the mathematical solver to MATLAB® by setting the **Mathematical Engine** field in the environment option to **MATLAB®** (Figure 107).

- Select **Options > Environment** in main menu bar. The **Environment Options** dialog will open.
- Select **Simulation** in the left pane.
- In **Mathematic Engine**, select **MATLAB®** from drop-down list.
- Click OK.

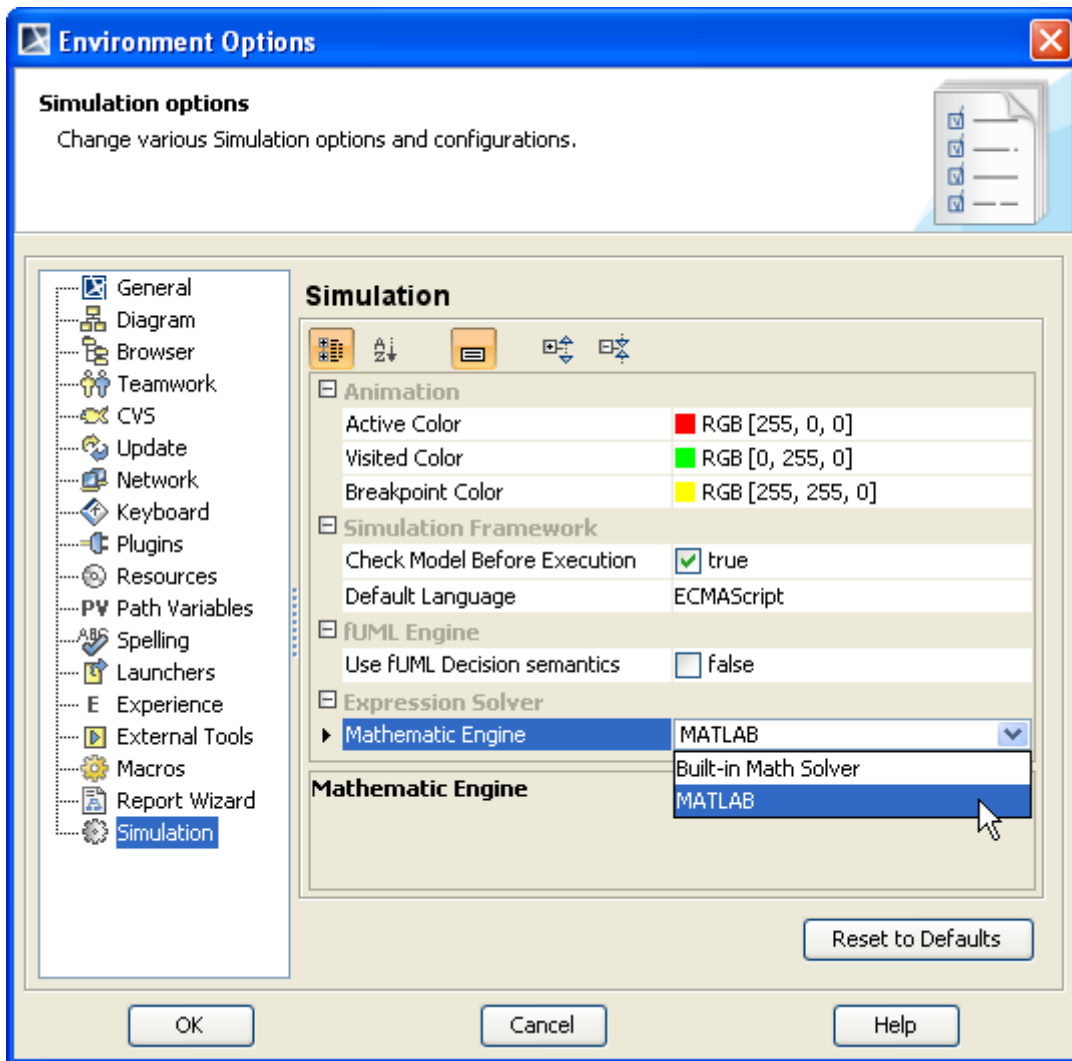


Figure 107 -- Setting MATLAB® as Mathematical Engine in Environment Options

12. Action Languages

You can use multiple languages as action languages in the expressions anywhere in a model. Cameo Simulation Toolkit supports Javascript, Beanshell, Groovy, and Jython by default. You can also download and plug other JSR233 compatible language implementations. Mathematical expressions can be solved using external math solvers, for example, MATLAB® if the integrators present. Such integrators will be provided in the subsequent release(s) of Cameo Simulation Toolkit.

Any value specification in a model (like guards, constraints, decisions, default values, opaqueBehaviors) can have the opaque expressions defined using an action language. The languages that are supported include:

- Javascript
- Beanshell
- Groovy
- Jython
- JRuby
- OCL

- Java binaries
- Math (see 11. Mathematical Engine)
- Other additional downloadable JSR-223 script engines (see <https://scripting.dev.java.net/>)