CONTENTS

1 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 21
   3.3 Simulation Time and Simulation Clock 22
   3.4 Automatic Start Active Objects 23
   3.5 User Interface Prototyping 25
   3.6 UI Modeling Diagram Execution 25
   3.7 ActiveImage and ImageSwitcher 28
   3.8 Time Series Chart 29
   3.9 Nested UI Configuration 31
      3.9.1 NestedUIConfig Stereotype Representing a Part of Execution Context 31
      3.9.2 NestedUIConfig Stereotype Representing a Part Using UI Configuration 34

4. Animation 39
   4.1 Active, Visited, and Last Visited Elements 39
   4.2 Customizing Animation Colors 40

5. Simulation Debugging 41
   5.1 Understanding Simulation Sessions 41
   5.2 Simulation Debugger 42
   5.3 Simulation Console 43
      5.3.1 Console Pane 43
      5.3.2 Simulation Information 44
      5.3.3 Simulation Log File 45
   5.4 Runtime Values Monitoring 45
      5.4.1 Variables Pane 45
      5.4.2 Monitoring Runtime Value with Time Series Chart 46
      5.4.3 Runtime Object Created from InstanceSpecification 48
      5.4.4 Exporting Runtime Objects to InstanceSpecification 48
   5.5 Breakpoints 50
      5.5.1 Adding Breakpoints 51
      5.5.2 Removing Breakpoints 52
   5.6 Disabling Updates in Simulation Panes 54

6. Validation and Verification 54

7. State Machine Simulation 56
   7.1 Supported Elements 56
   7.2 Adapting Models for State Machine Simulation 57
      7.2.1 Defining Trigger on Transition 57
      7.2.2 Using Guard on Transition 58
      7.2.3 Behaviors on Entry, Exit, and Do Activity of State 59
   7.3 Running State Machine Execution 59
1. Getting Started

Cameo Simulation Toolkit is a MagicDraw plugin which provides a unique set of tools supporting the standard-
ized construction, verification, and execution of computational complete models based on a foundational sub-
set of the 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
directly because the real system may have not been completely defined or available, or it cannot be experi-
mented due to costs, time, resources, or any other 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 mockup of the intended user interface. The solutions provided by Cameo
Simulation Toolkit allow you to predict how the system responds to user interactions, 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 function-
alities of Cameo Simulation Toolkit are as follows:

(i) Simulation Window:
   • Toolbars and Debugger Pane: to control an execution or simulation
   • Simulation Console: to execute log outputs and command lines for active engines
   • Sessions Pane: to select particular sessions of executions
   • Variables Pane: to monitor the runtime values of each execution session
   • Math Console: to communicate with mathematical engines
   • Breakpoints Pane
   • Triggers Options

(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 (including the simulation toolbars, context menu, and panes) and Open API for execution.

(ii) **State Machine Execution Engine**: The W3C SCXML (State Charts XML) standard, which is an open-source Apache implementation.

(iii) **Activities Execution Engine**: The OMG fUML (a foundational subset of the Executable UML) standard.

(iv) **Parametrics Execution Engine**: Enabling Cameo Simulation Toolkit to execute SysML parametric diagrams. The 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, import, and install 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 the connection to start MagicDraw updates and resources.

2. Under the **Plugins (commercial)** group, select the **Cameo Simulation Toolkit** check box (with the “Available” status) and click **Download/Install**.

3. Once the installation is complete, a dialog of complete installation notification will open. Click **OK**.

4. 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>`.


### 2. Model Execution

Cameo Simulation Toolkit allows you to execute 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 models.
Table 1 -- The Current Supported Execution Engines

<table>
<thead>
<tr>
<th>Execution Engine</th>
<th>Supported Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity Execution Engine</td>
<td>• Activity</td>
</tr>
<tr>
<td></td>
<td>• Activity Diagram</td>
</tr>
<tr>
<td></td>
<td>• Class whose classifier behavior is an Activity</td>
</tr>
<tr>
<td></td>
<td>• InstanceSpecification of a class whose classifier behavior is an Activity</td>
</tr>
<tr>
<td>State Machine Execution Engine</td>
<td>• State Machine</td>
</tr>
<tr>
<td></td>
<td>• State Machine Diagram</td>
</tr>
<tr>
<td></td>
<td>• Class whose classifier behavior is a State Machine</td>
</tr>
<tr>
<td></td>
<td>• InstanceSpecification of a Class whose classifier behavior is a State Machine</td>
</tr>
<tr>
<td>Parametrics Execution Engine</td>
<td>• Block that contains Constraint Properties</td>
</tr>
<tr>
<td></td>
<td>• SysML Parametrics Diagram</td>
</tr>
<tr>
<td></td>
<td>• InstanceSpecification of a Block that contains Constraint Properties</td>
</tr>
<tr>
<td>Interaction Execution Engine</td>
<td>• Sequence Diagram (see Cameo Simulation Toolkit API UserGuide.pdf in the &lt;md.install.dir&gt;/manual directory for more details)</td>
</tr>
</tbody>
</table>

To create a simulation, either (2.1) execute the elements that are supported by the execution engines or (2.2) create the execution configuration including setting the target element to be executed by the execution configuration and executing the model from the execution configuration.

2.1 Simulation by Executing Elements

Cameo Simulation Toolkit allows you to execute a model through the context menu. Right-click the element that you would like to execute to open the menu.

To execute a model through the context menu:

1. Right-click an element either (i) on a diagram (Figure 1) or (ii) in the containment browser (Figure 2), and then select Simulation > Execute.
Figure 1 -- Executing a Model through the Context Menu on a Diagram
2. The Simulation window will open. The Simulation session will automatically start and be displayed in the Sessions pane. The session corresponds to the selected element of the active diagram.
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 as follows:

- Behaviors
- Class
- Diagram
- Instance Specification

### 2.1.1 Behaviors

You can select a behavior whether it is an Activity or a State Machine, and execute it.

#### 2.1.1.1 Activity

If you choose to execute an Activity behavior (Figure 4), Cameo Simulation Toolkit will execute it on the Activity diagram whose context contains 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 an Activity Behavior
2.1.1.2 State Machine

If you choose to execute a State Machine behavior (Figure 6), it will be executed on the State Machine diagram whose context contains the selected State Machine (Figure 7).
Figure 6 -- Executing a State Machine Behavior
To execute the behavior from an Activity or a State Machine diagram directly, either:

(i) Open the diagram and click the **Execute** button on the **Simulation** window toolbar, or

(ii) Right-click 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 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 an activity or state machine (Figure 8), it will be executed once you have clicked the **Run Execution** button. For example, if you execute the Calculator class (Figure 1), the simulation will be performed on the Calculator state machine (Figure 7).
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 10 -- Animation of a Parametric Execution
2.1.3 Diagram

To execute a diagram:

- Right-click a diagram and select **Simulation > Execute** (Figure 11). The element of the context of the diagram will be executed the same way as behaviors or classes.

![Figure 11 -- Executing an Activity Diagram](image)

2.1.4 Instance Specification

You can also simulate an InstanceSpecification. Both the runtime object and values will be created from the selected InstanceSpecification and its slot values. These runtime object and values will be used for the execution. You can see more information about the runtime object and values in Section 5.4.3.

To execute an InstanceSpecification:

- Right-click an InstanceSpecification and select **Simulation > Execute**. The classifier of the selected InstanceSpecification will be executed the same way as behaviors or classes. 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 an InstanceSpecification is Being Executed

2.2 Simulation by Executing the Execution Configuration

You can create a simulation by executing the Execution Configuration through either the (i) context menu or (ii) Simulation Control toolbar. The Execution Configuration is a class element with the \(<\text{ExecutionConfig}\>\) stereotype applied.

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

- Right-click an Execution Configuration and select Simulation > Execution (Figure 13).
(ii) To execute an Execution Configuration through the Simulation Control toolbar:

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

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

3.1 ExecutionConfig Stereotype

Cameo Simulation Toolkit provides a model-based execution configuration through the «ExecutionConfig» stereotype (Figure 15). The «ExecutionConfig» configuration properties consist of:

- **executionTarget**: The element from which execution should be started.
- **silent**: If the value is true, no animation (or idle time) will be used.
- **UI**: The user interface for configuration mockups to be started with the execution.
- **log**: The element in which the execution trace will be recorded.
- **resultInstance**: An InstanceSpecification whose execution results will be saved as its slot values. If a resultInstance is not specified, the execution results will not be saved even though the executionTarget is the InstanceSpecification.
- **executionEngine**: An execution engine that can be used for model execution ordered by priority. The first engine gains the highest priority. If the execution configuration does not have a tagged value for this tag definition, the values that are defined in the Registered Execution Engine Priority Environment Options will be used.
- **autorun**: If the value is true, execution will start running automatically once it has been initialized. Otherwise, the Run Execution button in the Simulation Console pane must be clicked to run the execution.
- **clock ratio**: The ratio between a real-time clock and a simulation clock. For example, if the clock ratio is 10, it means that one second on the simulation clock is equal to 10 seconds on the real-time clock.
- **autoStartActiveObjects**: If the value is true, the runtime objects whose classifier is active will start their behavior automatically in asynchronous mode. Otherwise, their behavior will start using startObjectBehaviorAction.
- **executionListeners**: A list of execution listeners that will receive events from a model execution. An execution listener can be a SequenceDiagramGeneratorConfig.

You can select and execute an Execution Configuration directly from the «ExecutionConfig>> configuration properties.
3.2 Execution Log

To record all runtime event occurrences into a specific model element, create 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 takes place (Figure 16).

![Figure 16 -- Execution Log](image)

A model-based execution log or trace has many benefits including:

- the source of various customized reports and analysis using the MagicDraw validation mechanism (as both are model-based).
- the capability to import execution data into any other UML compliant tools.

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

- **Signal Instance** (when recordSignals = true) under the “Signal Instances” node: timestamp (that is the relative occurrence time in milliseconds: '0' when the execution starts), signal type, and target (Figure 17).
- **Sequence of Activation** and **Sequence of Deactivation** (when recordActivation = true) under the “Activation Sequence” node: timestamp and types of the element being activated or 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 and the Property and value(s) of a selected Property.
- **Constraint Failure** (when recordedConstraintFailures = true) under the “Constraint Failures” node: timestamp, element, target, and value(s).

### Note

You can use Execution Configurations as the target elements in other execution configurations in the next release of Cameo Simulation Toolkit.
3.3 Simulation Time and Simulation Clock

When you execute a model related to time, for example, a transition with a time trigger, the time that will be used for model execution is obtained from the simulation clock. This time is called simulation time. The ratio between time on the real-time clock and the simulation clock in the same interval is called “clock ratio.” Cameo Simulation Toolkit allows you to set a clock ratio for model execution. To set the clock ratio, specify the value of the clock ratio tag definition of execution config (Figure 18). For further information about the stereotype of execution configuration, see Section 3.1 ExecutionConfig Stereotype.

For example, if the clock ratio is 10, it means that 10 seconds on the real-time clock is equal to one second on the simulation clock. Therefore, the simulation clock is 10 times slower than the real-time clock.

You can adjust the clock ratio during model execution by using the Simulation Clock dialog.
To adjust the clock ratio during model execution:

1. Select a root session of the execution sessions in the Sessions pane and click the Simulation Clock button on the toolbar of the Simulation windows. The Simulation Clock dialog will open (Figure 19).

2. Click the Setting button. The Clock Ratio Setting dialog will open.

3. Type a ratio and click the OK button.

The simulation time will be used in the timestamp of signal instance in the Simulation Log (see Section 3.2 Execution Log) and time series chart (see Section 3.8 Time Series Chart).

### 3.4 Automatic Start Active Objects

An active object is a runtime object, which is typed by an active class (Class element with isActive = true). When the value of the autoStartActiveObjects tag definition is true. The classifier behavior of the active class will start automatically right after the object is instantiated. Figure 20 below shows the Stereo System execution config with autoStartActiveObjects = true. Therefore, the behavior of all active objects will start automatically. In this example, they will be the behavior of the Speaker, Headphone, and Player objects.
If the value of `autoStartActiveObjects` is false or the classes or blocks are not active classes, you have to start each object using the `startObjectBehavior` action. Figure 21 below shows the System init activity, which is the classifier behavior of the stereo system block. This behavior uses the `startObjectBehavior` actions to start the behaviors of the runtime objects of big speakers, small speakers, and dvd player. If the `autoStartActiveObjects` is set to true and the Speaker’s, Player’s, and Headphone’s blocks are active blocks, the System init can be simplified as shown in Figure 22.

Figure 20 -- The Execution Config with autoStartActiveObjects Is True

Figure 21 -- System init Activity to Start the Behavior of Runtime Objects Using startObjectBehavior Actions
3.5 User Interface Prototyping

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

(i) Representative of model elements (Classifiers)
(ii) External Java class file implementing a MockupPanel interface (Figure 23)

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

3.6 UI Modeling Diagram Execution

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

(i) 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, 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.

(ii) Panels
A UI Panel can hold any supported UI components (buttons, labels, sliders, check boxes, text fields, and even panels themselves).

- If a UI Panel resides in a UI Frame, drag a Property of the Classifier that 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, the UI Panel represents the Property.

- If a UI Panel (child) resides in another UI Panel (parent), drag a Property of the Classifier that types 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 the 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 the frame) in an existing UI Frame in another UI Panel. To reuse the existing components, drag the UI Frame model in the Containment Tree to that UI Panel (the «NestedUIConfig» stereotype will be automatically applied if it has not been; and its "config" tag will then be set to the UI Frame).

(iii) Group Boxes

Group boxes have a similar usage as Panels.

(iv) TextFields, Checkboxes, and Sliders

Drag a Property to one of these three UI components to bind the Property to that particular UI component (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, 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.

(v) Labels

Drag a Property to a UI Label to bind the Property to that particular 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, 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.

(vi) Buttons

A UI button can be used to (a) send Signal(s), (b) call Operation(s), or (c) call Behavior(s):

(a) Send 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 clicked, it will send the associated Signal.

(b) 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 clicked, it will call the associated Operation.

(c) Call Behavior(s)
Drag a Behavior, for example, 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 clicked, it will call the associated Behavior.

Figure 24 demonstrates an example of using MagicDraw’s User Interface Modeling Diagram with Cameo Simulation Toolkit.

The steps in this example include as follows:

1. Drag a Classifier to a UI frame.
2. Drag each Signal to each UI button to associate the Signal.
3. Drag any Classifier’s Property to a UI label to be represented.
4. Reference the frame in the “UI” tag of the ExecutionConfig.

See the Calculator.mdzip sample for detailed instructions.
When you drag any GUI elements to a diagram, click **Execute** to run the simulation animation.

---

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

### 3.7 ActiveImage and ImageSwitcher

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

- **Image**: the image that will be used in animation (from browsing the file or dragging the image directly from a web browser).
- **activeElement**: the element that will use the image once it is activated. (Normally, it is the state of a represented classifier.)
- **onClick**: the signal that will be triggered once the displayed image is clicked.

Figure 25 demonstrates an example of how to use ImageSwitcher and ActiveImage (see the Flashing-Light.mdzip sample):

![Diagram of Flashing-Light sample](image)

---

**Figure 25 -- The FlashLight Sample**
In the example, once the **FlashLight** ExecutionConfig is executed, the UI mockup will be displayed (Figure 26). You can then click the power button (the circle one) to start the execution. Once clicked, the light bulb will blink (see the FlashingLight.mdzip sample).

![FlashLight](image)

*Figure 26 -- The FlashLight Sample - Runtime Animation*

### 3.8 Time Series Chart

The Time series chart is a plot of a runtime value with respect to simulation time. You can show the time series chart of any runtime values during model execution.

To display the time series chart:

1. Open the context menu of the **Variables** pane.
2. Right-click the row of the runtime value which needs to be shown on the time series chart and select **Show in time series chart** (see section 5.4.2 Monitoring Runtime Value with Time Series Chart for more information).

The Cameo Simulation Toolkit’s time series chart can also serve as a predefined subtype of UI config. You can use it as a UI mockup of the ExecutionConfig element just like an image switcher.

To use the time series chart:

1. Create a TimeSeriesChart element to represent a classifier.
2. Specify the **value** tag definition and properties whose values will be monitored in the time series chart. These properties must be members of the classifier represented by the time series chart element.

You can customize the time series chart display using its properties (Table 2).
### Table 2 -- Time Series Chart Properties and Function

<table>
<thead>
<tr>
<th>Property</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixedRange</td>
<td>To specify whether the time series chart will automatically adjust the range in Y-axis. If the value is true, the range of Y-axis will be fixed; otherwise the range will be automatically changed.</td>
</tr>
<tr>
<td>gridX</td>
<td>To show or hide the vertical grid line.</td>
</tr>
<tr>
<td>gridY</td>
<td>To show or hide the vertical grid line.</td>
</tr>
<tr>
<td>maxValue</td>
<td>To specify an upper bound value of Y-axis.</td>
</tr>
<tr>
<td>minValue</td>
<td>To specify a lower bound value of Y-axis.</td>
</tr>
<tr>
<td>plotColor</td>
<td>To specify the plot color.</td>
</tr>
<tr>
<td>refreshRate</td>
<td>To specify the time interval to refresh the chart in milliseconds.</td>
</tr>
<tr>
<td>title</td>
<td>To specify the title of the time series chart.</td>
</tr>
</tbody>
</table>

**Figure 27 -- TimeSeriesChart in the MotionAnalysis.mdzip Sample Project**
3.9 Nested UI Configuration

Cameo Simulation Toolkit provides a «NestedUIConfig» stereotype to create a complex UI mockup, which consists of multiple UI configs. This stereotype contains two tag definitions: (i) feature and (ii) config. The feature tag is mandatory. It specifies which Property (part) of the context it will represent. The config tag specifies which existing UI configuration will be used to be displayed as the UI of the system part, which is represented by the Property specified in the feature tag.

You can use a NestedUIConfig stereotype to either (3.9.1) represent a part of an execution context, (this part can nest components that represent the nested properties) or (3.9.2) represent a part, which contains a reference to an existing UI configuration. The NestedUIConfig stereotype can be applied with the UI Panel or UI Group Box. When it is applied and its tag definitions are set, it can be represented as a part of its owner component.

3.9.1 NestedUIConfig Stereotype Representing a Part of Execution Context

A UI Panel or UI Group Box to which the NestedUIConfig stereotype is applied and the feature tag set can represent some parts and nest other components. One of the samples that shows such purpose is the test_nested_UI_panels.mdzip sample, which is located in the samples/simulations folder of the MagicDraw installed directory.

The following System Class Diagram shows the structures of Class System and Class Monitor (Figure 29).
On the same Class Diagram, there are Instance Specifications of Class System and Class Monitor that will be used in the simulation execution (Figure 30).

The Untitled1 User Interface Modeling Diagram displays the UI configuration that will be used in the simulation execution (Figure 31).
Figure 32 displays the UI configuration and the specification of the UI Panel named panel1, which represents the monitor1 Property, as a part of Class System. When the test_nested Panels ExecutionConfig is executed, the UI mockup will be displayed. Figure 33 exhibits the UI Panels and UI Group Boxes that represent the parts (Properties) of the Class System and in-depth nested parts as well.
3.9.2 NestedUIConfig Stereotype Representing a Part Using UI Configuration

You can also use the NestedUIConfig stereotype to show a part using an existing UI configuration. Any UI Panels or UI Group Boxes that will be used for this purpose must be applied with the NestedUIConfig stereotype and must also be set for both the feature and config tags. The benefit of this purpose is that the existing UI configuration can be reused to illustrate any parts of contexts that have the same Class represented. Thus, you do not always have to create a new UI configuration to represent another part that has the same Class represented.

The FlashLight sample will be used to show how to model a part using an existing UI configuration. The following steps shows you how to create a UI mockup that represents the entire system parts, which uses only one UI Frame.

1. Open the FlashLight sample, which is located in the `<md.installed>/samples/simulation directory (FlashingLight.mdzip). The following System Definition Class Diagram shows the definition of the FlashingLight system (Figure 34). By default, when executing and running this sample, you will see the Button and Light are shown in different Frames. Figure 35 demonstrates the runtime user interface mockup that represents the Class Button and Class Light.
2. Right-click the containment tree and select New Diagram > Custom Diagrams > User Interface Modeling Diagram) to create a User Interface Modeling Diagram.

3. Create a UI Frame on the UI Modeling Diagram (Figure 36) by dragging the Class System to the new created UI Frame.
4. Create two UI Panels in the UI Frame.
5. Drag the Light property to one of the UI Panels (Figure 37) to apply the NestedUIConfig stereotype to the UI Panel and set the Light property to the feature tag of that UI Panel.

6. Drag an existing UI configuration named LampBulb to the UI Panel that represents the Light property (Figure 38). This will set the dragged UI configuration to the config tag of that UI Panel. Figure 39 shows the tagged value specification of the UI Panel that represents the Light property.
7. Drag the Button property to another UI Panel (Figure 40).

8. Drag an existing UI configuration named PowerButton to the UI Panel that represents the Button property (Figure 41).
9. Create a new Simulation Configuration element on any Simulation Configuration Diagrams; set the executionTarget tag to Class System; and set the UI tag to the UI Frame that represents the Class System (Figure 42).

10. Execute and run the new created Execution Configuration. The UI mockup will appear as illustrated in Figure 43.
4. Animation

Active elements on 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 on a diagram, click the diagram to clear it.
- If a model is executed in silent mode through the selection of an execution configuration whose silent tag value is set to 'true', the animation will be disabled (see Section 3. Execution Configuration for more information).

4.1 Active, Visited, and Last Visited Elements

Active elements are the elements on which a simulation session is focused (see 5.1 Understanding Simulation Sessions for more information). They can also be considered as the elements that are currently being executed in a simulation session. They will be annotated with red (by default). Once an active element has been executed, it will become a visited element and be annotated with green by default (Figure 44). The last visited element will be annotated with orange (by default).
4.2 Customizing Animation Colors

There are three kinds of annotated elements in model execution: (i) Active, (ii) Visited, and (iii) Breakpoint elements. By default, active elements will be annotated with red, visited elements with green, and breakpoint elements with yellow. Cameo Simulation Toolkit allows you to customize the colors of annotated elements through the Environment Options dialog.

To open the Environment Options dialog:

- Click 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 45).
3. Customize the colors of the active, visited, breakpoint, or last visited elements.
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 the 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 such as an activity execution. If the executed activity contains any callBehaviorAction, a new simulation session will be created to execute each of the callBehaviorAction. The Sessions pane will display all simulation sessions during execution and order them by context elements in the tree node (Figure 46).
CAMEO SIMULATION TOOLKIT
Simulation Debugging

5.2 Simulation Debugger

Cameo Simulation Toolkit allows you to debug a running model using the debugger buttons such as Suspend, Resume, Step into, and Step over. Table 3 explains the functions of all of the debugger buttons.

Table 3 -- Execution and Debugger Buttons

<table>
<thead>
<tr>
<th>Button</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Run Execution]</td>
<td>Run Execution</td>
<td>To run a selected simulation session.</td>
</tr>
<tr>
<td>![Suspend]</td>
<td>Suspend</td>
<td>To pause the execution of a selected simulation session in the Sessions pane.</td>
</tr>
<tr>
<td>![Resume]</td>
<td>Resume</td>
<td>To resume a suspended simulation session.</td>
</tr>
<tr>
<td>![Step into]</td>
<td>Step into</td>
<td>To execute and run animation in the currently active element of a selected simulation session in the Sessions pane.</td>
</tr>
<tr>
<td>![Step over]</td>
<td>Step over</td>
<td>To execute the currently active element of a selected simulation session and run animation in the background.</td>
</tr>
<tr>
<td>![Terminate]</td>
<td>Terminate</td>
<td>To terminate a selected session in the Sessions pane. If the selected session contains sub-sessions, all of the sub-sessions will also be terminated.</td>
</tr>
</tbody>
</table>

You can examine and edit variables in the Variables pane (see Section 5.4.1 Variables Pane), pause the execution of a model at predefined breakpoints (see Section 5.5 Breakpoints), or execute one element at a time using the Step into or Step over button.

The Debugger pane includes a player-like control panel for a step-by-step execution (see Table 3 above), threads or behaviors with an expandable stack trace (see 5.1 Understanding Simulation Sessions), input/output...
console for custom commands or expressions evaluation (5.3 Simulation Console), Variables Pane/runtime structure (5.4 Runtime Values Monitoring), and the Breakpoints pane (5.5 Breakpoints).

5.3 Simulation Console

5.3.1 Console Pane

Cameo Simulation Toolkit provides Simulation Console to display simulation information during model execution (Figure 47). The information may contain a hyperlink to the model element in a MagicDraw project. The model element will be accessed from the containment browser when you click the hyperlink.

![Simulation Console](image)

*Figure 47 -- Simulation Console with Simulation Information during Parametric Execution*
5.3.2 Simulation Information

There are six levels of information that can be displayed in the Console pane (sorted in ascending order by priority):

- **TRACE**: displays all levels of information.
- **DEBUG**: displays debugging information.
- **INFO**: displays normal information.
- **WARN**: displays warnings.
- **ERR**: displays errors.
- **FATAL**: displays severe errors.

By default, only information with a priority equivalent to INFO or higher (WARN, ERR, and FATAL) will be displayed in the Console pane. You can customize the way information is displayed by editing the `simulation.properties` file in the data directory in the MagicDraw installation directory. You can use a text editor to edit this file.

To change the priority level, for example, open `log4j.category.SIM_CONSOLE`.

```
log4j.category.SIM_CONSOLE=INFO,SimConsoleApp,SimXMLApp
```
Change the first parameter’s priority level from **INFO** (default value) to **TRACE** to display all levels of simulation information in the **Console** pane.

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

You can see more information about customizing the information displayed in the Console pane from the comment in the **simulation.properties** file.

### 5.3.3 Simulation Log File

During model execution, the simulation information will be displayed in the **Console** pane. However, the **Console** pane is limited to display only 60,000 characters owing to the performance constraints. The characters that exceed the maximum character limit will not be displayed. Nevertheless, your old simulation information will be automatically archived in the **simulation.log** file in the user home directory (`<User home directory>/magicdraw/<version>`). The **simulation.log** file is an XML file (or a text file) that records all simulation information that has ever been displayed in the **Console** pane during model execution (see the comment in the **simulation.properties** file to customize the file).

### 5.4 Runtime Values Monitoring

#### 5.4.1 Variables Pane

You can select a session in the **Sessions** pane to display the runtime objects and values that correspond to the context element of a selected session in the **Variables** pane (Figure 49).

![Figure 49 -- The Variables and Sessions Panes of a Simulation Session](image)

When a model is being executed, the (5.4.1.1) 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 49, 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 49, the runtime object of the simulation session context is the “Calculator@155d21b” instance. Since the runtime instance is the “Calculator” Class type, it can contain structural features (which correspond to the Class attributes), such as “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, such as “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 49) 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 the Name column. A runtime value can be the input or the 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 5 -- Buttons and Functions in the Variables Pane Toolbar

<table>
<thead>
<tr>
<th>Button</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Refresh</td>
<td>To refresh the tree and values in the Variables pane.</td>
</tr>
<tr>
<td></td>
<td>Export to New Instance</td>
<td>To create a new InstanceSpecification and export a selected runtime object to a newlycreated InstanceSpecification.</td>
</tr>
<tr>
<td></td>
<td>Export to Instance</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 5 -- Buttons and Functions in the Variables Pane Toolbar

5.4.2 Monitoring Runtime Value with Time Series Chart

Cameo Simulation Toolkit allows you to show the plot between runtime values, which are the numerical value and simulation time. This plot is called Time Series Chart. To view this chart during model execution, right-click...
the row of a runtime value in the Variables pane and select Show in time series chart (Figure 50).

![Figure 50 -- The Context menu Showing the Runtime Value in Time Series Chart](image)

The time series chart shows the runtime value with respect to simulation time as shown in Figure 51.

![Figure 51 -- The Time Series Chart of Runtime Value of Property x](image)
5.4.3 Runtime Object Created from InstanceSpecification

At the starting point of model execution, a runtime object will be created to store runtime values. If the element to be executed 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 that are equivalent to the defined feature of the slots.

If the slot of the InstanceSpecification is empty, and the defined feature of the slots has defined a default value, the runtime value will be created from the default value and will be assigned to the runtime object’s structural feature instead. Figure 52 shows the runtime object that is created for executing the 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 the radius and thickness property of the Pipe class (0.05 and 0.002 respectively).

![Figure 52 -- Variables Pane Showing the Runtime Object](image)

5.4.4 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 the runtime object will be set to the slots of the 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 the row and select Export to New Instance (Figure 53).
2. The Select Owner dialog will open. Select the owner of the created InstanceSpecification (the system folder) and click OK (Figure 54).

(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 the row and select Export to Instance (Figure 55).
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 56).

![Select Instance Dialog](image)

**Figure 56 -- 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 57).

![Breakpoints Pane](image)

**Figure 57 -- An Example of a Breakpoints Pane**
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 a symbol of the model element on a diagram, and then select Simulation > Add Breakpoint(s) (Figure 58).

---

**Table 6 -- Breakpoints Pane Columns**

<table>
<thead>
<tr>
<th>Column</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>To display the enabled or disabled state of a breakpoint. If the value is true, 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).</td>
</tr>
<tr>
<td>Element</td>
<td>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).</td>
</tr>
<tr>
<td>Condition</td>
<td>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 or breakpoint when the result of the boolean expression is true. If the condition is not defined, the execution will always be suspended when it reaches that particular breakpoint.</td>
</tr>
<tr>
<td>Suspend</td>
<td>There are three kinds of execution suspensions: (i) <strong>On Entry</strong>, (ii) <strong>On Exit</strong>, and (iii) <strong>Both</strong>.</td>
</tr>
</tbody>
</table>

(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.
5.5.2 Removing Breakpoints

You can remove a Breakpoint 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 59).
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 60).
5.6 Disabling Updates in Simulation Panes

You can click the toggle button in the Simulation window to disable automatic updates of all the panes in the Simulation window (Figure 61). If updates of all the panes are disabled, the execution speed will increase.

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 62).

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 63).
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 stage 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’s statecharts. SCXML is capable of describing complex state machines, including sub-states, concurrency, history, time events, and many more. Most of the things that can be represented as UML statecharts such as business process flows, views on 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 as a demonstration tool to validate and verify the system behavior at key milestone reviews. In addition, Cameo Simulation Toolkit supports exporting the UML state machine to standard SCXML files for further analysis or transformations (through the state machine context menu).

7.1 Supported Elements

Most of the supported elements on a State Machine diagram are shown in Table 7.
Table 7 -- Supported Elements on the State Machine Diagram

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Executable (Yes/No)</th>
<th>Exportable to SCXML (Yes/No)</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>composite state</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>orthogonal state</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>submachine state</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>initial state</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>final state</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>onEntry</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>onExit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>onTransition</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>doActivity</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>time event</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>deep history</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>shallow history</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>transition-to-self</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>choice</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note: All elements on 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 are specified in Table 7. 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 a time event.
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 (Figure 65).

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 the example 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).
7.2.3 Behaviors on Entry, Exit, and Do Activity of State

States can have defined behaviors at Entry, Exit, or Do Activity (Figure 66). 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 66 -- The Execution of Behavior on the 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 the state machine execution, the state of a runtime object will be changed by a trigger. The trigger can be either a signal or a 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 a runtime object, which is the target for the signal, in the **Variables** pane. All signals that can be received by the selected runtime object will be listed on the **Triggers** drop-down menu on the **Simulation** window toolbar (Figure 67).

![Figure 67 -- The Triggers Drop-down Menu](image)

A signal can be sent to a runtime object through a User Interface mockup. See more information about UI mockup in Section 3.6 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 The test_regions.mdzip Sample**
- **7.4.2 The test_timers.mdzip Sample**
- **7.4.3 The test_guard.mdzip Sample**

#### 7.4.1 The test_regions.mdzip Sample

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 the **Console** pane contains all of the outgoing transitions triggers of all active states.
- If one of the parent state’s outgoing transitions is triggered, an exit activity will be executed before the state is deactivated.
7.4.2 The test_timers.mdzip Sample

This sample demonstrates the implementation of timing events on a State Machine diagram.

- Transitions with the 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 The test_guard.mdzip Sample

This sample demonstrates the ability to specify and resolve the guard conditions on 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.
- In this case, they come from the slots of the instance of the context classifier (see the Instance diagram).

8. Activity Simulation

8.1 Activity Execution Engine

Cameo Simulation Toolkit provides an Activity Execution Engine that allows you to perform an Activity Simulation (Execution) on Activity Diagrams or Activity Elements. Cameo Simulation Toolkit also 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 supports specific abstractions 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 12.4 Using MATLAB® as Mathematical Solver for more details)
- CallBehaviorAction with nested diagrams execution and animation
- SendSignalAction can be used to send a signal to a global event queue to be consumed by any other engines (such as state machine)
<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 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.</td>
</tr>
<tr>
<td>• 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. To change this mode to a regular UML (Boolean expression), click <strong>Options &gt; Environment</strong> on the main menu, and then select the <strong>Simulation</strong> node on the left-hand side of the <strong>Environment Options</strong> dialog. Next, select the <strong>Use fUML Decision Semantics value</strong> check box so that the value becomes false. The value is false by default in UML mode.</td>
</tr>
</tbody>
</table>
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 demonstrates how to create a simple, executable Activity model through 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 the 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. To create a new UML project, click File > New Project... on the main menu. The New Project dialog will open (Figure 68).

![Image of the New Project dialog]

**Figure 68 -- The New Project Dialog**
2. Select UML Project from the General-Purpose Modeling group and specify the project’s name, such as “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 69). Right-click x and select Specification to open its Specification dialog. Select Integer as the property type (Figure 70).

![Figure 69 -- Creating a New Property ‘x’ for the SumPrinter Class]
(ii) Repeat Step (i) to create property $y$ (Figure 71).

Figure 70 -- Selecting Property Type

6. Once the properties $x$ and $y$ have been created, define the behavior of the created class: Specify 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”.

Figure 71 -- **SumPrinter** Class with Properties $X$ and $Y$ of **Integer** Type
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 72 and Figure 73).

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 from the drop-down list on the right-hand side.
(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 74).
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` in the name field and select Integer as the type of parameter a (Figure 75).

Figure 74 -- PrintSumOfIntegers Opaque Behavior in the Containment Browser
(ii) Repeat Step (i) to create parameter b (Figure 76).
(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, such as 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 77).
Figure 77 -- JavaScript for Printing the Summation of Integer Values
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 the containment browser to the **PrintSum** activity diagram. A new action of **PrintSumOfIntegers** will be created.
2. Name the action “print” (Figure 78).

---

**Figure 78 -- Dragging the PrintSumOfIntegers Opaque Behavior to the Activity Diagram To Create a Print Action**
3. Add the Initial and Activity Final nodes to the activity diagram and connect them to the print action using a control flow (Figure 79).

4. Click **Action** and select the **Any Action...** button from the **Activity Diagram** toolbar on the **PrintSum** activity diagram (Figure 80).

5. Select **ReadStructuralFeatureAction** in the **Select Action Metaclass** dialog and click **OK** (Figure 81).
6. Click the **PrintSum** activity diagram to create the action and name it “readX” (Figure 82).

7. Open the **Specification** dialog of the **readX** action (Figure 83).
8. Click the **Structural Feature** and the “...” button to open the **Select Property** dialog to select the structural feature (Figure 84).
Select, search for, or create an element
Search for an element by using list or tree views. To find an element type text or wildcard (",",?) into the "Search by name" input field. Search elements by their qualified names or use camel case when searching if the appropriate mode is enabled.

9. Select the property $x$ of the `SumPrinter` class and click OK. The Select Property dialog will close.
10. Select **Pins** on the left-hand side pane of the **Specification** dialog. You need to create two pins for **ReadStructuralFeatureAction**: (10.1) The input pin to specify the runtime object of type **SumPrinter** whose runtime values correspond to the properties \(x\) and \(y\) used for execution and (10.2) the output pin of the type **Integer** to specify the value read from the structural feature. At this procedure, there are two steps to be followed:

10.1 Click the **Object** button and select **Input Pin** from the context menu (Figure 85). The **Input Pin** dialog will appear with a new input pin to be added to the action. Name this pin "self" and click the **Type** row. Select **SumPrinter** as its type from the drop-down menu, and then click the **Back** button (Figure 86).
10.2 Click the **Result** button and select **Output Pin** from the context menu (Figure 87). Name this pin “a” and select **Integer** as its type, and then click the **Close** button (Figure 88).

---

**Figure 86 -- Naming the Input Pin and Selecting Its Type**

**Figure 87 -- Adding an Output Pin to readX Action**
11. Click the `readX` action on the activity diagram and select **Display Pins** (the last icon) on the smart manipulator (Figure 89). The **Select Pins** dialog will open (Figure 90).
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 with Object Flow on the Smart Manipulator (Figure 91).

14. Repeat steps 4 to 13 to create a **readY** action, which is the **ReadStructuralFeatureAction**, with the following arrangements:
   - The name of the action is “**readY**”.
   - The structural feature is ‘**y**’ attribute of the **SumPrinter** class.
   - The name of the output pin of **readY** is ‘**b**’.
   - The output pin **b** of **readY** connects to pin **b** of the **print** action.
(vii) To create a ReadSelfAction to read a runtime object that will be supplied to the input pins of `readX` and `readY` actions:

1. Click **Action > Any Action...** on the Activity Diagram toolbar. The **Select Action Metaclass** dialog will open (Figure 93).

2. Select **ReadSelfAction** and click **OK**.

3. Click the **PrintSum** activity diagram to create an action and name it "readSelf" (Figure 94).
4. Right-click the readSelf action to open its Specification dialog (Figure 95) through the context menu.

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. Click the Back button and the Close button.

6. Go to the PrintSum activity diagram, click the readSelf action and select Display Pins on the smart manipulator to show the output pin of the readSelf action.

7. Create a Fork Horizontal and use Object Flow to connect it to the pins of the readX, readY, and readSelf actions on the diagrams (Figure 96).
The final step will be to create an **InstanceSpecification** whose classifier is the **SumPrinter** and assign the values to the slots that correspond to the 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 97).
3. Right-click the created **Instance** and open the **Specification** dialog of **instance**.
4. Click the **Classifier** field then the “...” button. The **Select Elements** dialog will open (Figure 98).
5. Select the SumPrinter class to edit the classifier and click OK.
6. Click Slots on the left-hand side pane of the Specification dialog and select $x$:Integer (Figure 99).
7. Click the Create Value button to create a new value of the slot (Figure 99). The Value box will open (Figure 100).
Figure 99 -- Creating Slot Value of \( x \)
8. Type in a number, for example, 2 as the value of the property $x$ slot.
9. Repeat steps 6 to 8 to assign “8” as the value of the property $y$ slot (Figure 101), and then click Close.

The model is now ready to be executed.
8.3 Executing Activity

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

The following example shows 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 102). The breakpoints will be shown in the Breakpoints pane of the Simulation window (Figure 103).
Figure 103 -- The Breakpoints Pane in the Simulation Window

To open the Simulation window, click Window > Simulation on the main menu (Figure 104).
2. Right-click instance in the containment browser and select Simulation > Execute (Figure 105) to execute the model from instance, which is the InstanceSpecification of the SumPrinter classifier.
Figure 105 -- Executing the InstanceSpecification of the SumPrinter Classifier

3. A new simulation session will be created and displayed in the **Sessions** pane of the **Simulation** window (Figure 106). The symbol of the elements that have breakpoints attached will be highlighted in yellow by default (Figure 107).
Figure 106 -- A Simulation Session in the Sessions Pane of the Simulation Window

4. Click the **Run Execution** button on the **Simulation** window toolbar to 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 107 -- 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 108).
6. The execution will be suspended again when pin \( b \) is activated. Click the **Resume Execution** button to continue the execution. In the **Console** pane of the **Simulation** window, you can see the printed value of 10, which is the summation between 2 and 8 (Figure 109).

![Figure 108 -- The Resume Execution Button in the Simulation Window](image)

6. The execution will be suspended again when pin \( b \) is activated. Click the **Resume Execution** button to continue the execution. In the **Console** pane of the **Simulation** window, you can see the printed value of 10, which is the summation between 2 and 8 (Figure 109).

![Figure 109 -- The Console Pane of the Simulation Window Showing the Printed Result of the Summation](image)

**Note**

If you do not want to display the animation (silent execution), you can create **Execution Configuration** to customize the execution, select **instance** as the **executionTarget**, and set **silent** to **true**. See Sections 2.2 Simulation by Executing the Execution Configuration and 3. Execution Configuration 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 own attributes or nested properties.

The Parametrics Engine will use the Mathematical Engine to solve the mathematical and logical expressions that are defined as the constraints of Constraint Blocks. CST uses the Built-in Math Engine by default. Optionally, MATLAB can be selected as the active Math Engine via Environment Options.
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 will be considered 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.mdlzip sample (Figure 110), 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.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The SysML profile is required for a parametric execution.</td>
</tr>
<tr>
<td>• A parametrics simulation evaluates the expressions in one direction, which specifies inputs to get outputs. For example, for the expression ( z = x + y ), the values of ( x ) and ( y ) must be given to evaluate ( z ).</td>
</tr>
<tr>
<td>• 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.</td>
</tr>
</tbody>
</table>
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 a parametric execution. However, they must be inherited from the basic SysML value types which are Boolean, Integer, Real (Figure 111), and Complex. You can see the example in the CylinderPipe.mdzip sample.
9.2.3 Using Binding Connectors

SysML provides a binding connector to connect elements whose values are bound together (Figure 113). 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 the 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 an operation, and then insert a 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 (Figure 114). 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.

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. To use the InstanceSpecification of the SysML block, you have to define the resultInstance of the «ExecutionConfig» stereotype in the InstanceSpecification (Figure 115).
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 (Figure 116).

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: \( z = x \times y \). If four values are passed on to the mathematical engine for each parameter \( x \) or \( y \), then two column matrices (4x1 matrices) will be constructed 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. To rewrite the expression to solve the problem, you need to change the expression to: \( z = \text{diag}(\text{diag}(x) \times \text{diag}(y)) \).

Note: You need to create InstanceValues for the slots that 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.
9.2.5.2 Constructing Values List from Complex Aggregation Structure

If you have an InstanceSpecification of a 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, you need to pass all of these values on to the mathematical engine.

If this is the case, the Parametrics 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`. On the SysML Parametric diagram in the CylinderPipe.mdzip sample (Figure 116), and the InstanceSpecification of the SysML block "RawMaterialSupplier" shown in Figure 117, 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
- thickness = 0.002, 0.002, 0.005

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 Parametrics Simulation Engine (by selecting Block A, InstanceSpecification a:A, or the Execution Configuration class symbol on the Execution Configurations Block Definition Diagram), and then either:
   - (i) right-click the element symbol and select Simulation > Execute on the context menu or
(ii) select the element symbol and click the **Execute** button on the **Simulation** window toolbar. (If you click the **Execute** button without selecting any elements, 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, it will automatically be used for the simulation. Otherwise, the **Select Element** dialog will open for you to select the InstanceSpecification (Figure 118).

![Select Element dialog](image)

**Figure 118 -- Selecting an InstanceSpecification in the Select Element Dialog**

2. Once the Parametric Simulation Engine has started, the runtime structure of the executed classifier will be shown in the **Variables** pane (Figure 119). You can modify the values in the **Value** column of the **Variables** pane.
3. Click the Run button to start the simulation. The Parametric Simulation Engine will simulate your Parametrics model (with animation on your diagram), and input the calculation result into the corresponding slot of the selected InstanceSpecification automatically (Figure 120).
9.4 Retrieving Simulated Values

You can save 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 120 -- Simulation with the Parametric Simulation Engine

Figure 121 -- 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 a 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 to execute the parametric simulation, for example:

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

An argument object is the runtime object of a classifier to be executed. To obtain this particular runtime object, you can use some UML actions such as ReadSelfAction, ReadStructuralFeatureValueAction, ValueSpecificationAction, or the Cameo Simulation Toolkit Open API. Figure 122 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.

![Parametric Activity Diagram in the CylinderPipe.mdzip Sample](image)

9.6 Sample Projects

The Parametric Simulation sample projects are available in the `<md.install.dir>/samples/simulation/Parametrics` directory. The SysML Parametric diagrams and InstanceSpecifications, which can be used for the sim-
ulation as described in Step 1 of Section 9.2.5 Working with Multiple Values, 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 (see Section 9.2.5). 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: Demonstrates 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. Here are some examples:

- A SendSignalAction can send a signal to a trigger transition on an active State Machine (using Activity Simulation to control State Machine Simulation). The Stopwatch sample gives the example of such collaboration.
- Activating a State can invoke an entry/do/exit Activity. The testDoActivity.mdzip and test_regions.mdzip samples gives the examples of such collaboration.

The Stopwatch sample will be further explained in details in the following section.

10.1 Stopwatch Sample

To execute and control the Stopwatch (see the StopWatch.mdzip sample), located in the `<md.install.dir>/samples/simulation/StopWatch` directory, you can either (10.1.1) manually handle the Stopwatch or (10.1.2) use an 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 ExectionConfig (in the Config Simulation Configuration Diagram) and select Simulation > Execute on the context menu.
3. Click the Run Execution button in the Simulation window to start the execution. The Stopwatch mockup pane will open and the StopWatch will be at the “ready” state.
4. Either (i) click the start button in the mockup pane, or (ii) select the context (StopWatch [ready]) in the Variables pane, and then select the start signal from the Triggers: combo box in the Simulation window to initiate the timer. You can use the following buttons or signals in different states:
• The stop button or signal will stop the timer when the current state is either running or paused.
• The reset button or signal will reset the timer to zero if the current state is stopped.
• The split button or 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 in proper working order. |

10.1.2 Controlling Execution Using Activity Diagram

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

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

1. Open StopWatch.mdzip.
2. Right-click the Stopwatch Testcase ExectionConfig (in the scenario Simulation Configuration Diagram) and select Simulation > Execute on the context menu.
3. Click the Run Execution button in the Simulation window to start the execution. 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 pane will open, and the state machine will start the simulation. 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. Recording Signals to Sequence Diagram

This section will use the StereoSystem.mdzip sample, located in the <md.install.dir>/samples/simulation/ directory, to demonstrate how to record signals to a sequence diagram.

During model execution, Cameo Simulation Toolkit can record all of the signals sent from a selected runtime object to a sequence diagram.

To record signals sent from a runtime object to a sequence diagram:

1. In the Variables pane, right-click the runtime object from which the signals to the sequence diagram will be sent and recorded.
2. Click Create Sequence Diagram from the context menu (Figure 123). An empty sequence diagram will be created.
Once you have created the Sequence diagram, the signals and lifelines will be recorded into the diagram whenever you execute a model (Figure 124).

You can also customize the recorded signals and lifelines using a SequenceDiagramGeneratorConfig.
A SequenceDiagramGeneratorConfig is a stereotype that is inherited from an ExecutionListener stereotype. It contains three tag definitions as follows:

(i) owner: the element, which is the owner of the generated sequence element. The generated diagram will be created in the generated sequence element.

(ii) ignoredSignals: a list of signals that will not be recorded to the generated sequence diagram.

(iii) ignoredLifeline: a list of elements that will not be recorded as lifelines to the generated sequence diagram.

To customize a generated sequence diagram:

1. Create a class element and apply a SequenceDiagramGeneratorConfig stereotype to it.
2. Open the Specification dialog of the created class and specify the value(s) of the tag definition of the SequenceDiagramGeneratorConfig stereotype.
3. Add the created class to the values of the executionListeners of ExecutionConfig element (see 3.1 ExecutionConfig Stereotype) and execute the model using this ExecutionConfig.

12. 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.

12.1 Math Console

Math Console in Cameo Simulation Toolkit is used to communicate with the mathematical engine. Cameo Simulation Toolkit is designed to work with various mathematical engines such as MATLAB® and OpenModelica. You can create a new mathematical engine as a MagicDraw plugin and register it to Cameo Simulation Toolkit.

The current release of Cameo Simulation Toolkit 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.

---

1. MATLAB® is a registered trademark of The MathWorks, Inc.
2. Currently not supported.
2. Select **Simulation** on the left-hand side pane and select a mathematical engine from the **Mathematical Engine** field (Figure 126).

![Figure 126 -- Selecting a Mathematical Engine in the Environment Options Dialog](image)

You can see the mathematical engine you have selected from the **Environment Options** dialog in the **Simulation** window. To start or stop the mathematical engine, click the **Start Math Engine** or **Stop Math Engine** button (Figure 127).
12.2 Exchanging Values between Cameo Simulation Toolkit and Mathematical Engine

12.2.1 Exchanging Values between Slot and Mathematical Environment

Cameo Simulation Toolkit allows you to exchange values between a slot and a Mathematical engine through the diagram context menu on the slot (Figure 128).

Table 8 -- Math Console Buttons and Their Functions

<table>
<thead>
<tr>
<th>Button</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start Math Engine</td>
<td>To start the mathematical engine.</td>
</tr>
<tr>
<td></td>
<td>Stop Math Engine</td>
<td>To stop the mathematical engine.</td>
</tr>
<tr>
<td></td>
<td>Clear</td>
<td>To clear all text displayed in Math Console.</td>
</tr>
</tbody>
</table>
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 open (Figure 129).

2. Specify a variable name that contains the value you want to import and click **OK**.
To export a value from a slot to a Mathematical engine:

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

### 12.2.2 Exporting Runtime Value to the Mathematical Engine

During model execution, you can export the runtime values in the **Variables** pane to the mathematical engine using the context menu of the selected value (Figure 130). This function allows you to analyze the exported runtime values using the Mathematical engine, for example, plot.

![Variables pane](image)

**Figure 130 -- The Context Menu to Export the Runtime Value to the Mathematical Engine**

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

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

### 12.3 Built-in Math Solver

The 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 the 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

#### 12.3.1 Using Math Solver in Math Console

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

\[
\begin{align*}
    x &= 10; \\
    y &= 20; \\
    z &= x + y
\end{align*}
\]
and \( z = 30 \) (the calculation result) will be displayed.

| Note | The calculation results for expressions ended with a semicolon (;) will be set to the corresponding variables in the Math Solver environment. It will not be displayed in Math Console. |

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

\[
a = \text{true}; \\
b = \text{false}; \\
c = a \& b;
\]

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

If an expression does not contain any assignment operators, the calculation result will be set to the variable \'ans\'. For example:

\[
x = 10; \\
20 + x
\]

\( \text{ans} = 30 \) will be displayed in 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 Math Console.

### 12.3.2 Variables

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

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

### 12.3.3 Values

The valid values that can be used in an expression are: (12.3.3.1) Real Number, (12.3.3.2) Complex Number, (12.3.3.3) Boolean, and (12.3.3.4) Matrix.

#### 12.3.3.1 Real Number

\[
x = 3.14159 \\
y = 2
\]

#### 12.3.3.2 Complex Number

\[
c = 3 + 4i \\
d = 1.25 + 0.25i
\]
12.3.3.3 Boolean

\[ a = \text{true} \]
\[ b = \text{false} \]

12.3.3.4 Matrix

\[ U = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix} \]
\[ A = \begin{bmatrix} \text{true} \\ \text{false} \\ \text{false} \\ \text{true} \end{bmatrix} \]

You can add a matrix to the built-in Math Solver by using the following syntax (a semicolon is used as a row separator and comma or space is used as a comma separator), for example:

\[ U = \begin{bmatrix} 1.0 & 2.0 & 3.0; 4.0, 5.0, 6.0; 7.0, 8.0, 9.0 \end{bmatrix} \]
\[ A = \begin{bmatrix} \text{true}; \text{false}; \text{false}; \text{true} \end{bmatrix} \]

You can refer to a matrix element with the row and column index specified in round brackets after a matrix name, for example (see \( U \) above):

\[ U(1, 1) \text{ is } 1.0 \]
\[ U(2, 3) \text{ is } 6.0 \]

You can also refer to a matrix element with only one index specified in round brackets after a matrix name. In this 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 (see \( U \) above):

\[ U(2) \text{ is } 4.0 \]
\[ U(6) \text{ is } 8.0 \]

### NOTE

An ‘i’ character in an expression can be parsed as either an imaginary unit or a character of a variable name. If the character ‘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, it will be parsed as a variable, for example:

- \( \text{ca} = 1i \)  ‘i’ is parsed as an imaginary unit.
- \( \text{cb} = i \)  ‘i’ is parsed as a variable.
- \( \text{cx} = 3.25i \)  ‘i’ is parsed as an imaginary unit.
- \( \text{cy} = 4i4 \)  ‘i’ is parsed as the first character of a variable name ‘i4’
12.3.4 Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>A real value that is closer than any other to e, the base of natural logarithms.</td>
</tr>
<tr>
<td>PI</td>
<td>A real value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.</td>
</tr>
</tbody>
</table>

12.3.5 Operators

### NOTE
- \( x \) 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.

#### 12.3.5.1 Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>( x+y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( U + V ) (( U ) and ( V ) are ( m \times n ) matrices)</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>( x-y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( U + V ) (( U ) and ( V ) are ( m \times n ) matrices)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>( x^y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( U^*V ) (( U ) is an ( m \times n ) matrix and ( V ) is an ( n \times p ) matrix)</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>( x/y )</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>( m%n )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( U + V ) (( U ) and ( V ) are ( m \times n ) matrices of integer values)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This operator operates element-wise on matrices.</td>
</tr>
<tr>
<td>!</td>
<td>Factorial</td>
<td>( m! )</td>
</tr>
<tr>
<td>^</td>
<td>Power</td>
<td>( x^y )</td>
</tr>
</tbody>
</table>

#### 12.3.5.2 Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
<td>( x=y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( a=b )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( U=V )</td>
</tr>
</tbody>
</table>
### 12.3.5.3 Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater</td>
<td>x&gt;y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U&gt;V</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less</td>
<td>x&lt;y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U&lt;V</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater or Equal</td>
<td>x&gt;=y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U&gt;=V</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less of Equal</td>
<td>x&lt;=y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U&lt;=V</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
<td>x==y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a==b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U==V</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
<td>x!=y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a!=b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U!=V</td>
</tr>
</tbody>
</table>

**NOTE** All comparison operators operate element-wise on matrices, for example:

\[
A = [1; 2; 3]; \\
B = [3; 2; 1]; \\
A>B \text{ is } [false \ false \ true];
\]

### 12.3.5.4 Boolean Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>NOT</td>
<td>!a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>!A</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>a&amp;b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A&amp;B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>^</td>
<td>XOR (exclusive OR)</td>
<td>a^b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A^B</td>
</tr>
</tbody>
</table>

**NOTE** All boolean operators operate element-wise on matrices, for example:

\[
A = [true; true; false; false]; \\
B = [true; false; true; false]; \\
A&B \text{ is } [true; false; false; false];
\]
### 12.3.6 Functions

**NOTE**
- \(x\) 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 a matrix of values.

A matrix can be passed to the function that operates element-wise on matrices, as its argument, for example:

\[
X = \begin{bmatrix}
1 & -2 & 3 \\
-4 & 5 & -6 \\
7 & -8 & 9
\end{bmatrix};
\]

\[
Y = \text{abs}(X)
\]

result:

\[
Y = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>\text{abs}(x)</td>
<td>To return an absolute value of (x) or a complex modulus of (c).</td>
</tr>
<tr>
<td></td>
<td>\text{abs}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acos</td>
<td>\text{acos}(x)</td>
<td>To return an arc cosine of an angle in the range of 0.0 through (\pi).</td>
</tr>
<tr>
<td></td>
<td>\text{acos}(c)</td>
<td>All angles are in radians.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acosd</td>
<td>\text{acosd}(x)</td>
<td>To return an inverse cosine of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>\text{acosd}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acosh</td>
<td>\text{acosh}(x)</td>
<td>To return an inverse hyperbolic cosine of a given value.</td>
</tr>
<tr>
<td></td>
<td>\text{acosh}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acot</td>
<td>\text{acot}(x)</td>
<td>To return an inverse cotangent of a given value.</td>
</tr>
<tr>
<td></td>
<td>\text{acot}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acotd</td>
<td>\text{acotd}(x)</td>
<td>To return an inverse cotangent of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>\text{acotd}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acoth</td>
<td>\text{acoth}(x)</td>
<td>To return an inverse hyperbolic cotangent of a given value.</td>
</tr>
<tr>
<td></td>
<td>\text{acoth}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acsc</td>
<td>\text{acs}(x)</td>
<td>To return an inverse cosecant of a given value.</td>
</tr>
<tr>
<td></td>
<td>\text{acs}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acscd</td>
<td>\text{acscd}(x)</td>
<td>To return an inverse cosecant of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>\text{acscd}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acsch</td>
<td>\text{acsch}(x)</td>
<td>To return an inverse hyperbolic cosecant of a given value.</td>
</tr>
<tr>
<td></td>
<td>\text{acsch}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asec</td>
<td>\text{asec}(x)</td>
<td>To return an inverse secant of a given value.</td>
</tr>
<tr>
<td></td>
<td>\text{asec}(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Syntax</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>asecd</td>
<td>asecd(x) asecd(c)</td>
<td>To return an inverse secant of a given value expressed in degrees. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asech</td>
<td>asech(x) asech(c)</td>
<td>To return an inverse hyperbolic secant of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asin</td>
<td>asin(x) asin(c)</td>
<td>To return an arc sine of an angle in the range of -pi/2 through pi/2. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asind</td>
<td>asind(x) asind(c)</td>
<td>To return an inverse sine of a given value expressed in degrees. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asinh</td>
<td>asinh(x) asinh(c)</td>
<td>To return an inverse hyperbolic sine of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>atan</td>
<td>atan(x) atan(c)</td>
<td>To return an arc tangent of an angle in the range of -pi/2 through pi/2. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>atan2</td>
<td>atan2(x, y) atan2(U, V)</td>
<td>To return an arc tangent of an angle in the range of -pi through pi. atan2(U, V) returns a matrix of the same size as the U and V matrices containing the element-by-element, inverse tangent of the real parts of U and V.</td>
</tr>
<tr>
<td>atand</td>
<td>atand(x) atand(c)</td>
<td>To return an inverse tangent of a given value, expressed in degrees. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>atanh</td>
<td>atanh(x) atanh(c)</td>
<td>To return an inverse hyperbolic tangent of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>ceil</td>
<td>ceil(x)</td>
<td>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. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>conj</td>
<td>conj(c)</td>
<td>To return a conjugated value of c. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>cos</td>
<td>cos(x) cos(c)</td>
<td>To return a trigonometric cosine of an angle. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>cosd</td>
<td>cosd(x) cosd(c)</td>
<td>To return a cosine of a given value expressed in degree. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>cosh</td>
<td>cosh(x) cosh(c)</td>
<td>To return a hyperbolic cosine of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>cot</td>
<td>cot(x) cot(c)</td>
<td>To return a cotangent of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>cotd</td>
<td>cotd(x) cotd(c)</td>
<td>To return a cotangent of a given value expressed in degrees. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>coth</td>
<td>coth(x) coth(c)</td>
<td>To return a hyperbolic cotangent of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Syntax</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>count</td>
<td>count(U)</td>
<td>To return a number of elements of a given matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>csc</td>
<td>csc(x)</td>
<td>To return a cosecant of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>csc(c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cscd</td>
<td>cscd(x)</td>
<td>To return a cosecant of a given value expressed in degree. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>cscd(c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>csch</td>
<td>csch(x)</td>
<td>To return a hyperbolic cosecant of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>csch(c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diag</td>
<td>diag(U)</td>
<td>To return a diagonal matrix and diagonals of the 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.</td>
</tr>
<tr>
<td></td>
<td>diag(U, m)</td>
<td>• k = 0 represents the main diagonal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• k &gt; 0 is above the main diagonal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• k &lt; 0 is below the main diagonal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If U is a square matrix, this function will return a column matrix formed by the elements of the kth diagonal of U.</td>
</tr>
<tr>
<td>exp</td>
<td>exp(x)</td>
<td>To return a Euler's number e raised to the power of a or c. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>exp(c)</td>
<td></td>
</tr>
<tr>
<td>eye</td>
<td>eye(m)</td>
<td>To return an identity matrix of dimension m x m.</td>
</tr>
<tr>
<td>factorial</td>
<td>factorial(m)</td>
<td>To return a factorial of m value.</td>
</tr>
<tr>
<td>floor</td>
<td>floor(x)</td>
<td>To return a largest (closest to positive infinity) value that is not greater than the value of x and is equal to a mathematical integer. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>floor(X)</td>
<td></td>
</tr>
<tr>
<td>IEEEremainder</td>
<td>IEEEremainder(x, y)</td>
<td>To compute the remainder operation in two arguments as prescribed by the IEEE 754 standard.</td>
</tr>
<tr>
<td>imag</td>
<td>imag(c)</td>
<td>To return a real value of an imaginary part of a given complex number. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>invert</td>
<td>invert(U)</td>
<td>To return an inverse or pseudo inverse of a given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the given matrix is a square matrix, the inverse of a U matrix will be returned using the LU factorization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the given matrix is not a square matrix, a pseudo inverse matrix will be returned using the QR factorization.</td>
</tr>
<tr>
<td>linsolve</td>
<td>linsolve(U, V)</td>
<td>X = linsolve(U,V) solves the linear system U*X = V using the LU factorization with partial pivoting when U is a square matrix.</td>
</tr>
<tr>
<td>In</td>
<td>In(x)</td>
<td>To return a natural logarithm (base e) of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>In(c)</td>
<td></td>
</tr>
<tr>
<td>log</td>
<td>log(x)</td>
<td>To return a natural logarithm (base e) of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>log(c)</td>
<td></td>
</tr>
<tr>
<td>log10</td>
<td>log10(x)</td>
<td>To return a logarithm base 10 of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>log10(c)</td>
<td></td>
</tr>
<tr>
<td>Function Name</td>
<td>Syntax</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>log2</strong></td>
<td>log2(x)</td>
<td>To return a logarithm base 2 of a given value.</td>
</tr>
<tr>
<td></td>
<td>log2(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>max(x, y)</td>
<td>To return a greater of two given values.</td>
</tr>
<tr>
<td></td>
<td>max(c, d)</td>
<td>• max(U) returns the largest element of a given matrix.</td>
</tr>
<tr>
<td></td>
<td>max(U)</td>
<td>• max(U, V) returns a matrix the same size as U and V with</td>
</tr>
<tr>
<td></td>
<td>max(U, V)</td>
<td>the largest elements taken from U or V. The dimensions of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U and V must be the same.</td>
</tr>
<tr>
<td><strong>mean</strong></td>
<td>mean(U)</td>
<td>To return a mean or average value of a given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• U is a row or column matrix: mean(U) returns the mean value of all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elements in the given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• U is a 2-D matrix: mean(U) returns a row matrix that</td>
</tr>
<tr>
<td></td>
<td></td>
<td>contains the mean value of each column of the given</td>
</tr>
<tr>
<td></td>
<td></td>
<td>matrix.</td>
</tr>
<tr>
<td><strong>median</strong></td>
<td>median(U)</td>
<td>To return a median value of a given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• U is a row or column matrix: median(U) returns the median value of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all elements in the given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• U is a 2-D matrix: median(U) returns a row matrix that</td>
</tr>
<tr>
<td></td>
<td></td>
<td>contains the median value of each column of the given</td>
</tr>
<tr>
<td></td>
<td></td>
<td>matrix.</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>min(x, y)</td>
<td>To return a smaller of two given values.</td>
</tr>
<tr>
<td></td>
<td>min(c, d)</td>
<td>• min(U) returns the smallest element of a given matrix.</td>
</tr>
<tr>
<td></td>
<td>min(U)</td>
<td>• min(U, V) returns a matrix the same size as U and V with</td>
</tr>
<tr>
<td></td>
<td>min(U, V)</td>
<td>the smallest elements taken from U or V. The dimensions of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U and V must be the same.</td>
</tr>
<tr>
<td><strong>ones</strong></td>
<td>ones(m, n)</td>
<td>To return an m x n matrix of all ones.</td>
</tr>
<tr>
<td><strong>pow</strong></td>
<td>pow(x, y)</td>
<td>To return a value of the first argument raised to the power of</td>
</tr>
<tr>
<td></td>
<td>pow(U, c)</td>
<td>the second argument.</td>
</tr>
<tr>
<td></td>
<td>pow(c, d)</td>
<td>This function operates element-wise on a given matrix U.</td>
</tr>
<tr>
<td><strong>random</strong></td>
<td>random()</td>
<td>To return a real value with a positive sign, greater than or equal to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0 but less than 1.0.</td>
</tr>
<tr>
<td><strong>real</strong></td>
<td>real(c)</td>
<td>To return a real value of the real part of a given complex number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td><strong>rint</strong></td>
<td>rint(x)</td>
<td>To return a value that is closest in value to an argument and is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>equal to a mathematical integer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td><strong>round</strong></td>
<td>round(x)</td>
<td>To return a closest value to an argument and is equal to a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mathematical integer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td><strong>sec</strong></td>
<td>sec(x)</td>
<td>To return a secant of a given value.</td>
</tr>
<tr>
<td></td>
<td>sec(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td><strong>secd</strong></td>
<td>secd(x)</td>
<td>To return a secant of a given value expressed in degree.</td>
</tr>
<tr>
<td></td>
<td>secd(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td><strong>sech</strong></td>
<td>sech(x)</td>
<td>To return a hyperbolic secant of a given value.</td>
</tr>
<tr>
<td></td>
<td>sech(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Syntax</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>sin</td>
<td>sin(x)</td>
<td>To return a trigonometric sine of an angle.</td>
</tr>
<tr>
<td></td>
<td>sin(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>sind</td>
<td>sind(x)</td>
<td>To return a sine of a given value, expressed in degree</td>
</tr>
<tr>
<td></td>
<td>sind(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>sinh</td>
<td>sinh(x)</td>
<td>To return a hyperbolic sine of a given value.</td>
</tr>
<tr>
<td></td>
<td>sinh(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>size</td>
<td>size(U)</td>
<td>To return a size of a given matrix. If only the matrix is passed to the function as an argument, the returned value is a 1x2 matrix. The first element is the number of rows and the second element is the number of columns.</td>
</tr>
<tr>
<td></td>
<td>size(U, m)</td>
<td>If the second parameter (m) is specified, this function will return the size of an mth dimension of a given matrix as a scalar value. The second argument can be 1 or 2 (1 for the row size and 2 for the column size). For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U = [1, 2, 3; 4, 5, 6];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size(U) is [2, 3];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size(U, 1) is 2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size(U, 2) is 3</td>
</tr>
<tr>
<td>sort</td>
<td>sort(U)</td>
<td>To sort the elements of a given matrix in an ascending or descending order. If the second argument is specified with 'ascend' or 'descend', the elements will be in an ascending or descending order respectively. If this function is called without a second argument, the elements will be sorted in an ascending order.</td>
</tr>
<tr>
<td></td>
<td>sort(U, 'descend')</td>
<td>• U is a row or column matrix: sort(U) and sort(U, ascend) sort all elements in the given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• U is a 2-D matrix: std(U) and std(U,flag) sort elements in each column of the given matrix.</td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrt(x)</td>
<td>To return a correctly rounded positive square root of a double value.</td>
</tr>
<tr>
<td></td>
<td>sqrt(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>std</td>
<td>std(U)</td>
<td>To return a standard deviation of a given matrix. The 'flag' argument can be 0 or 1. It specifies the method for calculating the standard deviation. If the flag = 0, the standard deviation is normalized by N-1. If the flag = 1, the standard deviation is normalized by N where N is the number of data. The value of the flag will be zero by default.</td>
</tr>
<tr>
<td></td>
<td>std(U, flag)</td>
<td>• U is a row or column matrix: std(U) and std(U, flag) returns the standard deviation of all elements in the given matrix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• U is 2-D matrix: std(U) and std(U,flag) returns a row matrix that contains the standard deviation of each column of the given matrix.</td>
</tr>
<tr>
<td>sum</td>
<td>sum(U)</td>
<td>To return a summation of all elements in a U matrix.</td>
</tr>
<tr>
<td>tan</td>
<td>tan(x)</td>
<td>To return a trigonometric tangent of an angle.</td>
</tr>
<tr>
<td></td>
<td>tan(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>tand</td>
<td>tand(x)</td>
<td>To return a tangent of a given value expressed in degree.</td>
</tr>
<tr>
<td></td>
<td>tand(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
</tbody>
</table>
12.3.7 Built-in Math Solver API for User-defined Functions

You can use the API provided by the built-in Math Solver to create user-defined functions. These functions can be used in the mathematical or logical expressions of constraint elements. To create a user-defined function, first, you have to create a new MagicDraw plugin. Second, you need to create a Java class that implements the UserDefinedFunction interface and third, register the created class to the built-in Math Solver.


12.3.7.1 Understanding UserDefinedFunction Interface

Cameo Simulation Toolkit provides a Java interface, which is the UserDefinedFunction interface to create the user-defined functions in Math Solver. The following three methods must be implemented in the Java class.

(i) String getName()

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

(ii) boolean isValidInputParameters(List<Value> parameters)

This method will be called by the built-in Math Solver to validate the input parameters before performing functional operations. The ‘parameters’ are the input parameters that are passed to the user-defined function. If all of them are valid, this method returns true. Otherwise, it will return false.

(iii) Value performFunction(List<Value> parameters)

This method will be called by the built-in Math Solver to perform the user-defined function operation. The ‘parameters’ are the input parameters which are passed to the user-defined functions. The implemented codes for calculating the result value from the given input parameters should be placed in this method.

For example, to evaluate a user-defined function for polynomial value evaluation from a given polynomial coefficient and the value of that polynomial, you need to create a new Java class and name as “PolyvalFunctionDescriptor” to implement the UserDefinedFunction interface.

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

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));
        result = result.add(tmp);
    }
    return result;
}
12.3.7.2 Registering User-defined Function to Math Solver Using SimpleMathEngine Class

The SimpleMathEngine class represents the built-in Math Solver. The Java class that implements the UserDefinedFunction interface must be registered to this class when the created plugin is initialized.

For example, if the class UDFSamplePlugin is inherited from the plugin, the following is the implementation code:

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

public class UDFSamplePlugin 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;
    }
}
```
12.4 Using MATLAB® as Mathematical Solver

Cameo Simulation Toolkit can use MATLAB®, which is installed on the local machine, to solve mathematical expressions. You can use external math solvers such as MATLAB® if the integrators are present. Such integrators will be provided in the subsequent release(s) of Cameo Simulation Toolkit.

NOTE: The current release of Cameo Simulation Toolkit can integrate with MATLAB® only on Microsoft Windows and Mac OS 10.6 (Snow Leopard).

12.4.1 Setting Up System to Call MATLAB® from Cameo Simulation Toolkit

To use MATLAB® on Microsoft Windows 32-bit and 64-bit:

1. Install MATLAB®.
2. Press Window + R to open the Run dialog.
3. Type “cmd” in the open combo box and click OK to open the command prompt window.
4. Type “matlab /regserver” and press Enter to register the MATLAB® components to Windows (Figure 131). The MATLAB command window will open and be ready to use.
5. Add the path of the MATLAB® bin and bin/win32 (or bin/win64 for Microsoft Windows 64-bit) folders to the Path environment variable using the following steps:
   5.1 Double-click System in Control Panel to open the System Properties dialog (Figure 132). Click the Advanced tab.
5.2 Click the **Environment Variables** button to open the **Environment Variables** dialog (Figure 133).
5.3 Select **Path** from the System variables group and click the **Edit** button to open the **Edit System Variable** dialog (Figure 133).

5.4 Enter the path to the MATLAB® bin and bin/win32 folders (or bin/win64 for Microsoft Windows 64-bit) in the **Variable value** box (Figure 134), for example, "C:\Program Files\MATLAB\R2010b\bin;C:\Program Files\MATLAB\R2010b\bin\win32;".

5.5 Click **OK**.


To use MATLAB® on Mac OS 10.6 (Snow Leopard):

1. Install MATLAB®.
2. Type the following commands into terminal to show all files in Finder:
3. Add the DYLD_LIBRARY_PATH variable to Mac OS:
   3.1 Create an empty text file in the /etc folder and name it: launchd.conf.
   3.2 Open it with a text editor, for example, TextEdit, and type the following text (no space):

   ```
   setenv DYLD_LIBRARY_PATH /Applications/MATLAB_R2010b.app/bin/mac64:
   /Applications/MATLAB_R2010b.app/runtime/mac64
   ```

   3.3 Save the text file as launchd.conf to the desktop.
   3.4 Move the launchd.conf file to the /etc folder.

4. Create a link to the MATLAB® executable file in /usr/bin if it does not yet exist.

5. Call the following commands into terminal:
   - $ cd /usr/bin
   - $ ln -s /Applications/MATLAB_R2010b.app/bin/matlab matlab

6. Type the following commands into terminal to reset Finder:
   - $ defaults write com.apple.finder AppleShowAllFiles FALSE
   - $ killall Finder

7. Restart Mac OS.

To use MATLAB® on Linux 32-bit and 64-bit (tested with Ubuntu):

1. Install MATLAB® (it is assumed that your MATLAB installation directory is /home/username/MATHWORKS_R2011A).

2. Make sure C Shell has already been installed on your Linux. To install C Shell on Ubuntu, you can type the following command into terminal:
   - ~$ sudo apt-get install csh

3. Create a link to the MATLAB® executable file in /usr/bin if it does not yet exist, and type the following commands into terminal:
   - ~$ sudo -i
   - ~$ cd /usr/bin
   - ~$ ln -s /home/username/MATHWORKS_R2011A/bin/matlab matlab

4. Add the MATLAB® bin folder to LD_LIBRARY_PATH of Java.

5. Use a text editor to open the mduml file in the bin folder in the MagicDraw installed directory. Type the following text under the line that contains cd "$APP_HOME” and save the mduml file (Figure 135):

   - on Linux 32-bit, type: export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:
   - on Linux 64-bit, type: export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:
     home/username/MATHWORKS_R2011A/bin/glxa64 : /home/username/MATHWORKS_R2011A/sys/os/glxa64
12.4.2 Selecting MATLAB® as the Mathematical Solver for Cameo Simulation Toolkit

You can use MATLAB® as your mathematical solver by selecting it the Mathematical Engine field in the Environment Options dialog.

To use MATLAB® as your mathematical solver:

1. Select Options > Environment on the main menu bar. The Environment Options dialog will open (Figure 136).
2. Select Simulation on the left-hand side pane.
3. Select MATLAB from the Mathematical Engine drop-down list.
4. Click OK.
13. 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 install other JSR233 compatible language implementations. Any value specifications in a model (like guards, constraints, decisions, default values, and 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 Section 1.2 Key Features)
- Other additional downloadable JSR-223 script engines (see http://scripting.dev.java.net/)