# CAMEO SIMULATION TOOLKIT

1. **Getting Started**  
   1.1 Introduction to Cameo Simulation Toolkit  
   1.2 Key Features  
   1.3 Installation  
   1.4 Simulation Project Templates  
      1.4.1 Simulation Project Template  
      1.4.2 Simulation SysML Project Template  
2. **Model Execution**  
   2.1 Simulation by Executing Elements  
      2.1.1 Behaviors  
      2.1.2 Class  
      2.1.3 Diagram  
      2.1.4 Instance Specification  
   2.2 Simulation by Executing an Execution Configuration  
   2.3 Subset Property  
3. **Execution Configuration and UI Modeling**  
   3.1 ExecutionConfig Stereotype  
   3.2 Execution Log  
   3.3 Simulation Time and Simulation Clock  
      3.3.1 Built-in Clock  
      3.3.2 Model-based Clock  
   3.4 Automatically Starting Active Objects  
   3.5 UI Modeling Diagram Execution  
   3.6 ImageSwitcher and ActiveImage  
      3.6.1 Representing Object States  
      3.6.2 Representing Enumeration Values  
   3.7 Time Series Chart  
      3.7.1 Specifying the Range of the Time Axis in a Time Series Chart  
      3.7.2 Exporting Plots Data  
   3.8 Nested UI Configuration Stereotype  
      3.8.1 Representing Parts of an Execution Context  
      3.8.2 Representing Parts Using Existing UI Configurations  
   3.9 Reusable UI Mockup  
4. **Animation**  
   4.1 Active, Visited, and Last Visited Elements  
   4.2 Customizing Animation Colors  
   4.3 Auto Open Diagrams Option  
   4.4 Opening a Diagram of an Executing Behavior  
5. **Simulation Debugging**  
   5.1 Understanding Simulation Sessions  
   5.2 Simulation Debugging Process  
   5.3 Simulation Console  
      5.3.1 Console Pane  
      5.3.2 Simulation Information  
      5.3.3 Simulation Log File  
      5.3.4 User Output Filter  
   5.4 Runtime Value Monitoring  
      5.4.1 Context, Runtime Object, and Runtime Value  
      5.4.2 Variables Pane  
      5.4.3 Time Series Chart  
      5.4.4 Creating Runtime Objects in the Variables Pane
10.7 Communicating with Evaluators through Simulation Console 194
10.8 Exchanging Values between Cameo Simulation Toolkit and the Parametric Evaluator 195
   10.8.1 Exchanging Values between Slot and Mathematical Environment 195
   10.8.2 Exporting Runtime Values to the Parametric Evaluator 196
10.9 Built-in Math 198
   10.9.1 Evaluating Strings from Command Input 198
   10.9.2 Variables 199
   10.9.3 Values 199
   10.9.4 Constants 201
   10.9.5 Operators 201
   10.9.6 Functions 203
10.10 Integration with External Evaluators 209
   10.10.1 Integration with MATLAB® 209
   10.10.2 Integration with Maple™ 214
   10.10.3 Integration with Mathematica® 218
10.11 Trade Study with Cameo Simulation Toolkit 220
10.12 Sample Projects 223
11. Simulation of SysML Models 224
11.1 Supported SysML Elements 224
   11.1.1 AcceptChangeStructuralFeatureEventAction 224
   11.1.2 AdjunctProperty 225
   11.1.3 BindingConnector 225
   11.1.4 Block 226
   11.1.5 BoundReference 226
   11.1.6 ChangeStructuralFeatureEvent 226
   11.1.7 ClassifierBehaviorProperty 227
   11.1.8 ConstraintBlock 227
   11.1.9 FlowProperty 227
   11.1.10 FullPort 227
   11.1.11 InvocationOnNestedPortAction 228
   11.1.12 NestedConnectorEnd 229
   11.1.13 Probability 229
   11.1.14 Proxy Port 230
   11.1.15 TriggerOnNestedPort 232
   11.1.16 ValueType 232
11.2 Requirements Traceability from the Variables Pane 234
   11.2.1 Requirement Refined by a Constraint Block 234
   11.2.2 Requirement Satisfied by a Property 235
12. Action Languages 235
   12.1 Supported Scripting Languages 236
   12.2 Importing External Libraries 236
1. Getting Started

Cameo Simulation Toolkit is a MagicDraw plugin, which provides a unique set of tools supporting the standardized construction, verification, and execution of computational complete models based on a foundational subset 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. The purpose of this Getting Started section is to provide you with an overview of the plugin and a brief explanation of how exactly this plugin works.

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 experimented 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. Cameo Simulation Toolkit provides the solutions that enable 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 functionalities of Cameo Simulation Toolkit are as follows:

(i) Simulation Window:
   - Toolbars and Debugger pane: to control execution or a model simulation.
   - Simulation Console: to execute log outputs and command lines for active engines.
   - Sessions pane: to select particular sessions of execution.
   - Variables pane: to monitor the runtime values of each execution session.
   - Breakpoints pane
   - Trigger 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 Parametric Evaluator 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) **Parametric Execution Engine**: Enabling Cameo Simulation Toolkit to execute SysML parametric diagrams. This engine requires SysML plugin for MagicDraw in order 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 the Resource/Plugin Manager option in MagicDraw to download, import, and install the plugin, or (ii) follow the manual installation instructions if you have already downloaded the plugin.

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

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

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 **Message** dialog informing you that the installation is complete 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>`.


1.4 Simulation Project Templates

Cameo Simulation Toolkit provides two templates that consist of fundamental diagrams and elements basically required for the user to further develop the simulation work to abundantly reduce time working on the preparation of the simulation from scratch. The two supporting templates are as follows (Figure 1):

1.4.1 Simulation Project Template

1.4.2 Simulation SysML Project Template
These template options are available in the New Project dialog. You can open the dialog by clicking File > New Project on the MagicDraw main menu.

### 1.4.1 Simulation Project Template

Working with UML-based notations, the Simulation Project template arranges basic and necessary diagrams represented by the Class and Package diagrams (Figure 2) including the State Machine diagrams. The project template additionally encompasses an Execution Configuration to which an Execution Target is set.

![Figure 2 -- The Simulation Project Template with the Class and Package Diagrams](image)

### 1.4.2 Simulation SysML Project Template

You will need both Cameo Simulation Toolkit and SysML plugins to use this project template. Instead of using the Class and Package diagrams, the Simulation SysML Project template makes use of blocks and Block Definition diagrams to represent the work in accordance with the principles of system modeling language (Figure 3). This template also provides State Machine diagrams and an Execution Configuration with an Execution Target.

![Figure 1 -- Simulation Project Template and Simulation SysML Project Template](image)
2. Model Execution

Cameo Simulation Toolkit allows you to execute a simulation of elements in a MagicDraw project. Executable elements are those that are 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 below shows a list of currently supported execution engines in Cameo Simulation Toolkit.
Table 1 -- Currently Supported Execution Engines

<table>
<thead>
<tr>
<th>Execution Engine</th>
<th>Supported Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity Execution Engine</td>
<td>• Activities.</td>
</tr>
<tr>
<td></td>
<td>• Activity Diagrams.</td>
</tr>
<tr>
<td></td>
<td>• Classes whose classifier behavior is an Activity.</td>
</tr>
<tr>
<td></td>
<td>• InstanceSpecifications of a class whose classifier behavior is an Activity.</td>
</tr>
<tr>
<td>State Machine Execution Engine</td>
<td>• State Machines.</td>
</tr>
<tr>
<td></td>
<td>• State Machine Diagrams.</td>
</tr>
<tr>
<td></td>
<td>• Classes whose classifier behavior is a State Machine.</td>
</tr>
<tr>
<td></td>
<td>• InstanceSpecifications of a Class whose classifier behavior is a State Machine.</td>
</tr>
<tr>
<td>Parametric Execution Engine</td>
<td>• Blocks that contain Constraint Properties.</td>
</tr>
<tr>
<td></td>
<td>• SysML Parametric Diagrams.</td>
</tr>
<tr>
<td></td>
<td>• InstanceSpecifications of a Block that contains Constraint Properties.</td>
</tr>
</tbody>
</table>

To create a simulation, either (2.1) execute elements that are supported by the execution engines or (2.2) create an execution configuration including setting a 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. You can right-click an element that you would like to execute to open the Simulation menu.

To execute a model through the context menu:

1. Right-click an element either (i) on a diagram (Figure 4) or (ii) in the containment browser (Figure 5), and then select Simulation > Execute.
Figure 4 -- Executing a Model through the Element’s Context Menu in a Diagram
2. The Simulation window will open. The simulation session will automatically start and open 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 6) to execute the model.

### NOTE
Cameo Simulation Toolkit uses different execution engines to execute different kinds of elements as follows:
- Behaviors
- Classes
- Diagrams
- Instance Specifications

#### 2.1.1 Behaviors
You can select a behavior (Activity, State Machine, or Interaction) and execute it.

##### 2.1.1.1 Activity
Whenever you execute an Activity behavior (Figure 7), Cameo Simulation Toolkit will simulate it on an Activity diagram whose context contains the selected Activity (Figure 8). 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 7 -- Executing an Activity Behavior
2.1.1.2 State Machine

Whenever you execute a State Machine behavior (Figure 9), Cameo Simulation Toolkit will simulate it on a State Machine diagram whose context contains the selected State Machine (Figure 10).
Figure 9 -- Executing a State Machine Behavior
2.1.3 Interaction

If you execute an Interaction behavior (Figure 11), Cameo Simulation Toolkit will simulate it on a Sequence diagram whose context contains the selected Interaction (Figure 12).
Figure 11 -- Executing an Interaction
To execute a behavior from an Activity, a State Machine diagram, or an Interaction directly:

- Either:
  
  (i) open the diagram and click **Execute** 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. Cameo Simulation Toolkit will create a simulation session to execute the selected class and create a runtime value whose type is the selected Class to store the simulation values. If the selected class has a defined classifier behavior, either an activity or a state machine (Figure 13), it will also be executed. For example, if you execute the Calculator class (Figure 9), the simulation will be performed on the Calculator state machine as shown in Figure 10 above.
If the class does not have a defined classifier behavior (Figure 14), the parametric will be executed instead (only if the selected class is a SysML Block containing Constraint Property(ies)).
Figure 15 -- Animation of a Parametric Execution
2.1.3 Diagram

To execute a diagram simulation:

- Right-click a diagram and select Simulation > Execute (Figure 16). Cameo Simulation Toolkit will execute the element of the context of the diagram the same way it does behaviors or classes.

2.1.4 Instance Specification

You can also simulate an InstanceSpecification. Both 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 runtime objects and values in Section 5.4.5 Creating Runtime Objects from Classifiers.

To execute an InstanceSpecification simulation:

- Right-click an InstanceSpecification and select Simulation > Execute. Cameo Simulation Toolkit will execute the classifier of the selected InstanceSpecification the same way it does
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 17).

![Diagram of package, classifier, system, and instance]

Figure 17 -- Runtime Values Resulting from the InstanceSpecification Execution

2.2 Simulation by Executing an Execution Configuration

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

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

- Right-click an Execution Configuration and select Simulation > Execute (Figure 18).
(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 appear in the list) on the Simulation Control toolbar and click the Run '<name of execution configuration>' Config button (Figure 19).

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

2.3 Subset Property

Cameo Simulation Toolkit supports and includes subset properties in execution. It initializes subsets at the beginning of execution and also updates their values during execution.
Figure 20 demonstrates a model with subsets.

When Cameo Simulation Toolkit starts executing the model, it will add all runtime values of the subsetting properties to the subsetted properties as illustrated in Figure 21.

You can update or add a value in the subsetting or subsetted properties. Removing or changing a value in the set of values of a subsetting property causes it to disappear from or change in the set of values in the subsetted property, and vice versa. Adding a value to the set of values in the subsetted property causes it to appear in the set of values in the subsetting property.
3. Execution Configuration and UI Modeling

3.1 ExecutionConfig Stereotype

Cameo Simulation Toolkit provides a model-based execution configuration through the «ExecutionConfig» stereotype (Figure 22). The «ExecutionConfig» configuration properties consist of:

- **UI**: A user interface for configuration mockups that will start with execution of a model.
- **silent**: If the value is true, execution will run without animation (or idle time).
- **executionTarget**: An element from which execution should start.
- **log**: An element in which the execution trace will be recorded.
- **resultLocation**: A Package, a Model, or an InstanceSpecification in which a context object will be stored after execution.
  - If the **resultLocation** is specified by an InstanceSpecification, the values of the context object will be saved as slot values of the specified InstanceSpecification.
  - If the **resultLocation** is specified by a Package or a Model, Cameo Simulation Toolkit will create a new InstanceSpecification owned by the Package or the Model after the execution. Then, the values of the context object will be saved as slot values of the created InstanceSpecification.
  - If the **resultLocation** is not specified, the execution results will not be saved even though the executionTarget is the InstanceSpecification.
- **enginesPriority**: Execution engines that can be used to execute a model ordered by priority. The first engine on the list has the highest priority. If the execution configuration does not have a tagged value for this tag definition, Cameo Simulation Toolkit will use the values that are defined in the registered Execution Engine Priority in the Environment Options dialog.
- **autorun**: If the value is true, execution of a model will start running automatically once it has been initialized. Otherwise, you need to click the Run Execution button in the Simulation window to run the execution.
- **clock ratio**: A ratio between a simulation clock and a real-time clock (1:10). 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 an 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`.

• **decimalPlaces**: Decimal places of all displayed numerical values in a model execution, for example, in the Variables pane, Tooltip, and Simulation Console pane. Their values must be integers. If the precision of displayed numerical values is greater than or equals to 10% of the absolute value, the numerical values will be displayed in exponential form.

• **fireValueChangeEvent**: If this value is true, a parametric execution will be re-executed immediately whenever the value of any structural feature changes. Re-execution will affect both values carried over by binding connectors and specified in a constraint.

• **timeValue**: A property that will be used in a model-based clock. If the tagged value is specified, the run-time value specified by the property will be used as the simulation time.

• **timeUnit**: A unit of a runtime value that specifies the simulation time. If the tagged value is unspecified, the millisecond will be used by default.

• **constraintFailureAsBreakpoint**: A flag that indicates a simulation will pause at the constraint element where the failure occurs.

You can select and execute an Execution Configuration directly from the `<<ExecutionConfig>>` configuration properties.

### 3.2 Execution Log

You can record all runtime event occurrences into a specific model element by:

(i) creating a new ExecutionLog element (a Class to which the «ExecutionLog» stereotype is applied) and
(ii) making a reference to the “log” property in an ExecutionConfig before a simulation takes place (Figure 24).

![Figure 24 -- 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 start time can be seen as the name of an attribute. You can also record the following runtime data (see Figure 25):

- **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 25).
- **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).
3.3 Simulation Time and Simulation Clock

When you execute a model related to time, for example, a transition with a time trigger, Cameo Simulation Toolkit will obtain simulation time from a simulation clock. The simulation time is the amount of time spent executing a model. Cameo Simulation Toolkit also uses the simulation time in a timestamp of a signal instance in the Simulation Log (see Section 3.2 Execution Log) and in a time series chart (see Section 3.7 Time Series Chart).

There are two types of simulation clocks in Cameo Simulation Toolkit:

3.3.1 Built-in Clock

3.3.2 Model-based Clock

The built-in clock is the default simulation clock. You can select the model-based clock by making the property as the time value tag definition of an execution config. See section 3.3.2 Model-based Clock below for further details on the model-based clock.

3.3.1 Built-in Clock

The built-in clock generates simulation time continuously during execution of a model. You can use the clock ratio option in Cameo Simulation Toolkit to increase or decrease the amount of simulation time. The clock ratio specifies the ratio between actual time and simulation time in the same interval. You can set a ratio of the built-in clock for model execution. The value you enter for the clock ratio tag definition of an execution config allows you to scale a ratio up or down (Figure 26). For further information about stereotypes of the execution configuration, see section 3.1 ExecutionConfig Stereotype above.
For example, if a clock ratio is 10, it means that 10 seconds current time are equivalent to one second simulation time in the built-in clock. Therefore, the simulation clock is 10 times slower than the actual clock.

### Note
- The current version of Cameo Simulation Toolkit allows you to pause or resume the built-in clock.
- When you pause an execution by either clicking the **Suspend** button or using breakpoints, Cameo Simulation Toolkit will stop the built-in clock operation and all running sessions temporarily. You can continue all of the activities by clicking the **Resume** button.

### 3.3.2 Model-based Clock

Occasionally, you may want to have full control over the amount of simulation time to execute your model. Cameo Simulation Toolkit allows you to use a run-time value of any selected property to determine the simulation time. With this option, you can control the value of the simulation time with your model. The model that enables you to use the run-time value as the simulation time is called “model-based clock”.

Using the model-based clock requires you to specify the property, that is the run-time value that you will use as the simulation time, as the value of a `timeValue` tag definition of an execution config. You can specify the unit value for the simulation time through the `timeUnit` tag definition of the execution config. If the `timeUnit` is unspecified, the millisecond will be used by default.
When you select to use a model-based clock, Cameo Simulation Toolkit will ignore the clock ratio. The following are constraints of the model-based clock:

- There should be only one object of a classifier whose property is specified as timeValue. In Figure 27 above, the property time of the block Clock has been set as the timeValue of the System Execution, which is an execution config for executing the System block. There should be only one Clock object in each execution.

- You should not use a time event to update simulation time in a clock model. Because, it obtains simulation time from the model-based clock. In addition, a time event will be fired at a specific time or duration and simulation time cannot be updated unless it is fired. Therefore, if you use a time event, simulation time will not be updated at all.
3.4 Automatically Starting Active Objects

An active object is a runtime object that is typed by an active class (a Class element whose isActive is true). When the value of an autoStartActiveObjects tag definition is true, the classifier behavior of the active class will start automatically right after the object is instantiated. Figure 28 below shows the Stereo System execution config where the value of autoStartActiveObjects is true. In this example, the behavior of the active objects (Speaker, Headphone, and Player) will start automatically once the objects start.

If the value of autoStartActiveObjects is false or the classes or blocks are not active classes, you have to start each object using a startObjectBehavior action. Figure 29 below shows the System init activity, which is the classifier behavior of a stereo system block. This behavior uses the startObjectBehavior actions to start the behaviors of the runtime objects (big speakers, small speakers, and dvd player). If the value of autoStartActiveObjects is true and the Speaker’s, Player’s, and Headphone’s blocks are active blocks, the System init can be simplified as shown in Figure 30.
Figure 29 -- System init Activity to Start the Behavior of Runtime Objects Using startObjectBehavior Actions

Figure 30 -- A Simplified System init Activity to be Used with autoStartActiveObjects Whose Value is True
3.5 UI Modeling Diagram Execution

The MagicDraw User Interface Modeling diagram becomes even more powerful and valuable when used with Cameo Simulation Toolkit. UI components including frames, panels, group boxes, text fields, check boxes, sliders, labels, buttons, combo boxes, spinners, and radio buttons are supported.

(i) Frames

You need to drag a Classifier to a UI Frame to bind them together (and Cameo Simulation Toolkit will automatically apply a «UI» stereotype and set its represents tag to the Classifier). Once bound, the UI Frame can represent 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, combo boxes, spinners, radio buttons, and even panels themselves).

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

During run-time, UI components in the UI Panel can also display their documentation as a tooltip (Figure 31).
(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 with that particular UI component (the «RuntimeValue» stereotype will be automatically applied; its “element” tag will 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 with that particular UI Label (the «RuntimeValue» stereotype will be automatically applied; its “element” tag will 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

You can use a UI button to (a) send Signal(s), (b) call Operation(s), or (c) call Behavior(s):

(a) To use a UI button to send a Signal:

- Drag a Signal to a UI button to associate it with the 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 as the name of such Signal).

In addition, if the signal has attributes, runtime values will be automatically created for all of the attributes and placed under the UI button. You can specify the values, which will be used at runtime, in the “value” tag of these runtime values. If you click this UI button during execution, it will send the associated Signal along with its runtime values.

(b) To use a UI button to call an operation:

- 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).

If you click this UI button is during execution, it will call the associated Operation.

(c) To use a UI button to call a behavior:

- Drag a Behavior, for example, an Activity, to a UI button to associate it with the 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).

If you click this UI button during execution, it will call the associated Behavior.

(vii) Combo Boxes and Spinners

Drag a Property typed by Enumeration to one of these two UI components (ComboBoxes and Spinners) 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 will represent the Property typed by Enumeration. Once represented, the UI component will reflect the value (the selected Enumeration Literal) of the represented Property in the Variables pane during execution, and vice versa.

(viii) Radio Buttons

Drag a Property typed by Enumeration to a UI Panel or GroupBox to bind the Property to that particular UI panel or GroupBox (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). All Enumeration Literals of the Property type will be automatically created as a vertical list of UI RadioButtons. Each UI RadioButton will represent Enumeration Literal of the Property type accordingly. Once represented, the UI RadioButton will display the value of the represented Property in the Variables pane during execution.

In addition, you can assign or re-assign other Enumeration Literals to an existing UI RadioButton by dragging other Enumeration Literals of the same Property type to the existing UI RadioButton. Also, you can create a single UI RadioButton to represent a particular Enumeration Literal by dragging the Enumeration Literal to a UI Panel or GroupBox (the «RuntimeValue» stereotype will be automatically applied, its “element” tag will then be set to the Enumeration Literal, and its “Text” tag will also be set to the name of such Enumeration Literal).
Figure 32 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 it with the button.
3. Specify a value for each UI button's runtime value (the runtime value is located under each UI button).
4. Drag any Classifier’s property to a UI label to be represented.
5. 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, sliders, combo boxes, spinners, and radio buttons 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.
3.6 ImageSwitcher and ActiveImage

ImageSwitcher is a predefined subtype of UI config. It is a simple, yet flexible and powerful animation tool. You can use ImageSwitcher to represent the state or the enumeration value of a runtime object. 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 an «ActiveImage» and has the following properties:

- **Image**: an image that will be used in animation (from browsing the file or dragging the image directly from a web browser).
- **activeElement**: an element that will use an image once it is activated. An active image represents a state of a runtime object, whereas an activeElement is the state of a classifier represented by the ImageSwitcher. Whereas the ImageSwitcher represents an enumeration, the activeElement is the enumeration literal owned by the enumeration.
- **onClick**: a signal that will be triggered once an image is clicked.

### 3.6.1 Representing Object States

You may use an ActiveImage to represent each state of a runtime object whose classifier is represented by the ImageSwitcher that owns the ActiveImage.

To use the ImageSwitcher and ActiveImage to represent a state:

1. Create an ImageSwitcher element and set its represent tag definition with a classifier whose state will be represented by an active image owned by the ImageSwitcher.
2. Create an ActiveImage in the ImageSwitcher for each state of the classifier represented by the ImageSwitcher.
3. Specify the image that will be the image attribute of each created ActiveImage and set the state, which will be represented by the ActiveImage, as the tagged value of the activeElement tag definition.

Figure 33 below demonstrates an example of how to use ImageSwitcher and ActiveImage to represent the states of a runtime object (see the FlashingLight.mdzip sample):
3.6.2 Representing Enumeration Values

To use the ImageSwitcher and ActiveImage to represent an enumeration value:

1. Create an ImageSwitcher element and specify the enumeration, whose values will be represented by an ActiveImage of the ImageSwitcher, as the tagged value of the **represent** tag definition of the ImageSwitcher element.

2. Create an Active image for each enumeration literal owned by the enumeration.

3. Specify an image that will be the **image** attribute of each created ActiveImage and set the enumeration literal that will be represented by the ActiveImage as the **activeElement** tag definition.

You can use `test_imageswitcher_for_enum.mdzip` as your example. This file is a sample project of Cameo Simulation Toolkit. It is located in the samples/simulation/tests folder of the MagicDraw installed directory.
this example, an ImageSwitcher was created to represent a VerdictKind enumeration. An ActiveImage was created for each enumeration literal of the VerdictKind as shown in Figure 34.

![Image Switcher and Active Image Representing Enumeration and Enumeration Literals](image)

**Figure 34 -- Image Switcher and Active Image Representing Enumeration and Enumeration Literals**

If a property is typed by an enumeration, you can use the ImageSwitcher and the ActiveImage to represent it with a UI frame mockup of the classifier that owns the property. Once you drag the property to the panel owned by the UI frame mockup, Cameo Simulation Toolkit will prompt you to select either an ImageSwitcher or a group of radio buttons to represent the value. If you select the ImageSwitcher, the ActiveImage that represents the enumeration literal, which is the default value of the property, will open in the panel or in the group box to which the property has been dragged.

Figure 35 below shows the User Interface Modeling diagram of the `test_imageswitcher_for_enum.mdzip` sample project. It contains a UI frame that represents the Test Result Dialog, which is the classifier owning the property typed by the VerdictKind enumeration. The property is `result:VerdictKind` whose default value is `pass`. So, the ActiveImage that represents the value `pass` will be shown in the panel.
Figure 35 -- UI Modeling Diagram Containing a UI Frame with Image Switcher

When the UI mockup opens on the MagicDraw window during model execution, the image shown in the panel containing the ImageSwitcher will be the ActiveImage that represents the enumeration literal, which is the runtime value of the property.

You can execute the Test Result Dialog Execution in the sample project `test_imageswitcher_for_enum.mdzip`. You can also change a result value by selecting one of the radio buttons that represent the value: pass, fail, inconclusive, or error (Figure 36). And the image that corresponds to the value will appear.

Figure 36 -- UI Mockup Showing the Property’s Value typed by Enumeration

You can see more information about reusing ImageSwitcher in the UI mockup in section 3.8 Nested UI Configuration Stereotype.

Another more comprehensive and realistic sample available in Cameo Simulation Toolkit is Blackjack.mdzip. This sample contains Blackjack game simulation where Players can only stand or hit and Dealer can stand on 17 or higher. Once you execute the Run Blackjack execution config, the Blackjack game will begin by showing
a Blackjack game table UI (Figure 37).

Note that this sample also demonstrates the Nested UI Configuration feature (see Section 3.8 for more information). As the Dealer/Player has 7 possible card slots, according to the Nested UI Configuration feature, the Dealer/Player thus has 7 corresponding Value Properties, for example, ‘Card1’, ‘Card2’, ‘Card3’, and so on.

![Blackjack Table](Blackjack Table)

The image showing card slots in Figure 37 demonstrates the application of ImageSwitcher and enumeration. Each card slot is in fact a UI Panel having the NestedUIConfig stereotype applied where the NestedUIConfig's config tag is set to the Card Image Switcher ImageSwitcher (and the NestedUIConfig's feature tag is set to the corresponding Value Property, for example, ‘Card1’). The Card Image Switcher ImageSwitcher is set to represent the Card Value enumeration. The Card Image Switcher ImageSwitcher contains 53 ActiveImages each of which is the corresponding Enumeration Literal of the Card Image enumeration (the Card Value enumeration also contains 53 Enumeration Literals).

- ‘0’ Enumeration Literal represents a blank image; a card slot without any card.
- ‘1’ - ‘52’ Enumeration Literals represent the images of 52 distinct cards (Figure 38).
Figure 38 -- Image Switcher Containing Active Image of 52 Cards and Blank Image
When the value of one of the 7 Value Properties changes, the card slot image will then change accordingly. See the Player’s cards in Figure 39 to better understand how it works.

- The value of the Player’s ‘Card1’ Value Property is ‘49’ Enumeration Literal, which is the activeElement of the ‘49’ ActiveImage. The ‘49’ ActiveImage represents ten of spades.
- The value of the Player’s ‘Card2’ Value Property is ‘28’ Enumeration Literal, which is the activeElement of the ‘28’ ActiveImage. The ‘28’ ActiveImage represents two of hearts.
- The value of the Player’s ‘Card3’ Value Property is ‘0’ Enumeration Literal, which represents a blank card slot. Therefore, no card is open next to the two of hearts card.

![Figure 39 -- UI and Variable Panel Showing the Value of 7 Value Properties and the Displayed Image](image-url)
3.7 Time Series Chart

A time series chart displays plots of runtime values with respect to simulation time. You can show a time series chart of any runtime values during model execution.

To display a time series chart:

1. Open the context menu of the Variables pane.
2. Right-click the row of a runtime value, which needs to be shown on the time series chart and select **Show in time series chart** (see section 5.4.3 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 ImageSwitcher.

To use a 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 (Figure 40).

![Figure 40 -- Time Series Chart Specification](image)
You can change the time series chart display by modifying 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>fixedTimeLength</td>
<td>To specify a fixed range of the time axis in a time series chart. If you specify a value, the time axis range will be fixed to that particular value. If you do not specify the value of Fixed Time Location, the plot(s) will move to the left if the time range is greater than the maximum value of the time axis. A fixed time length unit is specified by a time unit tag definition of the execution config. If you do not specify the time unit, Cameo Simulation Toolkit will use the millisecond as the default unit.</td>
</tr>
<tr>
<td>fixedTimeLocation</td>
<td>To specify the start time of the time axis in a time series chart. If you specify a value for the Fixed Time Length, but leave the Fixed Time Location value empty, the time series chart will work like an oscilloscope. The plot(s) will move to the left. A fixed time location unit is specified by a time unit tag definition of the execution config. If you do not specify the time unit, Cameo Simulation Toolkit will use the millisecond as the default unit.</td>
</tr>
<tr>
<td>fixedRange</td>
<td>To specify whether a time series chart will automatically adjust the range on 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 a vertical grid line.</td>
</tr>
<tr>
<td>gridY</td>
<td>To show or hide a horizontal 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 a plot color.</td>
</tr>
<tr>
<td>refreshRate</td>
<td>To specify a time interval to refresh a time series chart. A refresh rate unit is specified by a time unit tag definition of the execution config. If no time unit specified, the millisecond will be used as the refresh rate unit.</td>
</tr>
<tr>
<td>title</td>
<td>To specify the title of a time series chart.</td>
</tr>
<tr>
<td>yUnit</td>
<td>To specify the unit of value on Y-Axis.</td>
</tr>
</tbody>
</table>
Figure 41 -- TimeSeriesChart in the MotionAnalysis.mdzip Sample Project
3.7.1 Specifying the Range of the Time Axis in a Time Series Chart

You can specify the range of the time axis (horizontal axis) in a time series chart by providing the values of `fixedTimeLength` and `fixedTimeLocation`. The `fixedTimeLength` and `fixedTimeLocation` units are specified by `timeUnit` when a model-based clock is used (for more information about model-based clock see section 3.3.2 Model-based Clock). Otherwise, the unit for value is the millisecond.

You can use the property `fixedTimeLength` to specify the range of the time axis. If you do not define the value of `fixedTimeLength`, the time axis will expand as long as the maximum duration a simulation will run. Figure 43 shows a time series chart when the `fixedTimeLength` value is 100 seconds.
You can also change the minimum value of the time axis in a time series chart by specifying the value for `fixedTimeLocation`. Figure 44 shows a time series chart when the `fixedTimeLength` value is 100 seconds and the `fixedTimeLocation` value is 50 seconds.
If the value you define only that of the fixedTimeLength value, the minimum value of the time axis will change automatically to adjust to the total length of time that you have defined as the value of fixedTimeLength. This situation will occur when the simulation time is greater than fixedTimeLength. If this happens, the maximum time axis value will be the actual simulation time and the minimum time axis value will increase to preserve the current value of fixedTimeLength.

In Figure 45a below, if fixedTimeLength is 100 seconds, the range of the time axis will be [0, 100] (if you do not specify fixedTimeLocation, the minimum time axis value will be 0). However, if your simulation model runs longer than the simulation time or more than 100 seconds (Figure 45b), for example, 115 seconds; the range of the time axis will be [15, 115]. This particular behavior of Cameo Simulation Toolkit's Time Series Chart is similar to that of Oscilloscope when fixedTimeLocation is unspecified.
3.7.2 Exporting Plots Data

A time series chart provides two options for you to export your plot data in which time values (the values that appear on the horizontal axis) and the corresponding recorded values (the values that appear on the vertical axis) are plotted in a time series chart. You can export the plot values to either (i) a comma-separated value file (CSV file) or (ii) slot values of an instance model.

3.7.2.1 Exporting Plots Data to a CSV File

The values plotted in a time series chart are exportable to CSV files.

To export plots data of a time series chart to a CSV file:

1. Click the Export Data ... toolbar button on the time series chart's plot panel (Figure 46).
2. Enter a filename and select a location to save the file.
3. Click the **Save** button.

![Figure 46 -- The Export Data Toolbar Button on a Time Series Chart](image)

**Figure 46** -- The Export Data Toolbar Button on a Time Series Chart

![Figure 47 -- Plotted Values in a Time Series Chart](image)

**Figure 47** -- Plotted Values in a Time Series Chart
If you exported the plotted values in a time series chart (Figure 47) to a CSV file, the exported plots data in the CSV file would look like those shown in Figure 48. The axes labels would appear in the first paragraph of the file as the column header.

### 3.7.2.2 Exporting Plots Data to an Instance Model

Cameo Simulation Toolkit also allows you to export plots data in a time series chart to an Instance model. The model for exported plots data in the Simulation profile comprises of DataSet and Data classes. Whenever you export plots data to an instance model, Cameo Simulation Toolkit will create an InstanceSpecification of a DataSet class. The slot of a data property will contain a list of InstanceSpecifications of the Data class. The first instance on the list is an InstanceSpecification containing the time values while the second one is an InstanceSpecification containing the values that appear on the vertical axis of the first plot. If a time series chart has more than one plot, the third InstanceSpecification on the list will be an InstanceSpecification containing the values of the second plot and so on.

To export plots data to an instance model:

1. Click the **Export to Instance...** toolbar button of the time series chart's panel whose values you want to export (Figure 49). The element selection dialog will open.

![Figure 49 -- The Export to Instance Toolbar Button on a Time Series Chart](image-url)
2. Select a package that will own the instance model you want to create.
3. Click the OK button. Cameo Simulation Toolkit will create the instance model containing the plots data as shown in Figure 50.

![Figure 50 -- Instance Model Containing Plots Data in a Time Series Chart](image1)

### 3.8 Nested UI Configuration Stereotype

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 and displayed as the UI of the system part, which is represented by the Property specified in the feature tag. A Nested UI Configuration can also show images of UI that is defined as the tagged value of config tag definition, such as Panel, activelImage, and time series chart (Figure 51).
You can use a NestedUIConfig stereotype to either (3.8.1) represent a part of an execution context, (this part can nest components that represent the nested properties) or (3.8.2) 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.

### 3.8.1 Representing Parts of an Execution Context

A UI Panel or UI Group Box to which a NestedUIConfig stereotype is applied and a 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/simulation/tests` folder of the Magic-Draw installed directory.

The following System Class Diagram shows the structures of Class System and Class Monitor (Figure 52).
On the same Class Diagram, there are Instance Specifications of Class System and Class Monitor that will be used in the simulation execution (Figure 53).

The Untitled1 User Interface Modeling Diagram displays the UI configuration that will be used in the simulation execution (Figure 54).
Figure 54 -- UI Configuration of Class System

Figure 55 displays the UI configuration and the specification of the UI Panel named panel1, which represents the monitor1 Property, as a part of Class System.

Figure 55 -- UI Configuration and Its Specification

When the test_nested_panels ExecutionConfig is executed, the UI mockup will be displayed. Figure 56 exhibits the UI Panels and UI Group Boxes that represent the parts (Properties) of the Class System and in-depth nested parts as well.
3.8.2 Representing Parts Using Existing UI Configurations

You can also use a NestedUIConfig stereotype to show a part using existing UI configurations. 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 you can reuse existing UI configurations to illustrate any parts of contexts that have the same Class represented. Thus, you do not need to create a new UI configuration whenever you want 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 (FlashLight.mdzip). The following System Definition Class Diagram shows the definition of the FlashLight system (Figure 57). By default, when executing and running this sample, you will see the Button and Light are shown in different Frames. Figure 58 demonstrates the runtime user interface mockup that represents the Class Button and Class Light.
2. Right-click the Containment tree and select **New Diagram > Other Diagrams > User Interface Modeling Diagram** to create a User Interface Modeling Diagram.
3. Create a UI Frame on the UI Modeling Diagram (Figure 59) by dragging the Class System to the newly created UI Frame.
4. Create two UI Panels in the UI Frame.

5. Drag the Light property to one of the UI Panels (Figure 60) 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 61). This will set the dragged UI configuration to the config tag of that UI Panel. Figure 62 shows the tagged value specification of the UI Panel that represents the Light property.
7. Drag the Button property to another UI Panel (Figure 63).
8. Drag an existing UI configuration named PowerButton to the UI Panel that represents the Button property (Figure 64).

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 65).
10. Execute and run the newly created Execution Configuration. The UI mockup will appear as illustrated in Figure 66.
3.9 Reusable UI Mockup

Cameo Simulation Toolkit improves the flexibility in user interface modeling. You can reuse UI components by typing the property using the UI component. The object, which is the value of the property, represents a mock-up of the UI component. Once the object is created, the mock-up UI will be created and shown in the MagicDraw window. Whenever the object is deleted, the mock-up UI will be deleted as well. It is not necessary to add the UI component to the UI tag definition of the Execution Config stereotype.

To create and reuse a UI component:

1. Create a UI frame on a UI Modeling element diagram.
2. Apply a UI stereotype to the created UI frame by right-clicking the created UI frame icon in the Containment tree and click **Stereotype**.
3. Select a **UI** stereotype.
4. Click the **Apply** button (Figure 67).

5. Drag the UI frame icon from the Containment tree to the UI frame in the model in order to make the created UI Frame represent itself (Figure 68).
Create a property in the block. Whenever this property is initialized, the UI Frame will be executed.

Use the UI Frame as a type for the created property.

You can use a UI Frame to create other properties and define it as the type of the properties. A UI mockup will be started and deleted with an object that is the value of the property typed by a UI component (including but not limited to the property in step 6 above).

You can show or hide a UI mockup by using the context menu of the instance of the property.
To hide a UI mockup:

1. Right-click an instance of a property and click **Hide Mock-up** in the **Variable** panel.
2. Click the **Close** button.

To show a hidden UI mockup:

- Right-click an instance of a property and click **Show Mock-up**.

You can also show or hide the other UI components, for example, time series chart and ImageSwitcher.

### 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 you select **true** as the value of the **silent** tag in the execution configurations, Cameo Simulation Toolkit will not animate execution and will execute models in silent mode (see Section 3. Execution Configuration and UI Modeling for more information).

### 4.1 Active, Visited, and Last Visited Elements

Active elements are the elements a simulation session is focusing on (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 will be annotated with green by default (Figure 70). The last visited element will be annotated with orange (by default).
4.2 Customizing Animation Colors

There are four kinds of annotated elements in execution of a model: (i) Active, (ii) Visited, (iii) Breakpoint, and (iv) Last Visited elements. By default, active elements will be annotated with red, visited elements with green, last visited elements with orange, and breakpoints with yellow. Cameo Simulation Toolkit allows you to customize the colors of these 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 71).
3. Customize the colors of the active, visited, last visited elements, or breakpoints.
4.3 Auto Open Diagrams Option

The **Auto Open Diagrams** option in the **Environment Options** dialog and **Simulation** window allows Cameo Simulation Toolkit to open a tab where you can see a diagram that contains a currently active element in a non-silent execution (animation is enabled). If you want to enable this option, you need to select `true` as its value. This is useful when you want to track the simulation current status. Select `false` as the value of **Auto Open Diagrams** when you would like to focus on just one diagram and avoid automatic switching or opening of diagrams.

If the value of **Auto Open Diagrams** is `false` and you run a silent-mode simulation of an activity diagram that has a decision node without any decision input, Cameo Simulation Toolkit will open the diagram and highlight all elements including the decision node and open a **Question** dialog to ask for a decision when the execution reaches the decision node. However, if the value of **Auto Open Diagrams** is false and you are running a non-silent execution, Cameo Simulation Toolkit will open the diagram and highlight only the decision node.

**Note**: The default value of **Auto Open Diagrams** is `false`.

To select the **Auto Open Diagrams** option in the **Environment Options** dialog:

1. On the main menu, click **Options > Environment**. The **Environment Options** dialog will open (Figure 72).
To select the **Auto Open Diagrams** option in the **Simulation** window:

- Click the **Auto Open Diagrams** toggle button in the **Simulation** window (Figure 73).

---

Figure 72 -- The Auto Open Diagram Option in the Environment Options Dialog

2. Click **Simulation** on the left-hand side of the dialog and select the **Auto Open Diagrams** check box.
3. Click **OK**.

---

Figure 73 -- The Auto Open Diagram Option in the Simulation Window
4.4 Opening a Diagram of an Executing Behavior

When you simulate a model, the Sessions pane will open to display execution of each element in the model. During the simulation, if you ever need to see what diagram a behavior being executed has, you can double-click the session in the Sessions pane of the Simulation window. Figure 74 below shows the Sessions pane when Cameo Simulation Toolkit is executing the Blackjack sample (Blackjack.mdzip). If you double-click the behaviors outlined in red, a new tab will open to display the corresponding diagram.

5. Simulation Debugging

5.1 Understanding Simulation Sessions

Cameo Simulation Toolkit creates a simulation session(s) while a model is being executed. A 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 a tree node (Figure 75).
5.2 Simulation Debugging Process

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

<table>
<thead>
<tr>
<th>Button</th>
<th>Name</th>
<th>Shortcut Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run Execution</td>
<td>F8</td>
<td>To start a selected simulation session.</td>
</tr>
<tr>
<td></td>
<td>Suspend</td>
<td>F8</td>
<td>To pause a running simulation session in the <strong>Sessions</strong> tab.</td>
</tr>
<tr>
<td></td>
<td>Resume</td>
<td>F8</td>
<td>To resume a simulation session.</td>
</tr>
<tr>
<td></td>
<td>Step into</td>
<td>F5</td>
<td>To execute and animate a currently active element in a selected simulation session in the <strong>Sessions</strong> tab.</td>
</tr>
<tr>
<td></td>
<td>Step over</td>
<td>F6</td>
<td>To execute a currently active element in a selected simulation session and run animation in the background.</td>
</tr>
<tr>
<td></td>
<td>Terminate</td>
<td>None</td>
<td>To stop a session in the <strong>Sessions</strong> tab. 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 5.4.2 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 Value Monitoring), and **Breakpoints** pane (5.5 Breakpoints).
5.3 Simulation Console

5.3.1 Console Pane

Cameo Simulation Toolkit provides the Simulation Console pane in the Simulation window. The Console pane displays simulation information during model execution (Figure 77) including the date and time the simulation engine starts, the date and time the execution runs and stops, user output, and the location where result values are stored.

The Console pane may contain a hyperlink to a model element in a MagicDraw project. During execution of a model, scripts evaluation failures may happen and thus expression evaluation errors occur. If Cameo Simulation Toolkit cannot evaluate some scripts in an element, it will create a hyperlink in the Console tab to that element in the Containment tree. When you click the link, Cameo Simulation Toolkit will highlight the element in the Containment tree.

Figure 77 shows a hyperlink resulting from errors in evaluating scripts in the Console tab. The link points to the corresponding element in the Containment tree.
Table 4 shows the function of each button in the **Console** pane.

**Table 4 -- Console Pane Buttons**

<table>
<thead>
<tr>
<th>Button</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Clear Console" /></td>
<td>Clear Console</td>
<td>To remove all simulation information displayed in the <strong>Console</strong> tab.</td>
</tr>
<tr>
<td><img src="image" alt="Show Runtime Information" /></td>
<td>Show Runtime Information</td>
<td>To display the runtime information of the Cameo Simulation Toolkit in the <strong>Console</strong> tab. The runtime information consists of the registered execution engines, available scripting engines, and the active simulation sessions (Figure 76).</td>
</tr>
<tr>
<td><img src="image" alt="Show user output only" /></td>
<td>Show user output only</td>
<td>To display only the output generated from <strong>print</strong> commands in scripts in the <strong>Console</strong> tab.</td>
</tr>
</tbody>
</table>
5.3.2 Simulation Information

The Console pane can display six levels of information (sorted in ascending order by priority) as follows:

- **TRACE**: displays all levels of information.
- **DEBUG**: displays debugging information.
- **INFO**: displays normal information.
- **WARN**: displays warnings.
- **ERROR**: 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 tab.

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

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

5.3.3 Simulation Log File

When Cameo Simulation Toolkit is executing a model, the Console tab in the Simulation window will show you the simulation details. However, the Console tab 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 Configuration File directory, as below.

- Windows Vista / 7 / 8: C:/Users/<USERNAME>/AppData/Local/.magicdraw/<md.version.number>
- Windows XP / 2000: C:/Documents and Settings/<USERNAME>/Local Settings/Application Data/.magicdraw/<md.version.number>
- Windows NT4: C:/WINNT/Profiles/<USERNAME>/Local Settings/Application Data/.magicdraw/<md.version.number>
- Other OSs: <user.home>/magicdraw/<md.version.number>

The simulation.log file is an XML file (or a text file) that records all simulation details that have ever been displayed in the Console tab during model execution (see the comment in the simulation.properties file to customize the file).

5.3.4 User Output Filter

You may want to limit the information shown in the Simulation Console tab. If this is the case, you can click the Show user output only icon (Figure 78) and the Console tab will show only the output generated from print commands in scripts.

![Figure 78 -- User Output Filter in the Console Tab of the Simulation Window](image)

5.4 Runtime Value Monitoring

5.4.1 Context, Runtime Object, and Runtime Value

When you are executing a model simulation, Cameo Simulation Toolkit creates the context, runtime objects, and runtime values 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 79, 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 79, 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.

5.4.2 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 79).
The Variables pane (Figure 79) 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. You can also open the (iii) Causality column by clicking the Causality button.

(i) Name column

The Name column represents context and structural features of a model being executed. 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 execution. You can directly edit the runtime values in the Value column if they are of the following types: Boolean, Integer, Real, and String.

(iii) Causality column

If you click the Causality button on the toolbar of the Variables pane, the Causality column will open on the Variables pane. It shows that the value of a property represented in the row is the result of an evaluation (target) or is a given value. You can change the causality of the property if the parametric evaluator supports solving symbolic expressions, for example, MATLAB with the symbolic math toolbox.
### 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><img src="image" alt="Refresh" /></td>
<td>Refresh</td>
<td>To refresh the tree and values in the <strong>Variables</strong> pane.</td>
</tr>
<tr>
<td><img src="image" alt="Export" /></td>
<td>Export to New Instance...</td>
<td>To create a new InstanceSpecification and export a selected runtime object to a newly created Instance Specification.</td>
</tr>
<tr>
<td><img src="image" alt="Export" /></td>
<td>Export to Instance or Export to Instance...</td>
<td>To export a selected runtime object to an InstanceSpecification, which is used to create the runtime object, or to an existing InstanceSpecification (see section 5.4.10 Exporting Runtime Objects to InstanceSpecifications). All of the slot values of the InstanceSpecification will be replaced by the runtime values of the runtime object.</td>
</tr>
<tr>
<td><img src="image" alt="Causality" /></td>
<td>Causality</td>
<td>See 12. Action Languages.</td>
</tr>
</tbody>
</table>

**Note**

While executing a model, you can double-click a running session to open a diagram of that particular session that contains the progress of the simulation (Figure 80).

![Diagram](image)

**Figure 80 -- Double-Clicking a Running Session to Show the Diagram at Run-Time**
5.4.3 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 81).

![Variables pane](image)

**Figure 81 -- The Context Menu Showing the Runtime Value in Time Series Chart**

The **time series chart** shows the runtime value with respect to simulation time as shown in Figure 82.
You can create a new runtime object through the context menu of a property typed by a class or a block in the Variables pane. The context menu (Figure 83) allows you to choose one of the two options: (i) Initialize Object(s) or (ii) Initialize Object(s) Recursively.
To create a new runtime object in the **Variables** pane:

1. Right-click a property typed by a class or a block in the **Variables** pane.
2. Select either (i) **Initialize Object(s)** or (ii) **Initialize Object(s) Recursively**.

If you select the **Initialize Object(s)** option, Cameo Simulation Toolkit will create the object with a default value of the lower multiplicity (Figure 84). The property that is typed by a class or a block will not be created.

If you select the **Initialize Object(s) Recursively** option, the object and its internal structure will be created with a default value of the lower multiplicity (Figure 85).
5.4.5 Creating Runtime Objects from Classifiers

With Cameo Simulation Toolkit, you can use a Classifier with nested parts as the simulation context without the need to create Instance Specifications for those nested parts. Cameo Simulation Toolkit will create runtime objects for those parts automatically. If the type of a property is Data Type, the default value of the runtime value of that property will also be created, depending on the default value of the property's type. In addition, if the type of the part contains a specified Classifier Behavior, and the type itself is set as active, the execution of that behavior will also be started (the autoStartActiveObjects option in the ExecutionConfig must be set as true).

Figure 86 shows a Class Diagram located in the FlashingLight.mdzip sample, demonstrating property light, button and timer as parts of the System Class.

You may replace the ExecutionTarget of the ExecutionConfig named FlashLight with the System Class as illustrated in Figure 87. You can see the result of running this ExecutionConfig in Figure 88. The result shows that
Runtime Objects are automatically created for the parts of the System Class, and the behavior of each part is also automatically started.

![Figure 87 -- Modified ExecutionConfig FlashLight](image1)

![Figure 88 -- Sessions and Variables Panes Showing Runtime Objects with a Classifier](image2)

### 5.4.6 Creating Runtime Objects from InstanceSpecifications

At the beginning of execution of a model, Cameo Simulation Toolkit will create a runtime object to store runtime values. If the element to be executed is an InstanceSpecification or an ExecutionConfig whose executionTarget is InstanceSpecification, the runtime values will be generated from the slot values. These runtime values will later 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 a defined default value, Cameo Simulation Toolkit will generate a runtime value from the default value and assign it to the structural feature of the runtime object.

Figure 89 shows the example of a runtime object that is created for executing the pipe InstanceSpecification. The length slot of the InstanceSpecification contains only one value, which is 1.0, as shown in Figure 89. The runtime value, which is produced for the length structural feature of the runtime object, will equal this slot value (1.0), while the runtime values of radius and thickness will equal the default values of the radius and thickness property of the Pipe class (0.05 and 0.002 respectively).
5.4.6.1 Representation of a Runtime Object

The Variables pane in Cameo Simulation Toolkit displays **Name** and **Value** information of a runtime object that you are executing. If a runtime object is created by an Instance Specification, the name of the Instance Specification will appear in the column **Value** as shown in Figure 90.

You can also see a representation of the element’s name in tooltip as in the column **Value** if you place your mouse over the element being activated on the diagram pane (Figure 91).

You can open a **Message** dialog that contains the runtime object being activated by clicking the element and click the tooltip icon on the tooltip menu (Figure 92).
5.4.6.2 Locating Instance Specification in the Containment Tree

If you used an instance specification to create a runtime object and you want to find that particular instance specification in the Containment tree among the other items, Cameo Simulation Toolkit can locate and highlight it for you. This is particularly useful if you have a long list of items in the Containment tree and it takes some time to locate a specific item. All you need to do is select the Go To command from the context menu of that runtime object in the Variables pane.

**Note**

If a runtime object is not created from an instance specification, the Go To command will be disabled on the context menu.

(i) To locate and highlight an instance specification in the Containment tree through the Variables pane:

1. Right-click a runtime object in the Variables pane and select Go To. The name of the instance specification that you used to create the runtime object will appear on the submenu (Figure 93).

2. Click the instance specification’s name. Cameo Simulation Toolkit will locate and highlight it in the Containment tree for you (Figure 94).
5.4.7 Automatic Initialization of Context and Runtime Object

Cameo Simulation Toolkit is capable of initializing both context and runtime objects automatically even if an execution context does not exist and no object is configured to initiate the runtime objects.

5.4.7.1 Context Initialization

When executing a behavior, Cameo Simulation Toolkit allows you to execute its context as well.

To execute a behavior with its context:

- Right-click a behavior or a behavior's diagram, and then select Simulation > Execute with Context (Figure 95).
To execute a behavior without its context:

- Right-click a behavior or a behavior's diagram, and then select **Simulation > Execute** (Figure 96).
When you execute a behavior with its context, Cameo Simulation Toolkit will initialize the context of the selected behavior and execute the classifier behavior. During the execution of the classifier behavior, if the runtime object of the context classifier is stable, the selected behavior will be executed.

If a selected behavior has no context, the **Execute with Context** menu will be disabled (Figure 97).
5.4.7.2 Runtime Object Initialization

When you use an InstanceSpecification to initialize a run-time object, you will also need to use the corresponding slot values of such InstanceSpecification to initialize the feature values of the run-time object. Cameo Simulation Toolkit will use the default value of the corresponding feature (property) of each slot if the slot value is empty.

The value of each feature (property) will be automatically initialized only if its Lower-value Multiplicity is more than zero or is undefined. Otherwise (the feature’s Lower-value Multiplicity is zero), the feature value will be empty (nothing is initialized).

However, characteristic of the initialization depends on the type of the feature. The following sections explain the types of feature that can influence initialization characteristics.

a) Structural Feature Typed by Primitive Datatype

The default value of a structural feature, which is typed by a primitive datatype (Table 6), is its initialized value. If the default value of the feature is, however, not specified, the initialized value will be the corresponding one in Table 6.
Table 6 -- Value Used for Automatic Initialization of the Structural Feature typed by Primitive Datatype

<table>
<thead>
<tr>
<th>Primitive DataType (Qualified Name)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Boolean</td>
<td>false</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Boolean (SysML Profile.mdzip)</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::boolean</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Integer</td>
<td>0</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Integer (SysML Profile.mdzip)</td>
<td></td>
</tr>
<tr>
<td>PrimitiveValueTypes::UnlimitedNatural (SysML Profile.mdzip)</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::int</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::long</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::short</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Real</td>
<td>0.0</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Real (SysML Profile.mdzip)</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::double</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::float</td>
<td></td>
</tr>
<tr>
<td>Enumeration</td>
<td>first enumeration literal</td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::String</td>
<td>&quot;&quot; (Empty String)</td>
</tr>
<tr>
<td>PrimitiveValueTypes::String (SysML Profile.mdzip)</td>
<td></td>
</tr>
</tbody>
</table>

b) Structural Feature Typed by Non-Datatype

If the type of a structural feature is not a datatype:

- If the structural feature has a non-empty slot value, it will be initialized with such value.
- Otherwise, if the structural feature's default value is specified, it will be initialized with the default value.

If the structural feature also has internal feature(s), for each internal feature:

- If the internal feature has a non-empty slot value, it will be initialized with such value.
- Otherwise, if the internal feature's default value is specified, it will be initialized with the default value.

If an internal feature also has its own internal feature(s), the same rule above will be applied, recursively.

c) Structural Feature Typed by User-defined Datatype

For a structural feature typed by a user-defined datatype:

- If the Datatype has internal structural feature(s), the value of the structural feature will be initialized as described in "b) Structural Feature Typed by Non-Datatype".
- If the Datatype has no internal structural feature:
  - The value of the structural feature will be initialized as described in “a) Structural Feature Typed by Primitive Datatype” if the Datatype is inherited from a primitive datatype.
  - The initialized value of the structural feature will be its default value if the Datatype is not inherited from any primitive datatype. If the default value of the feature is, however, not specified, the initialized value will be empty.
5.4.8 Carrying Values Using Connectors

A connector can carry a runtime value over from one connectable element to another if the following three conditions are present.

(i) Data types on both elements are the same or compatible (equality of value can be defined), or one is the subtype of another.

(ii) If the elements are nested features, the owner of the nested features must exist.

(iii) If the elements are nested properties, in this example, b1 (Figure 98), a connector must be drawn from a3 to the port and from the port to b1 because UML does not have a concrete definition about nested properties. However, SysML does have a definition about nested properties; therefore, the connector can be drawn between nested properties without the help of any ports.

Figure 98 -- Using Port and Connector to Connect Nested Properties

When primitive data types such as n and a1 (Figure 99), are bound together, the runtime values related to the role of both connector ends must be equal (but not necessarily the same instance). If the feature value of a data type value on one end changes, the data type value on the opposite end will change as well. As a result, according to Figure 99, n must equal a1, r must equal a2, s must equal a3, and b must equal a4.

Figure 99 -- Connectors Connecting Primitive Datatypes
5.4.9 Checking Values against Feature Types

Cameo Simulation Toolkit will automatically run a compatibility check every time you assign a value to a feature during simulation. It checks the value against the feature’s type. The value you assign must correspond with the type of the feature, meaning you can assign only values that are compatible with the feature’s type. If you assign an incompatible value to a feature, Cameo Simulation Toolkit will not add it to the feature. Table 7 lists all compatible values for each datatype.

Table 7 -- Compatible Values for Datatypes

<table>
<thead>
<tr>
<th>Primitive DataType (Qualified Name)</th>
<th>Compatible Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Boolean</td>
<td>Strings “true” or “false” (non case sensitive).</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Boolean (SysML Profile.mdzip)</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::boolean</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Integer</td>
<td>Strings that can be cast to numbers (if the string is a real number, the floating point will be eliminated).</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Integer (SysML Profile.mdzip)</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::int</td>
<td></td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Real</td>
<td>Natural number datatypes:</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Real (SysML Profile.mdzip)</td>
<td>(i) UML Standard Profile::UML2 Metamodel::PrimitiveTypes::UnlimitedNatural</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::double</td>
<td>(ii) PrimitiveValueTypes::UnlimitedNatural (SysML Profile.mdzip)</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::float</td>
<td>Real numbers, including the following four datatypes (the floating point will be eliminated):</td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Real</td>
<td>(i) UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Real</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Real (SysML Profile.mdzip)</td>
<td>(ii) PrimitiveValueTypes::Real (SysML Profile.mdzip)</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::double</td>
<td>(iii) UML Standard Profile::MagicDraw Profile::datatypes::double</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::float</td>
<td>(iv) UML Standard Profile::MagicDraw Profile::datatypes::float</td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Real</td>
<td>Strings that can be cast to numbers.</td>
</tr>
<tr>
<td>PrimitiveValueTypes::Real (SysML Profile.mdzip)</td>
<td>Integer numbers including the following five datatypes:</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::double</td>
<td>(i) UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Integer</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::float</td>
<td>(ii) PrimitiveValueTypes::Integer (SysML Profile.mdzip)</td>
</tr>
<tr>
<td>UML Standard Profile::MagicDraw Profile::datatypes::int</td>
<td>(iii) UML Standard Profile::MagicDraw Profile::datatypes::int</td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::UnlimitedNatural</td>
<td>(iv) UML Standard Profile::UML2 Metamodel::PrimitiveTypes::UnlimitedNatural</td>
</tr>
<tr>
<td>PrimitiveValueTypes::UnlimitedNatural (SysML Profile.mdzip)</td>
<td>(v) PrimitiveValueTypes::UnlimitedNatural</td>
</tr>
<tr>
<td>UML Standard Profile::UML2 Metamodel::PrimitiveTypes::UnlimitedNatural</td>
<td></td>
</tr>
</tbody>
</table>
### 5.4.10 Exporting Runtime Objects to InstanceSpecifications

You can export the values of a runtime object through the **Variables** pane to:

(i) a new InstanceSpecification  
(ii) an existing InstanceSpecification.  
(iii) the InstanceSpecification that created the runtime object. Once exported, the values of a runtime object will be set to the slots of the InstanceSpecification.

What type of InstanceSpecification you want to export the values of a runtime object to, depends on what element you used to create the runtime object. If the element is a Classifier, you can export the values of the runtime object to either:

- A new InstanceSpecification  
- An existing InstanceSpecification.

If the element is an InstanceSpecification, you can export the values of the runtime object to either:

- A new InstanceSpecification  
- The same InstanceSpecification that you used create that runtime object.

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

1. Either

<table>
<thead>
<tr>
<th>Primitive DataType (Qualified Name)</th>
<th>Compatible Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>- UML Standard Profile::UML2 Metamodel::PrimitiveTypes::String</td>
<td>Every value is compatible.</td>
</tr>
<tr>
<td>- PrimitiveValueTypes::String (SysML Profile.mdzip)</td>
<td>Strings that can be cast into positive numbers.</td>
</tr>
<tr>
<td>- UML Standard Profile::UML2 Metamodel::PrimitiveTypes::UnlimitedNatural</td>
<td>Real and integer (positive) numbers including the following datatypes:</td>
</tr>
<tr>
<td>- PrimitiveValueTypes::UnlimitedNatural ( SysML Profile.mdzip)</td>
<td>(i) UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Integer</td>
</tr>
<tr>
<td></td>
<td>(ii) PrimitiveValueTypes::Integer (SysML Profile.mdzip)</td>
</tr>
<tr>
<td></td>
<td>(iii) UML Standard Profile::MagicDraw Profile::datatypes::int</td>
</tr>
<tr>
<td></td>
<td>(iv) UML Standard Profile::UML2 Metamodel::PrimitiveTypes::Real</td>
</tr>
<tr>
<td></td>
<td>(v) PrimitiveValueTypes::Real (SysML Profile.mdzip)</td>
</tr>
<tr>
<td></td>
<td>(vi) UML Standard Profile::MagicDraw Profile::datatypes::double</td>
</tr>
<tr>
<td></td>
<td>(vii) UML Standard Profile::MagicDraw Profile::datatypes::float</td>
</tr>
<tr>
<td>- Enumeration</td>
<td>If a value equals to one of the Enumeration Literals, the value is compatible.</td>
</tr>
</tbody>
</table>
(i) Click a runtime object whose value you want to export 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 Value To > New Instance...** (Figure 93).

Either method will open the **Select Owner** dialog (Figure 94).

2. Select the owner of the new InstanceSpecification (the system folder) and click **OK** (Figure 94).
(ii) To export a runtime value to an existing InstanceSpecification:

1. Either

   (i) click a runtime object whose value you want to export in the Name column, and then click the Export to Instance... icon on the Variables pane toolbar, or

   (ii) right-click the row and select Export Value To > Instance... (Figure 102).

Figure 102 -- Exporting a Runtime Value to an Existing Instance through the Context Menu

Either method will open the Select Instance dialog (Figure 103).

| Note       | To export the value of a runtime object using this method, the runtime object must be created from a Classifier. |
Figure 103 -- Selecting an InstanceSpecification in the Select Instance Dialog

2. Select an InstanceSpecification that will be used to save the runtime object (you can select only the InstanceSpecification whose classifier is the same as that of the runtime object) (Figure 103).
3. Click OK.

(iii) To export a runtime value to the InstanceSpecification that has created the runtime object:

1. Either
   (i) Click a runtime object whose value you want to export 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 Value To > Instance (Figure 104).
Figure 104 -- Exporting a Runtime Value to the Creating Instance through the Context Menu

Once Cameo Simulation Toolkit exported and saved the values of the runtime object to the Instance Specification that created it, you can see the hyperlink to that particular Instance Specification in the Simulation Console tab (Figure 105).

2. Click the hyperlink and you will see the exact Instance Specification in the Containment tree.

This method allows you to export the value of the runtime object to the slots of the Instance Specification that has created the object. You can see the notification of each successful export and a location hyperlink, if any, in the Simulation Console tab.

<table>
<thead>
<tr>
<th>Note</th>
<th>To export the value of a runtime object using this method, the runtime object must be created from an Instance Specification.</th>
</tr>
</thead>
</table>

5.5 Breakpoints

Cameo Simulation Toolkit allows you to add or remove breakpoints to or from model elements. A 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 106).

![Figure 106 -- The Breakpoints Pane](image)
<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>
</tbody>
</table>
| Suspend | To suspend a model execution. There are three kinds of execution suspensions: (i) **On Entry**, (ii) **On Exit**, and (iii) **Both**.  
(i) **On Entry**: to suspend an execution when a breakpoint’s element is activated.  
(ii) **On Exit**: to suspend an execution when a breakpoint’s element is deactivated.  
(iii) **Both**: to suspend an execution once a breakpoint’s element is either activated or deactivated. |
5.5.1 Adding Breakpoints

You can add a Breakpoint to a model element using the model's context menu Add Breakpoint(s).

To add a Breakpoint to a model element:

- Right-click a model element either in the containment browser or on a symbol of the model element on a diagram, and then select Simulation > Add Breakpoint(s) (Figure 107).

![Figure 107 -- Adding a Breakpoint](image)

5.5.2 Removing Breakpoints

You can remove a Breakpoint from a model using the model's context menu Remove Breakpoint(s).

To remove a Breakpoint:

- Right-click a model element that has a breakpoint(s) and select Simulation > Remove Breakpoint(s) (Figure 108).
You can also click the **Remove Breakpoint(s)** or **Remove All Breakpoints** toolbar button or select the context menu **Remove Breakpoint(s)** in the **Breakpoints** pane to remove all existing breakpoints (Figure 109).
5.6 Disabling Updates in Simulation Panes

You can click the toggle button in the Simulation window to disable automatic updates of panes in the Simulation window (Figure 110). Turning off auto-updates of panes will cause the execution speed to increase.
6. Validation and Verification

Before executing your UML or SysML model, you need to make sure that it has been correctly modeled or you can use the Cameo Simulation Toolkit’s validation feature to help you validate the 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 111).

   ![Figure 111 -- Model Validation Option in the Environment Options Dialog](image)

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 112).
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 the Harel Statechart. 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. When executing a State Machine, the SCXML engine is capable of finding an initial state automatically even if the initial node is not defined. This feature is also applicable to composite states and orthogonal states.

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

Table 9 below shows most of the supported elements on a State Machine diagram.
7.2 Adapting Models for State Machine Simulation

Currently, Cameo Simulation Toolkit can execute only the elements whose types are specified in Table 9. 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 Triggers on Transitions

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

---

**Table 9 -- 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</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.
7.2.2 Using Guards on Transitions

You can specify guard\(^1\) conditions on transitions using any action language. Open test_guard.mdzip to see an example of how to specify guards on transitions (Figure 114).

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 Activities of a State

States can have defined behaviors at any activity: Entry, Exit, or Do Activity (Figure 115). Cameo Simulation Toolkit will create a new simulation session to execute those defined behaviors. A defined behavior can be an Activity, a State Machine, an Interaction, or an OpaqueBehavior\(^2\). 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.

---

\(^2\) https://developer.mozilla.org/en/JavaScript/Reference
7.2.4 Signal Properties Mapping to Behavior Parameters

Once the activities Entry, Exit, and Do Activity of a state have been specified with a behavior that has input parameters, a signal, which triggers objects to change the state, can carry values to the input parameters.

In order to send the signal to the target object, an instance of the signal will be created. The signal instance can contain the values of its structural features. These values will be propagated to the parameters of the behavior specified at the Entry, Exit, and Do Activity when the following two conditions are met.

(i) The number and order of properties of the signal must be the same as the number and order of parameters in the behavior.

(ii) The type, the order (isOrdered), and the multiplicity of each property of the signal must be the same as the corresponding parameter of the behavior. If the type of signal properties are sub-type of the parameter type, it is considered a match.
If one of the two conditions is not satisfied, a warning will appear once the model is validated by pre-execution constraints before the execution starts.

A send signal action in an activity can create a signal instance with the values of its structural features. For example, the signal MySignal shown in Figure 116. It contains three properties which are: attrib1, attrib2, and attrib3. They are created with Real, String, and Boolean respectively. The order of the three signal attributes is shown in Figure 117.

To create the signal instance of MySignal that contains the values of attrib1, attrib2 and attrib3, a send signal action must be created with three argument pins as shown in Figure 118. The order and types of the pins must match the order and type of the signal properties. So the first pin must be typed by Real, the second by String, and the third by Boolean.
Now that the pins of the send signal action have been created, you can specify the values of the signal instance through them (Figure 119). Figure 119 shows the value specification actions that are used to create the values and added them to the pins of the send signal action.

At this point, the signal instance created by the send signal action in Figure 118, will be sent to the context object of Activity A. The signal instance contains 12.34, Hello World, and true as the values of attrib1, attrib2, and attrib3 respectively. You can get more information on State Machine simulation and Activity simulation in Sections 7. State Machine Simulation and 8. Activity Simulation respectively.
In a state-machine execution, the state of an object will change if it receives a send signal instance. Behaviors that are specified at the entry, do activity, and exit of the state will be invoked in the state transition. The values that come with the signal instance will be delivered to the parameters of the behaviors. For example, prior to receiving a signal, an object is in State1, however, it will move to State2 once it has accepted the signal (Figure 120). The behaviors to which the values of the signal instance will be delivered, are the behaviors specified at Exit of State1, Entry and Do Activity of State2.

In Figure 121 the object has accepted the signal instance, that is MySignal, and it will move from State1 to State2. A performOnEntry activity, which is the entry activity of State 2 will be executed. The performOnEntry activity contains three parameters which are param1, param2, and param3 (Figure 121). They are typed by Real, String, and Boolean respectively. You can see the order of the parameters in Figure 122. The values that come with the signal instance of MySignal will be delivered to these parameters. In this example, the value of attrib1 will be propagated to param1, the value of attrib2 will be propagated to param2, and the value of attrib3 will be propagated to param3. The values of param1, param2, and param3 are “12.34”, “Hello World”, and “true” respectively.
Figure 121 -- Details of performOnEntry Activity

Figure 122 -- Arguments of performOnEntry Activity
7.2.5 State Activation Semantics

Cameo Simulation Toolkit provides **State Activation Semantics** as one of the simulation options. This option allows you to determine whether Cameo Simulation Toolkit activates an Entry behavior before or after activating an entry of state. The option **After Entry** allows activating the entry of state after the Entry behavior is completely activated. The option **Before Entry** allows entering the entry of state before executing the Entry behavior. You can select either option.

To select the **State Activation Semantics** option:

1. Click **Options > Environment** on the MagicDraw main menu to open the **Environment Options** dialog (Figure 123).

2. Select the **Simulation** node on the left-hand side pane and select the simulation option **State Activation Semantics** (Figure 123).

3. Select either **After entry** or **Before entry** from the **State Activation Semantics** combo box (Figure 124).
4. Click **OK**.

### 7.3 Running a State Machine Execution

Cameo Simulation Toolkit will perform a state machine execution if the following elements are selected for the execution:

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

The function of triggers is to change the state of a runtime object during a state machine execution. The trigger can be either a signal or a time event. If it is a signal event trigger, a signal will 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 a 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 125).

![Figure 125 -- The Triggers Drop-down Menu](image)

You can use a User Interface mockup to send a signal to a runtime object. See more information about UI mockup in Section 3.5 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**
7.4.1 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 signals via connectors with or without ports in internal structure, accepting signals and time events, pins, parameters, decisions, structured activity nodes, and many more.
The Activity Execution Engine features include:

- fUML 1.1 specification support.
- Any action languages in opaqueBehaviors, opaqueExpressions\(^3\), decisions, guards, constraints (see 10.10.1 Integration with MATLAB\(^\circledR\) for more details).
- CallBehaviorAction with nested diagrams execution and animation.
- SendSignalAction to send a signal to a global event queue to be consumed by any other engines (such as state machine).
- CallOperationAction through a port.
- Sending signals through a port.
- Support for decision nodes with probabilities over all outgoing edges (see section 11.1.13 Probability).
- Support for decision nodes with a decision input that provides input to guard specifications on outgoing edges from each decision node (Figure 126).

---

\(^3\) https://developer.mozilla.org/en/JavaScript/Reference
<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>You can execute only activities that are owned by a Package or a Class. 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 should contain a value that matches 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 the UML mode.</td>
</tr>
</tbody>
</table>
8.2 Creating a 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 but 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 127).
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 128). Right-click **x** and select **Specification** to open its **Specification** dialog. Select **Integer** as the property type (Figure 129).

![Figure 128 -- Creating a New Property 'x' for the SumPrinter Class](image-url)
(ii) Repeat step (i) to create property $y$ (Figure 130).

![Figure 129 -- Selecting Property Type](image)

**Figure 129 -- Selecting Property Type**

(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".

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.

![Figure 130 -- SumPrinter Class with Properties X and Y of Integer Type](image)

**Figure 130 -- SumPrinter Class with Properties $X$ and $Y$ of Integer Type**
Now that the activity has been created, you need to 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 131 and Figure 132).

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.

---

Figure 131 -- Assigning the Classifier Behavior of **SumPrinter** in the Specification Dialog
(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 tree and select New Element > Opaque Behavior. A new opaque behavior will be created under the Data model.
2. Name it “PrintSumOfIntegers” (Figure 133).
3. Add two input parameters of type Integer: (i) a and (ii) b.

3.1 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 134).
3.2 Repeat step (3.1) to create parameter b (Figure 135).
(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, Python, or Ruby). In this example, JavaScript is 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 136).

---

**Specification of Opaque Behavior properties**

Specify properties of the selected Opaque Behavior in the properties specification table. Choose the Expert or All options from the Properties drop-down list to see more properties.

---

**Figure 136 -- JavaScript for Printing the Summation of Integer Values**

```javascript
print(a+b)
```

Body

Specifies the behavior in one or more languages.
The next step is to complete the PrintSum activity diagram of the SumPrinter class and add a ReadStructuralFeatureAction so that the values of properties \(x\) and \(y\), which are owned by the SumPrinter class, are readable. 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 138).
3. Add the Initial and Activity Final nodes to the activity diagram and connect them to the print action using a control flow (Figure 139).

4. Click **Action** and select the **Any Action...** button from the **Activity Diagram** toolbar on the **PrintSum** activity diagram (Figure 140).

5. Select **ReadStructuralFeatureAction** in the **Select Action Metaclass** dialog and click **OK** (Figure 141).
6. Click the **PrintSum** activity diagram to create the action and name it “readX” (Figure 142).

![Select Action Metaclass Dialog](image1.png)

*Figure 141 -- Selecting ReadStructuralFeatureAction in the Select Action Metaclass Dialog*

7. Open the **Specification** dialog of the action **readX** (Figure 143).

![Activity Diagram with Action readX](image2.png)

*Figure 142 -- The Activity Diagram with Action readX*
8. Click the **Structural Feature** and the “...” button to open the **Select Property** dialog to select the structural feature (Figure 144).
9. Select the property \( x \) of the \texttt{SumPrinter} class and click \texttt{OK}. The \texttt{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.

(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:

10.2.1 Click the Object button and select Input Pin from the context menu (Figure 145). 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 146).
10.2.2 Click the **Result** button and select **Output Pin** from the context menu (Figure 147). Name this pin “a” and select **Integer** as its type, and then click the **Close** button (Figure 148).
11. Click the `readX` action on the activity diagram and select **Display Pins** (the last icon) on the smart manipulator (Figure 149). The **Select Pins** dialog will open (Figure 150).
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 151).
14. Repeat steps 4 to 13 to create another action named readY, which is the ReadStructuralFeatureAction, with the following arrangements:

- The name of the action is "readY".
- The structural feature is the attribute 'y' 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 153).
2. Select ReadSelfAction and click OK.
3. Click the PrintSum activity diagram to create an action and name it, for example, readSelf (Figure 154).

4. Right-click the action readSelf to open its Specification dialog (Figure 155).
5. Select Pins on the left-hand side pane of the dialog and add a new output pin and name it, for example, self of type SumPrinter to the Result row.
6. Click the Back and Close buttons consecutively.
7. 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.
8. Create a Fork Horizontal and use Object Flow to connect it to the pins of the actions readX, readY, and readSelf on the diagrams (Figure 156).
The final step is 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, for example, instance (Figure 157).

3. Right-click the created Instance and open the Specification dialog of instance.
4. Click the Classifier field and the "..." button. The Select Elements dialog will open (Figure 158).
5. Select the class **SumPrinter** 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 159).
7. Click the **Create Value** button to create a new value of the slot (Figure 159). The **Value** box will open (Figure 160).

![Figure 159 -- Creating a Slot Value of x](image)
Figure 160 -- Assigning a Value to Property x Slot

8. Type 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 161), and then click Close.

Figure 161 -- The Created InstanceSpecification with Slot Values in the Containment Tree

The model is now ready for you to execute.
8.3 Executing Activities

You can add some breakpoints to the model created in Section 8.2 Creating a Model for Activity Execution before executing it. This section demonstrates how to suspend execution of the model 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 pins \( a \) and \( b \) of the action \texttt{print}. 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 \texttt{Simulation > Add Breakpoint(s)} (Figure 162). The breakpoints will be shown in the \texttt{Breakpoints} pane of the \texttt{Simulation} window (Figure 163).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure162.png}
\caption{Adding Breakpoints to Pin \( a \) of \texttt{print} Action}
\end{figure}
To open the Simulation window, click Window > Simulation on the main menu (Figure 164).
2. Right-click instance in the containment browser and select Simulation > Execute (Figure 165) to execute the model from instance, which is the InstanceSpecification of the SumPrinter classifier.
Figure 165 -- 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 166). The symbol of the elements with breakpoints attached will be highlighted in yellow by default (Figure 167).
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.

5. Click the Resume Execution button on the Simulation window toolbar to continue the execution (Figure 168).
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 169).

8.4 Duration Analysis

Cameo Simulation Toolkit is capable of calculating the minimum, maximum, and average duration of any activity. It uses the values taken from duration constraints applied to each element in an activity as the basis to calculate the activity’s duration.

Cameo Simulation Toolkit can perform an analysis of activity duration in two scenarios: (8.4.1) duration analysis of visited elements and (8.4.2) duration analysis of executed trace.

8.4.1 Duration Analysis on Visited Elements

You can view the duration of a running activity by using the diagram's context menu **Analyze Duration of Visited Elements**.

---

**Note**

If you do not want to display animation (silent execution), you can create **Execution Configuration** to customize the execution, select **instance** as the **executionTarget**, and set **silent** to **true**. See Section 2.2 Simulation by Executing an Execution Configuration and Section 3. Execution Configuration and UI Modeling for more information.
To see the duration of a visited element:

1. Pause a currently running activity by either (i) adding a breakpoint or (ii) clicking the Pause button.
2. Right-click the diagram and select Simulation > Analyze Duration of Visited Elements (Figure 170).

3. The minimum, maximum, and average duration of the visited elements together with the executed trace listing all of the visited elements will appear in the Simulation Console pane (Figure 171).
Figure 171 -- Duration of Visited Elements in Simulation Console

8.4.2 Duration Analysis on Executed Trace

You can also perform an analysis of activity duration after an activity has been executed. Prior to the analysis, be sure that the execution log has been configured and the value of a Record Activation tag is true (see section 3.2 Execution Log for more information about the execution log).

To see the duration of a visited element:

1. Right-click the execution log and select Simulation > Analyze Duration of Executed Trace (Figure 172). The Select Execution Session and Activity dialog will open (Figure 173).
Figure 172 -- Analyze Duration of Executed Trace Menu on the Execution Log Context Menu

Figure 173 -- Select Execution Session and Activity Dialog

2. Select a session and an activity, and then click the Calculate Duration button. The result will appear in the Simulation Console pane (Figure 174).
8.5 Executing a Call Action without a Target Pin

When Cameo Simulation Toolkit executes a CallOperationAction that does not have a target pin, it will select a runtime object, which is the context of the activity, as a target. For example, a System class, which owns a Sub-System class in Figure 176, is the context of the SystemBehavior activity shown in Figure 175. The CallOperationAction will call the SayHello operation using the activity context, which in this case is the System class.

If an action belongs to an activity partition, and the activity partition represents part of a classifier, which is a context of that activity, Cameo Simulation Toolkit will select the runtime object specified by the part as the target of the action.

However, Cameo Simulation Toolkit will direct the target object to the callOperationAction only if an element represented by the activity partition is a property (or an inherited property) of the classifier, which owns the activity and if there is only one runtime object specified by the property represented by the activity partition. For example, if the CallOperationAction that calls the Print operation is in the activity partition, which represents the SubSystem property of the System class, this will result in a call to invoke the Print operation in the SubSystem property.

![Figure 175 -- Activity SystemBehavior](image)
9. Interaction Simulation

Cameo Simulation Toolkit comes with an interaction engine that can simulate an interaction element based on the UML semantics. An interaction is a unit of behavior that focuses on information exchange among internal properties of a class. You can use it to describe the main behavior of the class by specifying the interaction as a classifier behavior of the class. You can also assign it to be a method of an operation of the class. If an interaction is a classifier behavior, it will be executed whenever the object is started. If it is a method of an operation, it will be executed when the operation is called.

9.1 Supported Elements in Interaction Execution

9.1.1 Lifeline

You may use a lifeline in a sequence diagram to represent a property owned by a class or a block that is the context of the diagram. Sending messages between objects specifying the properties presented by the lifelines can occur. Cameo Simulation Toolkit uses lifelines to find a source and a target object of the message.
You can use a lifeline in a SysML project to represent a nested property of a class or a block, which is the context of a diagram. A path to the nested property appears as a dot notation on the lifeline symbol as shown in Figure 178. Cameo Simulation Toolkit uses the path on the lifeline symbol to find the object.

**Note**
Cameo Simulation Toolkit uses a lifeline with a dot notation to represent a nested property when you record a simulation as a sequence diagram. See 9.4 Recording a Simulation as a Sequence Diagram for more information about recording a simulation.
9.1.2 Message

According to the UML specification, a message defines a particular communication between lifelines of an interaction. Cameo Simulation Toolkit execute messages whose message sorts are `asynchSignal`, `synchCall`, `asynchCall`, and `reply`. You can specify a connector for the message. If the role of one connector end is the property specified by a source object of the message, Cameo Simulation Toolkit will send the message along the connector. A target object will be the object at the other end of the connector.

9.1.2.1 Asynchronous Signal Message (asynchSignal)

Cameo Simulation Toolkit executes an asynchronous signal message by creating a signal object from a signal specified as a signature of the message. It sends the signal object to a target object asynchronously. Figure 179 shows that the bank sent the signal `maintain` to the teller using asynchronous signal message. When the target object (ATM object specifying the teller) received the signal, the state changed to `Maintained`. 
9.1.2.2 Synchronous Call Message (synchCall)

When executing a synchronous call message, Cameo Simulation Toolkit executes a behavior, which is the method of called operation of a target object, thoroughly from the beginning to the end before it can carry out the next message. The execution method is similar to that of the call operation action with “isSynchronous = true”.

You can substitute values for input parameters of the called operation by specifying argument values on a synchronous call message.
Figure 180 -- A Simulation of Synchronous Call and Reply Messages

Figure 180 illustrates the execution of synchronous call messages. A synchronous call message 1 shows the bank object calling the operation `retrieveAccount(accountNumber : String)` of the ledger object. The substitution value for `accountNumber` is the value of `lookupAccountNumber`, which is 0003 in this example. `lookupAccountNumber` is the property of the bank. The simulation looked for the account object with that particular `accountNumber` and returned it to the bank object with a reply message 2 (2 is a reply message of 1). The returned account object would be specified as the value of the `buyerAccount` represented by the lifeline `buyerAccount:Account`. The bank object would then call the operation `getBalance()` of the account object with a synchronous call message 3 that caused the balance value to reply to the bank with a reply message 4 (4 is a reply message of 3). Finally, the bank called itself to print the balance value with the operation `printBalance(balance:Real)`.

9.1.2.3 Asynchronous Call Message (asynchCall)

The execution of asynchronous call message is similar to that of synchronous call message. In this execution, Cameo Simulation Toolkit executes a behavior, which is a method of a called operation of a target object, on a new thread. Cameo Simulation Toolkit will immediately proceed to the next message once the execution of the behavior starts. It will not wait until the execution of the behavior completes. This type of execution is similar to that of a call operation action with "isSynchronous = false".

9.1.2.4 Reply Message (reply)

A Reply message is a message sent as a reply to a call message (synchronous or asynchronous). If there is an argument value specified on the reply message, Cameo Simulation Toolkit will evaluate the argument value. If the result of evaluating the argument is the name of a property owned by a target object, a context object, or a source object, the value returned from the operation will be defined as the value of such property. The reply message 2 in Figure 180 has an opaque expression whose body is "buyerAccount" as an argument value.
buyerAccount is a property of the Balance Lookup class which is the context of a sequence diagram. Therefore, the returned account object from the operation of the ledger object retrieveAccount(accountNumber : String) will be set as a value of the property buyerAccount when the reply message 2 is executed.

9.1.3 Duration Constraint

Cameo Simulation Toolkit allows you to delay delivery of a pair of consecutive messages using a duration constraint. The default time unit of the duration constraint is the millisecond (ms), but you can also use other time units by specifying it after a duration value. For example, you can use 10s to denote 10 seconds. Cameo Simulation Toolkit uses the maximum value of a duration constraint to delay delivery of a message if both the minimum and maximum values are specified.

Figure 181 shows atm sent the first message to consortium to verify a card. The consortium replied the bank to verify the account by sending the second message after 100 milliseconds. The bank then sent the third message to the consortium after 2000 milliseconds (Cameo Simulation Toolkit uses the maximum value to delay delivery of the message). Finally, consortium sent a reply to atm by sending the fourth message 100 milliseconds after it has received the third message.

9.2 Creating Model for Interaction Execution

You can simulate an Interaction element, a (UML or SysML) Sequence Diagram, or a classifier whose classifier behavior is defined by an Interaction. This section demonstrates how to create a simple executable Interaction model through the following steps:

(i) Create a class containing two properties typed by different classes.
(ii) Create an opaque behavior, owned by one of the two properties of the class specified in (i), for printing the summation of two Integer input parameters.
(iii) Write a script to print the summation of the two input parameters.
(iv) Create an operation, owned by the properties specified in (i), and specify the opaque behavior in (ii) as its method.
(v) Create an Interaction as the classifier behavior of the class specified in (i).
(vi) Create call message to call operation specified in (iv) by another property.
(i) To create a class containing two properties typed by different classes:

1. To create a new UML project, click **File > New Project...** on the main menu. The **New Project** dialog will open (Figure 182).

![New Project Dialog](image)

2. Select **UML Project** from the **General-Purpose Modeling** group and specify the project’s name, for example, ‘SimpleSequenceExecution’.
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 tree and select **New Element > Class**. A new class element will be created. Name the created class, for example, ‘System’ (Figure 183).

![Creating a New Class Named System](image)

5. Create two more classes and name them, for example ‘a’ and ‘b’.
6. Right-click the **System** model in the Containment tree and select **New Element > Property**. Name the property, for example, ‘a1’ (Figure 184).
7. Right-click **a1** and select **Specification** dialog. Select **A** as the property type (Figure 185).
8. Repeat step 6 to create property $b_1$ of type B (Figure 186).
(ii) To create an opaque behavior, owned by one of the two properties of the class specified in (i), to print the summation of two Integer input parameters:

1. Right-click the class **B** and select **New Element** > **Opaque Behavior**. Name the opaque behavior, for example, ‘Add’ (Figure 187).

![Figure 187 -- Opaque Behavior Add Owned by Class B under the Data Model](image)

2. Right-click the opaque behavior **Add** and select **New Element** > **Parameter**. Name the parameter, for example, ‘par1’ (Figure 188).

![Figure 188 -- Creating New Parameter ‘par1’ of the Opaque Behavior Add](image)

3. Right-click the parameter **par1** and select **Specification** dialog. Select **Integer** as the parameter type (Figure 189).
4. Repeat step 2 to create another parameter and name it, for example, ‘par2’ of type Integer (Figure 190).

(iii) To write a script to print the summation of the two input parameters:

1. Open the Specification dialog of the opaque behavior Add and write a script in the Body field (you can use any scripting language that is supported by MagicDraw Macro Engine, such as
BeanShell, Groovy, JavaScript, Python, or Ruby). In this example, JavaScript is used:

```javascript
print(par1+par2); to print the summation of the two Integer parameters \texttt{par1} and \texttt{par2}.
```

(iv) To create an operation owned by the properties specified in (i), and to specify the opaque behavior as in (ii) as its method:

1. Right-click \texttt{B} and select \texttt{New Element > Operation}. Name the operation, for example, ‘Add’ (Figure 192).

![Figure 191 -- JavaScript for Printing the Summation of Integer Values](image-url)
2. Right-click the operation `Add` and select **New Element > Parameter**. Name the parameter, for example, ‘par1’ (Figure 193).

3. Right-click the parameter `par1` and select **Specification**. Select **Integer** as the parameter type and **in** as the parameter direction (Figure 194).
4. Repeat steps 2 and 3 to create another parameter. Name it, for example, \texttt{par2} of type \texttt{Integer}, having \texttt{in} as its direction (Figure 195).

5. Right-click the operation \texttt{Add} and select \texttt{Specification}. The \texttt{Specification} dialog will open.
6. Select the opaque behavior **Add** as the **Method** of the operation (Figure 196 and Figure 197).

![Figure 196 -- Selecting Opaque Behavior Add as the Operation Method](image)

![Figure 197 -- An Add Operation whose Method is an Opaque Behavior Add](image)
(v) To create an Interaction as the classifier behavior of the class specified in (i):

1. Right-click System and select New Diagram > Sequence Diagram. Select all properties in the Display Lifelines dialog and click OK (Figure 198).

![Display Lifelines dialog](image)

*Figure 198 -- Selecting All of the Properties in the Display Lifelines Dialog*

2. Right-click the class System in the containment browser and select Specification to open its Specification dialog.

3. Make sure that the interaction System is the classifier behavior of the class System (Figure 199).
(vi) To create a call message to call the operation specified in (iv) by another property:

1. Double-click the interaction **System** to open the sequence diagram containing two lifelines representing \( a_1 \) and \( b_1 \) (Figure 200).

![Sequence Diagram Containing Two Lifelines Representing a1 and b1](image)
2. Select **Call Message** from the **Diagram Modeling Elements** toolbar and create a call message from a1 to b1 (Figure 201).

![Interaction Diagram](image)

**Figure 201 -- Call Message from a1 to b1**

3. Double-click the call message created in step 2 to open the **Specification** dialog, and then select the operation **Add** as the **Signature (operation)** of the call message (Figure 202)
4. Select **Argument** on the left-hand side of the dialog to specify a value of the element. Type, for example, 4 and 5 as the values of parameters **par1** and **par2** respectively (Figure 203 and Figure 204).
9.3 Executing an Interaction Model

This section illustrates the steps to execute the interaction model mentioned in section 8.2 Creating a Model for Activity Execution.
To execute the interaction model in section 8.2 above:

1. Right-click the package **Data** in the containment browser and select **New Element > Package**. Name the package, for example, InstancePackage (Figure 205).

   ![Figure 205 -- Creating Package Named InstancePackage](image)

2. Right-click the **InstancePackage** and select **New Element > Instance Specification**. Name the instance specification, for example, Sys (Figure 206).

   ![Figure 206 -- Creating Instance Specification Named Sys](image)

3. Right-click the instance specification **Sys** and select **Specification** to open the specification dialog.
4. Click the “...” browse button in the **Classifier** row to open the **Select Elements** dialog (Figure 207) and select class **System** as the classifier of the instance specification **Sys** (Figure 208).

![Figure 207 -- The Browse Button in Classifier Row](image-url)
5. Repeat steps 2 and 3 above to create the other instance specifications named \textit{aa1} and \textit{bb1} of the classifiers A and B respectively (Figure 209).

\textit{Figure 209 -- Creating Instance Specifications aa1 and bb1 of the Classifiers A and B Respectively}
6. Right-click the instance specification **Sys** and select **Simulation > Execute** (Figure 210). The **Simulation** window will appear.

7. Click the **Run Execute** button in the **Simulation** window (Figure 211) and see the result in the **Console** tab (Figure 212).

---

**Figure 210 -- Executing the Instance Specification Sys Simulation through the Context Menu**

**Figure 211 -- Running the Simulation of the Instance Specification Sys in the Simulation Window**
Figure 212 -- Simulation Results in the Simulation Console
9.4 Recording a Simulation as a Sequence Diagram

The recording capability of Cameo Simulation Toolkit allows you to:

(i) record created objects as CreateMessages connected between Lifelines that represent the object creator and features of the created object respectively.

(ii) record signals as SendMessages connected between Lifelines that represent signal senders and signal receivers respectively. Connectors will be assigned to the messages if signals are sent via ports or connectors.

(iii) record operation calls as CallMessages connected between Lifelines that represent operation caller and operation owners respectively. Connectors will be assigned to messages if operations are called via ports.

(iv) record changes of states and primitive values as StateInvariants on Lifelines that represent features of objects that own the states or the values.

This section demonstrates how to record signal, state change, operation call, and value change as a sequence diagram during execution of a model. The sample StereoSystem.mdzip, located in the `<md.install.dir>/samples/simulation/` directory, will be used throughout this section.

To record signals sent from and to a runtime object and subsequent state/value changes of the related objects as a sequence diagram:

1. In the Variables pane, select and right-click a runtime object.
2. Click Create Sequence Diagram on the context menu (Figure 213). An empty sequence diagram will be created.

   ![Figure 213 -- The Context Menu to Create an Empty Sequence Diagram](image)

3. Whenever you execute a model (for example, Stereo System as shown in Figure 214), Cameo Simulation Toolkit will:
   
   (i) create the first Lifeline, which represents the selected runtime object.
   
   (ii) record each signal sent from the selected runtime object as a Message in the sequence diagram.
   
   (iii) record each operation call caused by a call message, a CallOperationAction, or a ALH.callOperation with argument and return value as messages in the sequence diagram.
   
   (iv) record an object that receives a signal and(or) an operation call as a Lifeline, unless it exists in the diagram, the object will be called ‘lifeline object’.
   
   (v) record each change in the state of a lifeline object as a Statelnvariant on the Lifeline, with the changed state symbol.
   
   (vi) record each change in the feature value of a lifeline object as a Statelnvariant on the Lifeline. Changes in value are enclosed in constraint brackets, for example, `{a=10}`.
Note

StateInvariants are designated by yellow rounded rectangle symbols. See Figure 214 for examples.

Figure 214 -- The Recorded Sequence Diagram of Stereo System in StereoSystem.mdzip Sample Project

To see what connector a signal or an operation call is sent through:

- Double-click the message or right-click it and select Specification to open the specification window.
To see the values of arguments sent with a signal or an operation call:

1. Either double-click a message or right-click it and select **Specification** to open the specification window.
2. Select **Arguments** in the tree on the left-hand side of the specification dialog to see the value of each argument.

You can customize recorded messages (signals) and lifelines using a **SequenceDiagramGeneratorConfig**.

![SequenceDiagramGeneratorConfig](image_url)

**Figure 215 -- Using SequenceDiagramGeneratorConfig to Customize Recorded Messages (Signals) and Lifelines**

A **SequenceDiagramGeneratorConfig** is a stereotype that is inherited from an **ExecutionListener** stereotype. It contains the following six tag definitions:

(i) **owner**: an element that owns a generated Interaction element. A generated Sequence diagram will be created under that particular Interaction element. You need to select only the element that can own an Interaction element, otherwise a model inconsistency will occur.

(ii) **ignoredSignals**: a list of signals that will be ignored (will not be recorded) during a simulation recording.

(iii) **ignoredLifelines**: a list of elements (objects) that will be ignored (will not be recorded as lifelines) during a simulation recording.

(iv) **recordedObjectPath**: is used to specify an object and display it in a generated sequence diagram.

(v) **recordStateChange**: a boolean option. If true, state changes will be recorded.

(vi) **recordValueChange**: a boolean option. If true, value changes will be recorded.

To customize a sequence diagram recording:

1. Create a class element and apply the **SequenceDiagramGeneratorConfig** stereotype to it.
2. Open the **Specification** dialog of the created class, and then specify the value(s) of the tag definition(s) of the **SequenceDiagramGeneratorConfig** stereotype.
3. Add the created class to the values of the **executionListeners** tag of the **ExecutionConfig** element (see 3.1 ExecutionConfig Stereotype) you are going to execute.
10. Parametric Evaluator

Cameo Simulation Toolkit comes with the parametric evaluator to help you solve constraint expressions in your model. The parametric evaluator is designed to work with the SysML Parametric Diagram. But, you can also use it to solve constraints on any UML classes. With the parametric evaluator, you can define a mathematical or a logical expression as a constraint on a block or a class to limit the values of its properties. If the expression is an equation, the parametric evaluator will evaluate the expression of the constraint and update the values of the properties with the result of the evaluation. If the expression is a logical expression, the parametric evaluator will use the expression to validate the values of the properties.

10.1 Specifying the Language for the Expression

The parametric evaluator only evaluates expressions that are written in the syntax it supports. By default, the parametric evaluator uses the built-in math to solve expressions. The built-in math uses a syntax that is similar to the Octave syntax (see section 10.9 Built-in Math for more information about built-in math). You can also write an expression using a scripting language that is supported by MagicDraw.

You can specify a scripting language to evaluate an expression through the Specification dialog of the opaque expression, which is the specification of the constraint.

To specify the language to evaluate an expression in the Specification dialog:

1. Right-click an opaque expression in the Containment tree and select Specification. The Specification dialog of the opaque expression will open (Figure 216).

![Figure 216 -- Selecting the Language of the Opaque Expression in the Specification Dialog](image)
2. Click the row next to the **Language** option to open the text box and enter the name of the language you want.
3. Click **Close** to close the dialog.

You can also select the language for constraints on a SysML constraint block through the context menu on the constraint block or a constraint property typed by the constraint block (Figure 217).

To select the language for the constraint of a SysML constraint block:

1. Right-click a SysML constraint block or a constraint property typed by the constraint block on the diagram.
2. Click **Language** and select any supported language from the list.

![Figure 217 -- Selecting the Language for Expressions in the Context Menu](image)

The language of the **Default Parametric Evaluator** in Cameo Simulation Toolkit is Built-in Math. Therefore, it will use Built-in Math to evaluate an opaque expression whose language you do not specify. You can see the language options of the **Default Parametric Evaluator** in the **Environment Options** dialog.

To change the language of the **Default Parametric Evaluator** in Cameo Simulation Toolkit:

1. Click **Options > Environment** on the main menu to open the **Environment Options** dialog.
2. Select **Simulation** on the left-hand side of the dialog.
3. Click the row next to the **Default Parametric Evaluator** option to select a parametric evaluator (Figure 218).
10.2 Value Binding

Value binding is the method to maintain values of properties, which are bound together, to be the same. The properties whose values are bound, must be connected together with a connector. The type of the properties which are bound together must be the same or one is a subtype of another. If the type of the properties is a primitive type, you can bind them with either a UML connector that does not have a type, or a SysML binding connector with a «BindingConnector» stereotype applied. If the type of the properties is a class or a block, you can only use a SysML binding connector to tie them.

10.2.1 Primitive Value Binding

Primitive value binding connects two value properties or properties that are typed by a primitive type so that whenever the value of one property changes, the value of the other property will also change. Cameo Simulation Toolkit maintains the values of those connected properties to be the same at both ends. Changing the value at one end causes Cameo Simulation Toolkit to update the value of the property at the other end of the connector. Figure 219 and Figure 220 show the primitive value binding with a SysML binding connector and a...
UML binding connector respectively.

Figure 219 -- Primitive Value Binding with a SysML Binding Connector

Figure 220 -- Primitive Value Binding with a UML Binding Connector
10.2.2 Object Binding

If a binding connector connects properties that is typed by either a class or a block, the runtime value that specifies each property will be the referent of the same block object. Since you can use only a SysML binding connector to bind objects, the SysML plugin is required.

![Figure 221 -- Object Binding with a SysML Binding Connector](image)

10.2.3 Binding in a Complex Aggregate Structure

A binding connector connecting deep nested properties in a block must apply a stereotype «NestedConnectorEnd» at both ends and specify the propertyPath (see section 12.1.11 NestedConnectorEnd for more information about the nested connector end). If there are multiple values specify a property in the propertyPath (the upper bound of the multiplicity of the property is greater than 1 or is infinite), Cameo Simulation Toolkit will construct a value list at each end of the connector and maintain the values in the list. You can use (IsOrdered = true) to order each property in the propertyPath to ensure that the order of values on the list remains the same.

Figure 222 below illustrates an example of binding where the aggregate structure is complex. It shows an executable InstanceSpecification of the block System, which is system: System. It has three instances of subsystems as the values of the slot subsystem:Subsystem[0..*]. In the SysML Parametric diagram, the value property value of the slot subsystem:Subsystem[0..*] are bound to the value property value of the block System. So, the values of the value property value of object System will be [25, 50, 75] respectively.
10.3 Evaluating Expressions

10.3.1 Mathematical Equation

When solving a constraint that is specified by a mathematical equation, Cameo Simulation Toolkit will find a 'target' by evaluating a 'given'. By default, a target is on the left-hand side of the equation and a given on the right-hand side of an equation.

Figure 223 shows an example of constraint parameters target and given in a mathematical equation.
As you can see in the example (Figure 223), the block Circle contains a constraint property typed by the constraint block Circle Area. The constraint block Circle Area has a constraint that is defined by the mathematical expression {area = 3.14159 * (radius ^ 2)}. The target and the given in the equation are the constraint parameters 'area' and 'radius' respectively. These constraint parameters are bound to the value properties of the block Circle. The constraint parameter area of e1 is bound to the value property area of the block Circle, and the constraint parameter radius of e1 is bound to the value property radius of the block Circle. So, the values of the constraint parameters and the value properties bound together are always equal.

If you execute the block Circle, the parametric evaluator in Cameo Simulation Toolkit will create an object Circle and its internal values. It will substitute the value of radius of the block object Circle to the equation and calculate the value of area. Once it obtains the value of area, parametric evaluator will assign it to the area value property of the object Circle. If you change the value of the radius, Cameo Simulation Toolkit will automatically calculate and update the value of the area with the result from evaluating the equation.

You can also define a constraint on a UML class to constrain the values of its properties. Figure 224 shows a constraint on the UML class Circle. The constraint is defined by the same Mathematical expression as in Figure 223 (area = 3.14159 * (radius ^ 2)). If you execute the class Circle, the parametric evaluator will evaluate the value of radius in the equation. It will then use the value resulting from evaluating the value of radius to update the value of property area.
10.3.2 Logical Expression

You can also use a logical expression to define a constraint. A logical expression is an expression that uses comparison operators. Cameo Simulation Toolkit will use the expression to validate values at runtime. If the result of the evaluation is false, it will highlight the values in the Variables pane in red to denote that the values fail the constraint. Otherwise, it will highlight the values in green.

Let’s use the block Circle previously shown in Figure 223. If we modify the expression of the constraint block Circle Area by changing the assignment operator “=” to the equality operator “==” as shown in Figure 225 and Figure 226. The parametric evaluator will use the expression to validate the values of both area and radius. If the result is false, the values will be highlighted in red (Figure 225). But, if the result is true, the values will be highlighted in green (Figure 226).
You can find the constraint object to which the value of the property is bound in the Variables pane.

To see a constraint object to which a value property is bound:

- Right-click a constraint object row in the Variables pane and select Go To. The constraint to which the value of the property is bound will appear (Figure 227).

When you export the object to the instance specification, if the exported object contains a constraint object (an object of a constraint block) and if the expression of the constraint block is a boolean expression, Cameo Simulation Toolkit will set the result of the evaluation, which is a literal boolean, to a slot defined by the constraint property typed by the constraint block.
10.4 Evaluation with Causality

As described in section 10.3.1 Mathematical Equation, the parametric evaluator is capable of solving expressions in the mathematical equation to find the value of the target from the value of the given. We could say that the target is an unknown value that you want to find and the given is a known input value. Normally, a target is a variable on the left-hand side of an equation and a given is a variable on the right-hand side.

Sometimes, however, you know the value of the variable on the left-hand side of an equation and need to find the value of the variable on the right-hand side. You can use Cameo Simulation Toolkit to obtain the given variable if you integrate an external evaluator, which supports solving symbolic equations, to Cameo Simulation Toolkit. MATLAB with Symbolic Math Toolbox (see section 10.10 Integration with External Evaluators for more information about integrating with an external evaluator) is one of the external evaluators you can use.

If the language that defines an expression of a constraint block needs to be solved by an external evaluator that is capable of solving symbolic equations, you can specify what property is the target and the given through the Causality column in the Variables pane.
causality of area and radius will be target and given respectively. However, you can click the drop-down list box in the Causality column to change the causality of area and radius to be given and target. Cameo Simulation Toolkit will then evaluate the value of radius from the given value of area.

The expression of the constraint block Circle Area in Figure 229 shows two roots: (i) a positive value 2.8209 and (ii) a negative value -2.8209. The parametric evaluator needs only one root to evaluate the value of a radius from the given value of an area. Therefore, the Roots selection dialog will open for you to select which root you want as shown in Figure 230.

![Figure 230 -- Roots Selection Dialog](image)

The number of roots varies according to the expression and the multiplicity of a property, which is the target. Therefore, it is possible to select more than one root. The following scenarios show you how to work with multiple roots.

**Scenario 1**

Figure 231 below shows the constraint Test Multiple Root 1 is applied to block A.
Figure 231 -- The Constraint Test Multiple Root 1 is Applied to Block A

**Scenario 2**

The given types of causality of y and x are given and target respectively. Once Cameo Simulation Toolkit finished executing block A and performing a parametric execution to satisfy the constraint Test Multiple Root 1, the equation would result in three values of x and all are the possible roots for the equation. Therefore, the Roots selection dialog would open, allowing you to select one of these three values as the root (Figure 232).

![Roots selection dialog](image)

**Scenario 3**

However, you may select more than one value as the root. But, the number of values you can select cannot be more than the upper multiplicity. Otherwise, an error will occur and an Error message dialog will open. Fig-
Figure 233 shows that two values were selected as the roots when in fact, the upper multiplicity of x value property was one. Therefore, an error occurred and the Error dialog would open.

10.5 Dynamic Constraint

The concept of dynamic constraint is to allow parametric calculation of a Constraint Property with another constraint rather than one from its type (Constraint Block). Such constraint can be dynamically obtained during the execution from typing Constraint Block or one of its subtypes. This section uses the ForwardContractValuation.mdzip sample to demonstrate this concept.

To dynamically apply different constraints to a Constraint Property:

1. Create a Constraint Block with all needed Constraint Parameter(s).
2. Create Constraint Blocks which inherit the Constraint Block created in step 1.
3. Specify the constraint expression of each Constraint Blocks created in step 2.
4. Type the Constraint Property with the Constraint Block created in step 1.
5. Create a behavior, for example State Machine, that will assign the runtime object of different Constraint Blocks created in step 2 to the Constraint Property.

In ForwardContractValuation.mdzip, we will dynamically assign a constraint to the Valuation_Rule Constraint Property, which is typed by the Valuation Constraint Block (having two subtypes: Long Valuation and Short Valuation Constraint Blocks with two different constraint expressions) as in Figure 234. The context of this simulation will be the System Block.
In Figure 235, the state machine is used as the classifier behavior of the System block. The Calculate Contract Value for Long Position and Calculate Contract Value for Short Position states apply different valuations to the System block via its entry activity.

The Calculate Contract Value for Long Position's entry activity assigns the runtime value of the Long Valuation Constraint Block to the Valuation_Rule Constraint Property (Figure 236). Consequently, the values of the constraint parameters will be calculated with the constraint expression specified in Long Valuation. Similarly, Figure 237 demonstrates the Short Valuation dynamic assignment.
Figure 236 -- Entry Activity of Calculate Contract Value for Long Position State

Figure 237 -- Entry Activity of Calculate Contract Value for Short Position State
10.6 Manual Value Updates Using the Parametric Evaluator

Changing values on a property of an object will cause the parametric evaluator to automatically update the other related values with the constraint defined on the object. Or, you may want to update the values with the constraint once you set all necessary values of the properties. The latter is possible by executing your model with the ExecutionConfig whose fireValueChangeEvent = false (see section 3.1 for more information about the ExecutionConfig and its tag definitions) and call the API provided by the parametric evaluator whenever you desire. Figure 238 shows an example of updating values manually with the parametric evaluator.

Signature:

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

Parameter:

Object is an object whose internal values will be updated by the parametric evaluator.

You can call the API with the MagicDraw script engine. For example, you may define it in the body of an opaque behavior. Then, use a call behavior action to call the opaque behavior somewhere in your model.

Figure 238 -- Manual Value Update with Activity in CylinderPipe.mdzip
10.7 Communicating with Evaluators through Simulation Console

You can communicate with a parametric evaluator directly through the command prompt, which is located in the lower part of the Simulation Console pane. You can click the arrow of the language selection drop-down list to the right of the command prompt and select a language you want. Once selected, the language will appear in the command prompt in the Console pane. For example, if you select Matlab, the language in the command prompt will change from “»” to “matlab»” (Figure 239).

![Figure 239 -- Selecting a Language for the Command Prompt](image)

You can enter an expression or a command, which is written in the syntax of the language you selected, in the command prompt and press the Enter key on your keyboard to execute it (Figure 240).

![Figure 240 -- Executing Expression in the Command Prompt](image)

**Note**

You can also use the command prompt in the Simulation Console pane to communicate with the script engine by selecting a scripting language from the language selection drop-down list. For example, selecting JavaScript will cause the language to change to “js>”.

```matlab
matlab>> a=zeros(1, 5)

a =

0 0 0 0
```

```matlab
matlab>>
```
10.8 Exchanging Values between Cameo Simulation Toolkit and the Parametric Evaluator

10.8.1 Exchanging Values between Slot and Mathematical Environment

Cameo Simulation Toolkit allows you to exchange values between a slot and a Parametric Evaluator through the diagram’s **Value Exchange** context menu (Figure 241).

To import a value from the Parametric Evaluator to a slot:

1. Right-click a slot in the Containment tree to which you will export a value and select **Value Exchange > Import Value from Evaluator**. The **Value Exchange** dialog will open (Figure 242).
To export a value from a slot to a Parametric Evaluator:

1. Right-click a slot in the Containment tree whose value you want to export and select Value Exchange > Export Value to Evaluator. The Value Exchange dialog will open.
2. Type a variable name to which you will export the value and click OK.

10.8.2 Exporting Runtime Values to the Parametric Evaluator

During execution of a model, you can export runtime values to the Parametric Evaluator using the context menu of the selected values in the Variables pane (Figure 243). This capability allows you to analyze the exported runtime values using the Parametric Evaluator, for example, a plot.

To export runtime values to the Parametric Evaluator:

1. Right-click the row that contains the runtime values to be exported in the Variables pane and select Export Value To > Parametric Evaluator. The Value Exchange dialog will open (Figure 244).
2. Specify a variable name to which you will export the values and click **OK** (Figure 244).

The runtime values will be exported to the variable of the current default parametric evaluator. You can select another Parametric Evaluator from the **Simulation** option in the **Environment Options** dialog as shown in Figure 233.

For example, if MATLAB® is selected, the runtime value will be exported to the variable of MATLAB® as illustrated in Figure 234.

---

5. MATLAB® is a registered trademark of The MathWorks, Inc.
10.9 Built-in Math

Cameo Simulation Toolkit comes loaded with a built-in Math Solver. As a default Parametric Evaluator, Math Solver can solve simple mathematical and logical expressions. You can use it to:

- Evaluate the mathematical and logical expressions defined in the Constraints of Constraint Blocks for Parametric Simulation on a SysML Parametric diagram.
- Evaluate the mathematical and logical expressions in Simulation Console.

10.9.1 Evaluating Strings from Command Input

You can type generic mathematical equations directly in command input of the Simulation Console when it prompts for the built-in math. For example:

```math
math» x = 10;
math» y = 20;
math» z = x + y
```

\[ z = 30.0000 \] (the calculation result) will be displayed on the simulation console.

Or, if you type, for example, in command input of the Simulation Console:

```math
math» a = true;
math» b = false;
math» c = a & b;
```

If `false` is the result of a calculation, it will be assigned to the variable `c`, but it will not be displayed in the simulation Console tab.
If an expression does not contain any assignment operators, the result will be assigned to the variable ‘ans’. For example:

```math
x = 10;
20 + x
```

`ans = 30.0000` will be displayed in the simulation Console tab.

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

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

`a = 3` will be displayed in the simulation Console tab.

### 10.9.2 Variables

You can use variables (operands) in the built-in Math Solver if they conform to the following naming conventions:

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

### 10.9.3 Values

The valid values that you can use in an expression are as follows:

1. **(10.9.3.1) Real Number**
   - x = 3.14159
   - y = 2

2. **(10.9.3.2) Complex Number**
   - c = 3 + 4i
   - d = 1.25 + 0.25i
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:

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

10.9.3.3 Boolean

a = true
b = false

10.9.3.4 Matrix

U = \[
1.0, 2.0, 3.0; \\
4.0, 5.0, 6.0; \\
7.0, 8.0, 9.0
\]
A = [true; false; false; true]

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 = [1.0, 2.0, 3.0; 4.0, 5.0, 6.0; 7.0, 8.0, 9.0]
A = [true; false; false; true]

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) is 1.0
U(2, 3) 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) is 4.0
U(6) is 8.0
10.9.4 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>

10.9.5 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.

### 10.9.5.1 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>
<tr>
<td>\</td>
<td>Left division</td>
<td>( x%y ) is equivalent to ( (1/x) ) * ( y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( U % V ) (( U ) and ( V ) are ( m ) \times ( n ) matrices) is equivalent to ( (1/U) ) * ( V )</td>
</tr>
<tr>
<td>.*</td>
<td>Element-wise multiplication</td>
<td>( U ).* ( V ) (( U ) and ( V ) are ( m ) \times ( n ) matrices)</td>
</tr>
<tr>
<td>./</td>
<td>Element-wise division</td>
<td>( U ./ \ V ) (( U ) and ( V ) are ( m ) \times ( n ) matrices)</td>
</tr>
<tr>
<td>.\</td>
<td>Element-wise left division</td>
<td>( U .\ V ) (( U ) and ( V ) are ( m ) \times ( n ) matrices) is equivalent to ( (1/U) ) .* ( V )</td>
</tr>
<tr>
<td>.^</td>
<td>Element-wise power</td>
<td>( U .^ \ V ) (( U ) and ( V ) are ( m ) \times ( n ) matrices)</td>
</tr>
</tbody>
</table>

**NOTE**
An Element-wise operator performs an operation on each pair of elements, which is in the same location, of the operand matrixes.
### 10.9.5.2 Assignment Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Name</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>Assignment</td>
<td><code>x=y</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>a=b</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>U=V</code></td>
</tr>
</tbody>
</table>

### 10.9.5.3 Comparison Operators

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

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

\[
A = [1; 2; 3] \\
B = [3; 2; 1] \\
\text{Then} \\
A>B \text{ is } [false \ false \ true];\]
## 10.9.5.4 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];
Then
A&B is [true; false; false; false];

## 10.9.6 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 represents a matrix of values.
- A matrix can be passed to a function that operates element-wise on matrices, as its argument, for example:

X = [1, -2, 3; -4 5 -6; 7 -8 9];
Y = abs(X)
result:
Y = [1 2 3; 4 5 6; 7 8 9]

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>abs(x)</td>
<td>To return an absolute value of x or a complex modulus of c. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>abs(c)</td>
<td></td>
</tr>
<tr>
<td>acos</td>
<td>acos(x)</td>
<td>To return an arc cosine of an angle in the range of 0.0 through pi. All angles are measured in radians. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>acos(c)</td>
<td></td>
</tr>
<tr>
<td>acosd</td>
<td>acosd(x)</td>
<td>To return an inverse cosine of a given value expressed in degrees. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>acosd(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>acosh</td>
<td>acosh(x)</td>
<td>To return an inverse hyperbolic cosine of a given value.</td>
</tr>
<tr>
<td></td>
<td>acosh(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acot</td>
<td>acot(x)</td>
<td>To return an inverse cotangent of a given value.</td>
</tr>
<tr>
<td></td>
<td>acot(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acotd</td>
<td>acotd(x)</td>
<td>To return an inverse cotangent of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>acotd(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acoth</td>
<td>acoth(x)</td>
<td>To return an inverse hyperbolic cotangent of a given value.</td>
</tr>
<tr>
<td></td>
<td>acoth(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acsc</td>
<td>acsc(x)</td>
<td>To return an inverse cosecant of a given value.</td>
</tr>
<tr>
<td></td>
<td>acsc(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acscd</td>
<td>acscd(x)</td>
<td>To return an inverse cosecant of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>acscd(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>acsch</td>
<td>acsch(x)</td>
<td>To return an inverse hyperbolic cosecant of a given value.</td>
</tr>
<tr>
<td></td>
<td>acsch(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asec</td>
<td>asec(x)</td>
<td>To return an inverse secant of a given value.</td>
</tr>
<tr>
<td></td>
<td>asec(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asecd</td>
<td>asecd(x)</td>
<td>To return an inverse secant of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>asecd(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asech</td>
<td>asech(x)</td>
<td>To return an inverse hyperbolic secant of a given value.</td>
</tr>
<tr>
<td></td>
<td>asech(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asin</td>
<td>asin(x)</td>
<td>To return an arc sine of an angle in the range of -pi/2 through pi/2.</td>
</tr>
<tr>
<td></td>
<td>asin(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asind</td>
<td>asind(x)</td>
<td>To return an inverse sine of a given value expressed in degrees.</td>
</tr>
<tr>
<td></td>
<td>asind(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>asinh</td>
<td>asinh(x)</td>
<td>To return an inverse hyperbolic sine of a given value.</td>
</tr>
<tr>
<td></td>
<td>asinh(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>atan</td>
<td>atan(x)</td>
<td>To return an arc tangent of an angle in the range of -pi/2 through pi/2.</td>
</tr>
<tr>
<td></td>
<td>atan(c)</td>
<td>This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>atanh</td>
<td>atanh(x)</td>
<td>To return an inverse hyperbolic tangent of a given value.</td>
</tr>
<tr>
<td></td>
<td>atanh(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>ceil</td>
<td>ceil(x)</td>
<td>To return a 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)</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)</td>
<td>To return a cosine of a given value expressed in degrees. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>cosh</td>
<td>cosh(x)</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)</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)</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)</td>
<td>To return a hyperbolic cotangent of a given value. This function operates element-wise on matrices.</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>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>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>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>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></td>
</tr>
<tr>
<td></td>
<td></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>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>Function Name</td>
<td>Syntax</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</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. If the given matrix is a square matrix, the inverse of a U matrix will be returned using the LU factorization. If the given matrix is not a square matrix, a pseudo inverse matrix will be returned using the QR factorization.</td>
</tr>
<tr>
<td>lin solve</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>ln</td>
<td>ln(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>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>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>log2</td>
<td>log2(x)</td>
<td>To return a logarithm base 2 of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>max</td>
<td>max(x, y)</td>
<td>To return a greater of two given values. max(U) returns the largest element of a given matrix. max(U, V) returns a matrix the same size as U and V with the largest elements taken from U or V. The dimensions of U and V must be the same.</td>
</tr>
<tr>
<td></td>
<td>max(c, d)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max(U)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max(U, V)</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>mean(U)</td>
<td>To return a mean or average value of a given matrix. U is a row or column matrix: mean(U) returns the mean value of all elements in the given matrix. U is a 2-D matrix: mean(U) returns a row matrix that contains the mean value of each column of the given matrix.</td>
</tr>
<tr>
<td>median</td>
<td>median(U)</td>
<td>To return a median value of a given matrix. U is a row or column matrix: median(U) returns the median value of all elements in the given matrix. U is a 2-D matrix: median(U) returns a row matrix that contains the median value of each column of the given matrix.</td>
</tr>
<tr>
<td>min</td>
<td>min(x, y)</td>
<td>To return a smaller of two given values. min(U) returns the smallest element of a given matrix. min(U, V) returns a matrix the same size as U and V with the smallest elements taken from U or V. The dimensions of U and V must be the same.</td>
</tr>
<tr>
<td></td>
<td>min(c, d)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>min(U)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>min(U, V)</td>
<td></td>
</tr>
<tr>
<td>num2str</td>
<td>num2str(x)</td>
<td>To return a string specifying a given number x.</td>
</tr>
<tr>
<td>ones</td>
<td>ones(m, n)</td>
<td>To return an m x n matrix of all 1s.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Syntax</td>
<td>Function</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>pow</td>
<td>pow(x, y)</td>
<td>To return a value of the first argument raised to the power of the second argument. This function operates element-wise on a given matrix U.</td>
</tr>
<tr>
<td></td>
<td>pow(U, c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pow(c, d)</td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>random()</td>
<td>To return a real value with a positive sign, greater than or equal to 0.0 but less than 1.0.</td>
</tr>
<tr>
<td>real</td>
<td>real(c)</td>
<td>To return a real value of the real part of a given complex number. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>rint</td>
<td>rint(x)</td>
<td>To return a value that is closest in value to an argument and is equal to a mathematical integer. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>round</td>
<td>round(x)</td>
<td>To return a closest value to an argument and is equal to a mathematical integer. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td>sec</td>
<td>sec(x)</td>
<td>To return a secant of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>sec(c)</td>
<td></td>
</tr>
<tr>
<td>secd</td>
<td>secd(x)</td>
<td>To return a secant of a given value expressed in degree. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>secd(c)</td>
<td></td>
</tr>
<tr>
<td>sech</td>
<td>sech(x)</td>
<td>To return a hyperbolic secant of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>sech(c)</td>
<td></td>
</tr>
<tr>
<td>sin</td>
<td>sin(x)</td>
<td>To return a trigonometric sine of an angle. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>sin(c)</td>
<td></td>
</tr>
<tr>
<td>sind</td>
<td>sind(x)</td>
<td>To return a sine of a given value, expressed in degree. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>sind(c)</td>
<td></td>
</tr>
<tr>
<td>sinh</td>
<td>sinh(x)</td>
<td>To return a hyperbolic sine of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>sinh(c)</td>
<td></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. 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>size(U, m)</td>
<td></td>
</tr>
</tbody>
</table>

U = [1, 2, 3; 4, 5, 6];
size(U) is [2, 3]
size(U, 1) is 2
size(U, 2) is 3
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<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></td>
</tr>
<tr>
<td></td>
<td>sort(U, 'ascend')</td>
<td></td>
</tr>
<tr>
<td></td>
<td>U is a row or column matrix: sort(U) and sort(U, ascend)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sort all elements in the given matrix.</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrt(x)</td>
<td>To return a correctly rounded positive square root of a double value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>sqrt(c)</td>
<td></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></td>
</tr>
<tr>
<td></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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td>str2num</td>
<td>str2num(s)</td>
<td>To return a number specified by a given string s.</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. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>tan(c)</td>
<td></td>
</tr>
<tr>
<td>tand</td>
<td>tand(x)</td>
<td>To return a tangent of a given value expressed in degree. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>tand(c)</td>
<td></td>
</tr>
<tr>
<td>tanh</td>
<td>tanh(x)</td>
<td>To return a hyperbolic tangent of a given value. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>tanh(c)</td>
<td></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. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>toDegrees(c)</td>
<td></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. This function operates element-wise on matrices.</td>
</tr>
<tr>
<td></td>
<td>toRadians(c)</td>
<td></td>
</tr>
<tr>
<td>transpose</td>
<td>transpose(U)</td>
<td>To return a transposition of a given matrix.</td>
</tr>
<tr>
<td>zeros</td>
<td>zeros(m, n)</td>
<td>To return an m x n matrix of all 0s.</td>
</tr>
</tbody>
</table>
10.10 Integration with External Evaluators

10.10.1 Integration with MATLAB®

You can use MATLAB® to evaluate expressions written in MATLAB, a numerical computing environment and programming language. To do this, you need to specify MATLAB as the language of opaque expressions. You can use any math solver that is external to Cameo Simulation Toolkit like MATLAB® if the integrators are present. Such integrators will be provided in the subsequent release(s) of Cameo Simulation Toolkit.

### 10.10.1.1 Calling MATLAB® from Cameo Simulation Toolkit

You need to first install MATLAB® and set up your system in order to call MATLAB and use it in Cameo Simulation Toolkit.

To use MATLAB® on a 32-bit or a 64-bit version of Microsoft Windows:

1. Install MATLAB®.
2. Press Windows + 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 247). The MATLAB command prompt will open and will 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 248). Click the Advanced tab.

---

6. MATLAB® is a registered trademark of The MathWorks, Inc.
5.2 Click the **Environment Variables** button to open the **Environment Variables** dialog (Figure 249).
5.3 Select Path from the System variables group and click the Edit button to open the Edit System Variable dialog (Figure 249).

5.4 Enter the path to the MATLAB® bin and bin/win32 folders (or bin/win64 for Microsoft Windows 64-bit version) in the Variable value box (Figure 250), for example, 

```
C:\Program Files\MATLAB\R2010b\bin;C:\Program Files\MATLAB\R2010b\bin\win32;
```

5.5 Enter the path to the MATLAB® runtime/win32 (or runtime/win64 for Microsoft Windows 64-bit version) in the Variable value box (Figure 250), for example, 

```
;C:\Program Files\MATLAB\R2010b\runtime\win32;
```

5.6 Click OK.

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

1. Install MATLAB®.
2. Type the following commands in the terminal to show all files in Finder:
   • $ defaults write com.apple.finder AppleShowAllFiles TRUE
   • $ killall 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/maci64:
   /Applications/MATLAB_R2010b.app/runtime/maci64
   ```
   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 in the terminal:
   • $ cd /usr/bin
   • $ ln -s /Applications/MATLAB_R2010b.app/bin/matlab matlab
6. Type the following commands in the terminal to reset Finder:
   • $ defaults write com.apple.finder AppleShowAllFiles FALSE
   • $ killall Finder
7. Restart Mac OS.

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

1. Install MATLAB® (it is assumed that your MATLAB installation directory is /home/username/MATHWORKS_R2011A).
2. Make sure that C Shell has already been installed on your Linux. To install C Shell on Ubuntu, you can type the following command in the 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 in the terminal:
   • ~$ sudo -i
   • ~$ cd /usr/bin
   • ~$ ln -s /home/username/MATHWORKS_R2011A/bin/matlab matlab
   • ~$ export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/home/username/MATHWORKS_R2011A/bin/glnx86
   • ~$ export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/home/username/MATHWORKS_R2011A/sys/os/glnx86
   • ~$ export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/home/username/MATHWORKS_R2011A/bin/glnxa64
   • ~$ export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/home/username/MATHWORKS_R2011A/sys/os/glnxa64
4. Add the MATLAB® bin folder to LD_LIBRARY_PATH of Java.
5. Use a text editor to open the magicdraw file in the bin folder in the MagicDraw installed directory. Type the following text under the line that contains cd "$APP_HOME" and save the magicdraw file (Figure 251):
   • on Linux 32-bit, type:
   ```
   export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/home/username/MATHWORKS_R2011A/bin/glnx86
   ```
   • on Linux 64-bit, type:
   ```
   export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:/home/username/MATHWORKS_R2011A/bin/glnxa64
   ```
10.10.1.2 Selecting MATLAB® as the Parametric Evaluator

You can use MATLAB® to evaluate mathematical expressions by selecting it from the Parametric Evaluator option in the Environment Options dialog.

To use MATLAB® as your parametric evaluator:

1. Select Options > Environment on the main menu bar. The Environment Options dialog will open (Figure 253).
2. Select Simulation on the left-hand side pane.
3. Select MATLAB from the Default Parametric Evaluator drop-down list.
4. Click OK.
10.10.2 Integration with Maple™

Cameo Simulation Toolkit supports Maple™, a mathematical computation engine, to analyze and solve mathematical expressions. Once you have installed Maple™, you can specify it as the language of opaque expressions.

10.10.2.1 Calling Maple™ from Cameo Simulation Toolkit

You need to first install Maple™ on your local machine and set up your system in order to call Maple and use it Cameo Simulation Toolkit.

---

7. Maple™ is a registered trademark of Waterloo Maple, Inc.
To use Maple™ on a 32-bit or a 64-bit version of Microsoft Windows:

1. Install Maple™.
2. Add the path of the Maple™ bin folder to the path environment variable using the following steps:
   2.1 Double-click System in the Control Panel to open the System Properties dialog.
   2.2 Click the Advanced tab (Figure 253).

   ![System Properties dialog]

   Figure 253 -- Setting Environment Variables in the Environment Variables Dialog

   2.3 Click the Environment Variables button to open the Environment Variables dialog (Figure 254).
2.4 Select Path from the System variables group and click the Edit button to open the Edit System Variable dialog (Figure 254).

2.5 Enter the path to the Maple™ bin folder in the Variable value box (Figure 255). There are two methods to enter the path: (i) if the Variable value box has a default value, you need to add the following path at the end of the text: `C:\Program Files\Maple 17\bin.X86_64_WINDOWS` or (ii) if the Variable value box is empty, you can just add the following path `C:\Program Files\Maple 17\bin.X86_64_WINDOWS`.

2.6 Click OK.

1. Install Maple™.

2. Open Maple™ and follow these steps:
   2.1 Type: `kernelopts(mapledir)` into command input to find the maple dir.
   2.2 Type: `kernelopts(bindir)` into command input to find the bin dir.

3. Open the /etc/launch.conf (if this file does not exist, create launch.conf with any text editor) and follow these steps:
   3.1 Type `setenv DYLD_LIBRARY_PATH <BINDIR>` where `<BINDIR>` is the bin dir from step 2.2
   3.2 Type `setenv MAPLE <MAPLEDIR>` where `<MAPLEDIR>` is the maple dir from step 2.1
   3.3 Save launch.conf.

4. Restart the computer.

10.10.2.2 Selecting Maple™ as the Parametric Evaluator

You can use Maple™ to evaluate mathematical expressions by selecting it from the Parametric Evaluator option in the Environment Options dialog.

To use Maple™ as your parametric evaluator:

1. Select Options > Environment on the main menu bar. The Environment Options dialog will open (Figure 256).
2. Select Simulation on the left-hand side pane.
3. Select Maple from the Default Parametric Evaluator drop-down list.
4. Click OK.
10.10.3 Integration with Mathematica®

Cameo Simulation Toolkit supports Mathematica®, a mathematical computation engine, to analyze and solve mathematical expressions. Once you have installed Mathematica®, you can specify it as the language of opaque expressions.

10.10.3.1 Calling Mathematica® from Cameo Simulation Toolkit

You need to first install Mathematica® on your local machine, and then set up your system to allow Cameo Simulation Toolkit to use the installed Mathematica®.

To use Mathematica® on a 32-bit or a 64-bit version of Microsoft Windows and Linux:

1. Select **Options > Environment** on the main menu. The **Environment Options** dialog will open (Figure 257).
2. Select **Simulation** on the left-hand side pane.

---

8. Mathematica® is a registered trademark of Wolfram Research, Inc.
3. Set the **Local Mathematica Directory** field to the location of the directory to which your Mathematica® was installed.

---

**Figure 257 -- Setting Local Mathematica® Directory in the Environment Options Dialog**

To use Mathematica® on Mac OSX:

1. Install Mathematica® and Cameo Simulation Toolkit.
2. Restart MagicDraw twice. It is now ready for you to use.

### 10.10.3.2 Selecting Mathematica® as the Parametric Evaluator

You can use Mathematica® to evaluate mathematical expressions by selecting it from the **Parametric Evaluator** option in the **Environment Options** dialog.

To use Mathematica® as your parametric evaluator:

1. Select **Options > Environment** on the main menu bar. The Environment Options dialog will open (Figure 258).
2. Select **Simulation** on the left-hand side pane.
3. Select **Mathematica** from the Default Parametric Evaluator drop-down list.
4. Click **OK**.
10.11 Trade Study with Cameo Simulation Toolkit

Trade study analysis or a trade-off study is one of the important analyses that can help system engineers to select a configuration that best fits specific criteria among a set of proposed viable configurations.

With Cameo Simulation Toolkit, you can conduct a trade study analysis as shown in the sample project Trade-Study for Brayton Cycle.mdzip, which comes bundled with Cameo Simulation Toolkit. This sample demonstrates the use of Cameo Simulation Toolkit to perform a trade study analysis of compressors selection process for a Gas-Turbine plant operating using the air-standard Brayton cycle. This sample focuses on the thermodynamic efficiency of the plant. The trade study analysis is to select one among the five compressors (A, B, C, D, and E) the thermodynamic efficiency level of which is highest for the plant.

In this sample, the Efficiency Analysis block represents a context that analyzes the thermodynamic efficiency of the plant. If an object of the Efficiency Analysis block is executed with a specific plant configuration (Figure 260), Cameo Simulation Toolkit can evaluate the thermodynamic efficiency of the plant or the efficiency value property of the Efficiency Analysis block by using the parametric model defined in the SysML Parametric diagram of the Efficiency Analysis block. To perform a trade study, multiple objects of Efficiency Analysis must be run. Each of them has a plant object (InstanceSpecification) with a different compressor configuration.

To represent a trade study context, you must create a new class or block, in this example, the Trade-Study block (Figure 259). The Trade-Study block has a property, the analysis property in this sample, typed by the Efficiency Analysis block. As this trade study is intended to run 5 configurations, 5 analyses should be done.
Consequently, the upper value of the multiplicity of the analysis property should be * (for example, [1..*]). It enables the object of the Trade-Study block to contain multiple objects of the Efficiency Analysis block, as a set of configurations to perform the trade study.

To prepare a set of viable configurations for the trade study, InstanceSpecification elements of the Efficiency Analysis are created for each plant configuration as shown in Figure 260.
To create a Trade Study, the `tradeStudy` InstanceSpecification of the `Trade-Study` block is made and its `analysis` property’s slot value is filled with the InstanceSpecifications of the `Efficiency Analysis` block as shown in Figure 261.
Once the available execution config in this sample has been executed (the `tradeStudy` object/InstanceSpecification is run), the result of the trade study (the maximum thermodynamic efficiency) can be found in the `maxEfficiency` value property (Figure 261). In this sample, the fourth configuration is the most efficient one.

![Variables](image)

Figure 262 -- Trade Study Result

10.12 Sample Projects

The Parametric Simulation sample projects are available in the `<md.install.dir>/samples/simulation/Parametrics` directory. The SysML Parametric diagrams and InstanceSpecifications are as follows:

(i) `simple_parametrics.mdzip`, `TradeTransformModel.mdzip`, and `Financial.mdzip`: basic Parametric Simulation.

(ii) The `CylinderPipe.mdzip` sample: demonstrates how to deal with multiple values. It shows the calculation for the cost of raw materials that will be used to manufacture the cylinder pipes. It also demonstrates the use of `OpaqueBehaviorAction` to execute the parametric.

(iii) The `ActParIntegrate.mdzip` sample: demonstrates the use of `OpaqueBehavior` to execute the parametric.

(iv) `SCARA manipulator.mdzip`: demonstrate the use of Parametric Simulation to evaluate the position of end-effector of the SCARA manipulator from the given angles of actuators.

(v) `MotionAnalysis.mdzip` and `SpringDisplacementUsingTimevariable.mdzip`: show how to use Time Series Chart for plotting the runtime values.
11. Simulation of SysML Models

11.1 Supported SysML Elements

The following is a list of SysML elements that Cameo Simulation Toolkit supports.

11.1.1 AcceptChangeStructuralFeatureEventAction

An accept change structural feature event action is an accept event action that waits for a change structural feature event. The event will be sent to an object when the value of a specified structural feature on the event changes. When the accept change structural feature event action is activated, the execution will stay on the action until it accepts the change structural feature event that will be sent once the value of the specified structural feature changes. Figure 263 shows the accept change structural feature event action that waits for the change structural feature event of value property, in this example, \textit{time}.

![Figure 263 -- An Accept Change Structural Feature Event Action in an Activity Diagram](image)

When the value of the value property \textit{time} changes, the accept change structural feature event action will be activated. The token that flows through the pin \textit{before} is the old value of the value property \textit{time} and the token that flows to the pin \textit{after} is the current value of the value property \textit{time}. Cameo Simulation Toolkit will then execute the next action.
11.1.2 AdjunctProperty

Adjunct Property is a property to which the stereotype «AdjunctProperty» is applied. Its tag definition principal: Element[1] is for an element to determine the value of an adjunct property. With regard to the SysML Specification, a principal can be a Connector, a CallAction, an ObjectNode, a Variable, a Parameter, a Submachine State, or an InteractionUse. But, Cameo Simulation Toolkit supports only the adjunct property whose principal is a CallBehaviorAction, a CallOperationAction, an ActivityParameterNode, or a Parameter.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>• CallBehaviorAction and CallOperationAction are subtypes of CallAction.</td>
</tr>
<tr>
<td>• ActivityParameterNode is a subtype of ObjectNode.</td>
</tr>
</tbody>
</table>

Figure 264 shows an adjunct property in the Integrator activity. The adjunct property is typed by the activity TimeDifferent. Besides having the Integrator activity as a classifier behavior, the block Integrator also contains a classifier behavior property typed by the Integrator activity (see section 12.1.6 for more information about classifier behavior property). When the Integrator object is initialized from the block Integrator, the behavior execution of its classifier behavior will start and be set as a value of the classifier behavior property. While executing the classifier behavior, Cameo Simulation Toolkit will activate the call behavior action getTimeDiff, which is the principal of the adjunct property. It will create the behavior execution of the getTimeDiff and set it as the value of the adjunct property in the Variables pane.

11.1.3 BindingConnector

A binding connector is a connector to which a stereotype «BindingConnector» is applied. According to the SysML specification, a binding connector specifies that properties at both ends of a connector have equal values. See section 10.2 Value Binding for more information about value binding with a binding connector.
11.1.4 Block

A SysML block is a UML class stereotyped with «Block». Cameo Simulation Toolkit executes a SysML block the same way it executes a UML Class.

11.1.5 BoundReference

According to the SysML specification, a bound reference must have a binding connector to a property or a nested property of an owning block. Therefore, Cameo Simulation Toolkit will set the value of the bound reference so that it works with the properties connected with the binding connector. In exceptional circumstances, for example, if a classifier that types a bound reference is a subtype of a classifier that types a property at the other end of a binding connector, Cameo Simulation Toolkit will use the classifier that types the bound reference to initialize the object. You can see more information about the execution of binding connector in section 11.1.3 BindingConnector.

![Variables](image)

Figure 265 -- Executing a Block Vehicle that Contains Bound References

11.1.6 ChangeStructuralFeatureEvent

A change structural feature event is an event that occurs when a value of a specified structural feature on the event changes. The change structural feature event will be accepted by a accept change structural feature event action. You can find more information about execution of an accept change structural feature event action and a change structural feature event in section 11.1.1 AcceptChangeStructuralFeatureEventAction.
11.1.7 ClassifierBehaviorProperty

A classifier behavior property is a property to which the stereotype «ClassifierBehaviorProperty» is applied. The value of a classifier behavior property is a behavior execution of the classifier behavior of an object. Therefore, the value of the classifier behavior property exists only after the behavior of the object has been started. See Figure 264 above for example. The block Integrator has a classifier behavior property typed by the activity Integrator. You will see the behavior execution as the value of the classifier behavior in the Variables pane.

11.1.8 ConstraintBlock

A constraint block is a subtype of a block. It is a class stereotyped with «ConstraintBlock». It has a constraint with an expression to constrain the values of its constraint parameters. If an object initialized from a constraint block and a value bound to a constraint parameter of that object is changed, Cameo Simulation Toolkit will evaluate the expression of the constraint. You can see more information about how Cameo Simulation Toolkit evaluates an expression in section 10.3 Evaluating Expressions.

11.1.9 FlowProperty

A flow property is a property to which the stereotype «FlowProperty» is applied. A flow property has a flow direction. A flow due to flow properties will incur flow properties matching. Matching flow properties must have the same direction and type. If the direction and type of multiple flow properties at either end of a connector match, a flow will occur between the flow properties with the same name. When the flow occurs between matching flow properties, the value of the flow property with a direction 'out' or 'inout' will be delivered to the flow property with the direction 'in' or 'inout'.

11.1.10 FullPort

A Full port is a port stereotyped with «FullPort». It specifies a separate element of an owning block. Initializing the owning block will cause Cameo Simulation Toolkit to initialize a port object from a classifier that types the full port.
11.1.11 InvocationOnNestedPortAction

Cameo Simulation Toolkit will use the tagged value onNestedPort to send a signal if a send signal action is stereotyped with «InvocationOnNestedPortAction». 
11.1.12 NestedConnectorEnd

If the ends of a connector that connects properties are stereotyped with «NestedConnectorEnd», Cameo Simulation Toolkit will use the information from the propertyPath of the nested connector end to find the right objects that specify the properties at both ends of the connector. Therefore, you can send a signal along the connector to a deep nesting object. You can also use the binding connector to bind objects that specify the properties at different nesting levels.

11.1.13 Probability

Probability is a stereotype in SysML. You can apply it to outgoing edges of decision nodes and object nodes. When Cameo Simulation Toolkit executes a decision node or an object nodes whose outgoing edges stereotyped with probability, it will use the probability values collected from all outgoing edges to select one outgoing edge that it will go to.
Normally, the summation of the probability values from all outgoing edges should be 1.0 (100%). If this is not the case, Cameo Simulation Toolkit will scale the probability values with the summation.

### 11.1.14 Proxy Port

A Proxy port is a port stereotyped with «ProxyPort». When Cameo Simulation Toolkit executes an object that has a proxy port, the value that specifies the proxy port will be the referent of the object that is the target of the flow. A classifier of the target object must be inherited from an interface block that types a proxy port.

(i) If the proxy port is not connected by a delegation connector to the internal structure of the owning object, the target will be the object itself.
(ii) If the proxy port is connected by a delegation connector to the internal structure of the object owning the port, the target of the flow will be the object specifying the role at the other end of the delegation connector.
If Cameo Simulation Toolkit cannot find the target object, or the classifier of the target object does not inherit from the interface block that types the proxy port, Cameo Simulation Toolkit will initialize the port object directly from its type.

11.1.15 TriggerOnNestedPort

When you model a state-machine in your system whose transition has a trigger stereotyped with «TriggerOnNestedPort», Cameo Simulation Toolkit will use the tagged value onNestedPort to check the port and find which object receives the triggering events.

11.1.16 ValueType

SysML allows you to define a value type for typing value properties in your model. A value type is a data type stereotyped with «ValueType». You can specify the quantity kind and the unit of the value type with the quantity kinds and units defined in the QUDV library. However, Cameo Simulation Toolkit supports only the value properties typed by the SysML primitive value types or their subtypes. If you want to have an executable SysML model, you have to make your custom value types inherited from the SysML primitive value types, including Complex, Real, Integer, Boolean, and String.
Figure 271 -- Value Types Inherited from Real

Figure 272 -- A SysML Model Using the SysML Value Types Defined
11.2 Requirements Traceability from the Variables Pane

When you model with SysML, you can create a relationship between a SysML requirement and any element in your model. Cameo Simulation Toolkit allows you to navigate from the runtime value or from the object, to the related SysML requirement during execution of the model.

11.2.1 Requirement Refined by a Constraint Block

If you use a constraint block, which refines a SysML requirement, to type a constraint property of a block, you can select the object of the constraint property to navigate into a SysML requirement that is related to that object in the Containment tree.

To navigate to a SysML requirement related to a constraint property in the Containment tree:

1. Right click a constraint property object in the Variables pane and select Go To (Figure 273).
2. Select the name of a requirement that appears on the sub menu. Cameo Simulation Toolkit will show you the SysML requirement that is related to the constraint property specified by the selected object in the Containment tree.

Figure 273 -- Navigating into a SysML Requirement related to a Constraint Property

Figure 273 shows the block Circle whose constraint property typed by the constraint block Maximum Area, which refines the requirement Maximum Area. It also shows you that if you right-click the object that specifies the constraint property when Cameo Simulation Toolkit is executing the block Circle, and select the Go To context menu, you will see the requirement Maximum Area. Selecting the requirement on the menu will highlight that requirement in the Containment tree.
11.2.2 Requirement Satisfied by a Property

You can also select a value that specifies a property in the Variables pane to navigate into a SysML requirement, which is satisfied by the property, in the Containment tree. When you right-click the value and select the Go To context menu, you can see a specific requirement satisfied by the property (Figure 274).

12. Action Languages

UML and SysML allow you to define an action or a behavior using an expression or a scripting language, for example, opaque action, opaque behavior, and function behavior. Cameo Simulation Toolkit can execute the script or the expression in your model during the model execution, when an element that contains the script is activated.

Normally, a script will be defined in an opaque expression. Therefore, if your model has any value specifications (like guards, constraints, decisions, default values, and opaque behaviors) that can be defined with opaque expressions, they will be executed during the model execution.
Some special cases, however, apply for activity edges that do not have any guard expression and decision nodes that do not have any Decision Input specified. The name of the activity edge and the decision node will be evaluated by Cameo Simulation Toolkit using a default scripting language as shown in (Figure 275).

**Figure 275 -- Action language in Opaque Action and Evaluation of Decision Node Name**

You can use different languages for different opaque expressions in your model. Cameo Simulation Toolkit will use a scripting engine that matches the specified language of the opaque expression to execute the script content.

### 12.1 Supported Scripting Languages

Cameo Simulation Toolkit supports any scripting language that conforms to the standard JRS-223 such as BeanShell, Groovy, JavaScript, Python, and Ruby. These scripting languages are included in MagicDraw and other MagicDraw re-branded products, for example Cameo Systems Modeler. You can also download and install other JSR-223 compatible language implementations (see http://scripting.dev.java.net/). Cameo Simulation Toolkit supports the following scripting languages:

- Beanshell
- Groovy
- JavaScript
- Python (Jython)
- Ruby (Jruby)
- OCL
- Java binaries
- Math (see Section 1.2 Key Features)

### 12.2 Importing External Libraries

When you use a scripting language in your model, you may need to call some external libraries to do some specific tasks. Therefore, you need to add the external libraries to your project before executing it.
To add an external library to a project:

1. Click **Options > Project** on the main menu to open the **Project Options** dialog (Figure 276).

2. Select **General project options** on the left-hand side of the dialog and select **External Libraries** from the **Simulation Script Engine** option.

3. Click the "..." button of **External Libraries** to open the **Select Files and/or Directories** dialog (Figure 277).
4. Click the **Add** button, to open the **Open** file dialog.
5. Browse the file that you want to add to your project and click the Open button. The path filename will be added to the Select Files and/or Directories dialog.