SimBionic®

Version 3.0

Developer's Manual

Stottler Henke

Smarter Software Solutions

Stottler Henke Associates, Inc.

1650 S. Amphlett Blvd., Suite 300

San Mateo, CA 94402

Tel: 650.931.2700

Fax: 650.931.2701

Web: www.stottlerhenke.com

Email: info@stottlerhenke.com

11/22/2016

Table of Contents

1. Overview

SimBionic® software makes it possible to specify real-time intelligent software agents quickly, visually, and intuitively by drawing and configuring behavioral transition networks (**BTNs**). Each BTN is a network of nodes connected by connectors (links), similar to a flow chart or finite state machine (FSM). Because BTNs show the agent's logic graphically, they are easier to build, understand, and test. Visual logic also makes it easier to show, discuss, and verify the behaviors with members of the development team, subject matter experts, and other stakeholders. BTNs are especially effective for developing real-time reactive planners or adaptive execution systems which detect, track, and classify complex sequences of events and state conditions and then take appropriate actions.
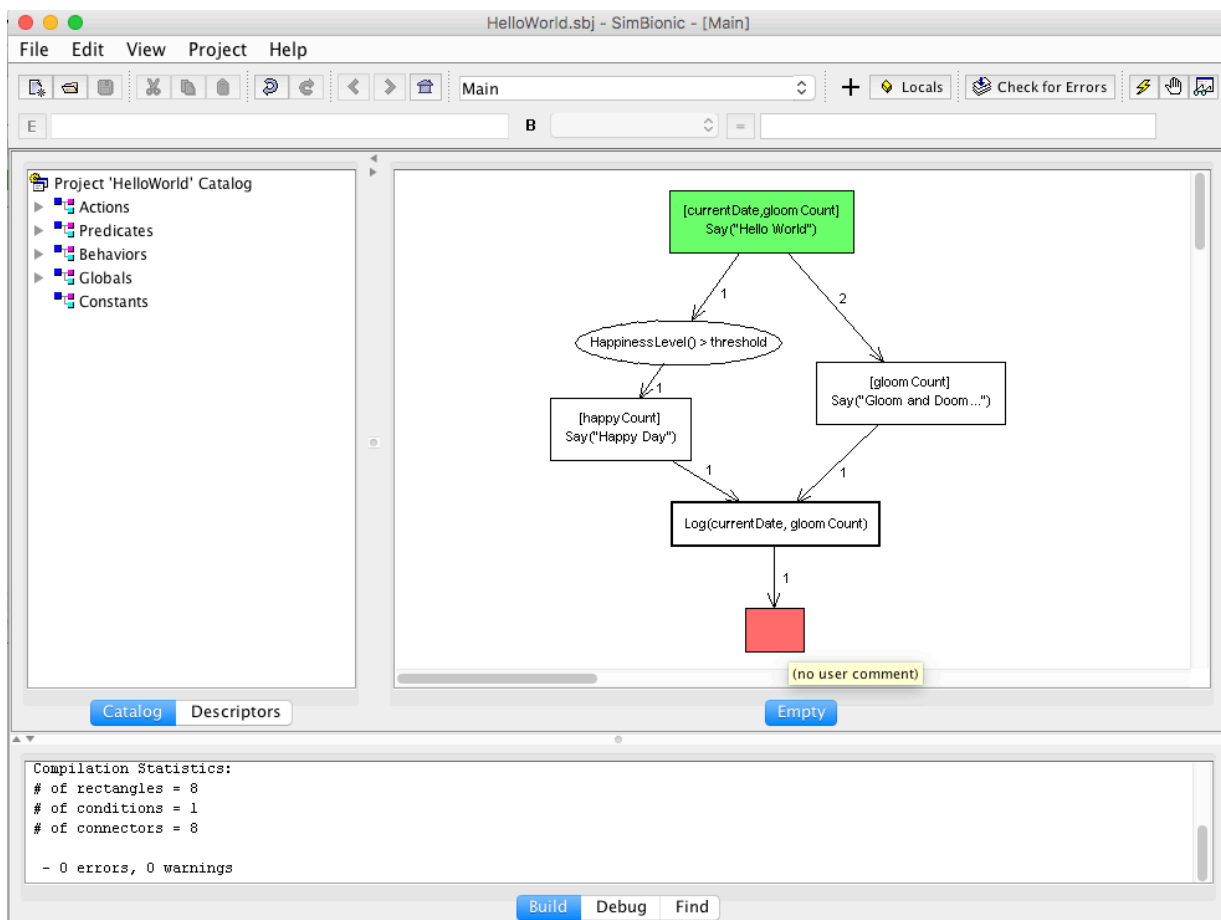
Figure 1. The SimBionic Editor Enables Visual Specification of Intelligent Agent Behaviors.

BTNs provide additional features which make them much more powerful than conventional flow charts or FSMs. For example, BTNs can access local and global variables and call Javascript functions and Java methods, using the JavaScript engine embedded within the Java run-time system. BTNs are hierarchical: a node within a BTN can also invoke other BTNs. Hierarchical BTNs simplify the logic of higher-level BTNs by encapsulating details within lower-level BTNs. BTNs can read and write from/to message queues and blackboards to enable agents to cooperate

and share information. Support for exception handlers makes it possible to develop agent behaviors which handle unexpected events and situations cleanly and gracefully.

A SimBionic **entity** is a thread of execution. The SimBionic engine can contain one or more entities, each executing its own set of BTNs simultaneously. When using SimBionic to develop a game or simulation, different entities might correspond to simulated characters or computer-generated forces, each acting independently. In a decision support system or intelligent training system, each entity could be an expert object that uses its unique knowledge to detect, track, and classify certain types of events or situations, select and present relevant data, and/or make recommendations. BTNs are compact and execute quickly and efficiently, so many entities can run in parallel.

SimBionic software provides three components. The **SimBionic Visual IDE** (or **Editor**) application enables developers to specify intelligent agent behaviors by creating and saving BTNs that are read and executed by the **SimBionic Run-time System.** This run-time software library can be embedded within a Java® technology software application to query for state information and execute actions, as specified by the BTNs. The **SimBionic Debugger** application helps developers test and debug behavior logic by stepping through the execution of the BTNs and inspecting the values of local and global variables.
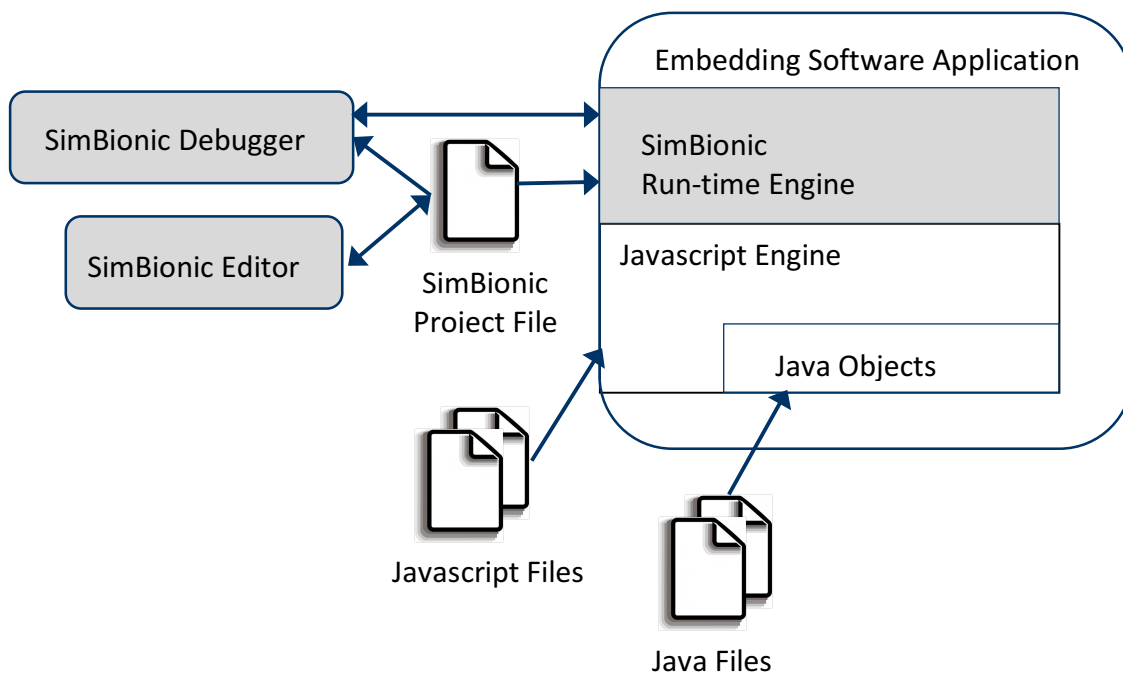
Figure 2. Data Flow Among Software Applications and Modules. SimBionic Modules are Shaded.

2. Getting Started

2.1 Platform and Configuration Requirements

SimBionic runs on computers running Windows 7, Windows 8, Windows 10, and macOS, configured as follows:

Java 8	Available from http://www.java.com
RAM	256 MB
Free disk	20 MB
Operating System	Windows 7, Windows 8, Windows 10, Mac OS X 10.9+

2.2 Installing SimBionic

SimBionic software is delivered as a zip file. Unzip the file to install. The following files and subfolders will be installed in the installation directory:

coreActionsPredicates\	Predefined actions and predicates
javadoc\	Developer documentation
lib\	Third-party libraries required to run the editor and their corresponding license files
samples\	Various sample behaviors to review
test\	Java code that will run the samples
Run SB Editor.bat	Launcher for the editor in Windows
SimBionic-3.x.x.jar	Deployment jar file
SimBionic-dev-3.x.xjar	Development jar file

2.3 Uninstalling SimBionic

To remove SimBionic, move the folder containing SimBionic to the trash.

2.4 Third Party Software

JUnit (junit.org) is required to run the examples in the test\ directory. The Eclipse Integrated Development Environment (eclipse.org) is recommended as well.

The SimBionic Authoring tool depends upon the RSyntaxTextArea (https://github.com/bobbylight/RSyntaxTextArea), released under the modified BSD license. The license file is included with the zip distribution.

2.5 Adding SimBionic to an Eclipse Java Project

1. Place the two jar files in Eclipse project's lib/ folder. Add the 'dev' version of the jar file to the build path.

2. Add the coreActionsPredicates/ folder to the project at the top level.

3. Create a launch configuration to launch the SimBionic editor from within the project. As shown in Figure 4, the main class should be:

   com.stottlerhenke.simbionic.editor.gui.SimBionicFrame.
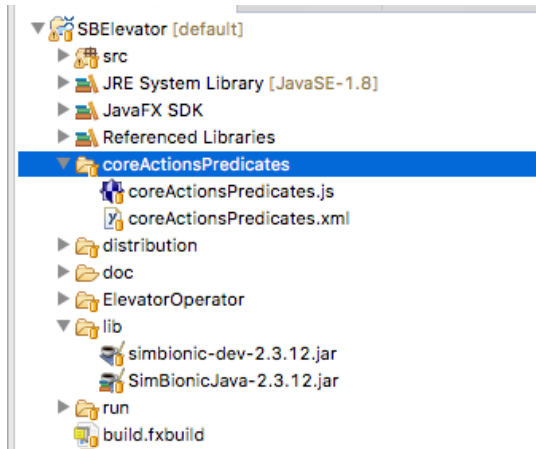
Figure 3. Example of Adding SimBionic to an Existing Java Project in Eclipse.
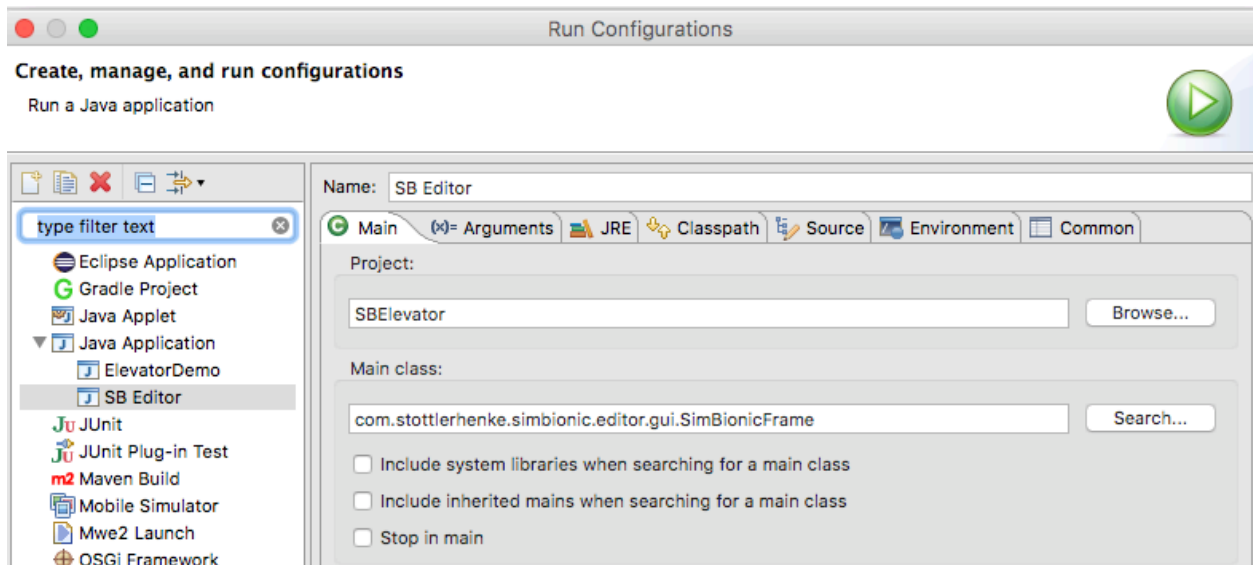
Figure 4. Launch Configuration to Run the SimBionic Editor.

2.6 Creating a Software Application Using SimBionic

To develop a software application which uses SimBionic:
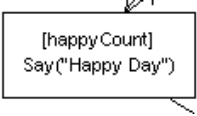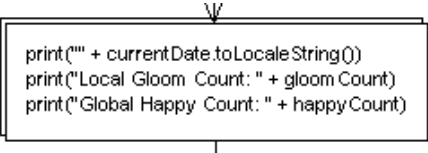
1. Identify (and, if necessary, develop) the JavaScript functions and Java objects that will be used by your SimBionic BTNs.

2. Using the SimBionic Editor, create and edit a SimBionic project file that specifies BTNs, action and predicate declarations, constants, and global variables. Section 3 describes Behavior Transition Networks, and Section 4 describes how to use the SimBionic Editor.

3. Add Java code to your application that creates, initializes, and invokes the SimBionic engine. This step is described in Section 3.

4. Compile and run the application that embeds the SimBionic engine.
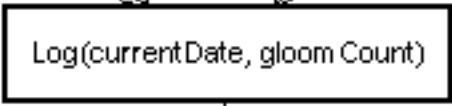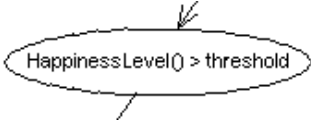
3. Behavior Transition Networks

The Hello World example will be used to describe the basics of BTNs as shown in Figure 1. The **Main** BTN starts in the top green node and flows from this node as indicated by the directed connectors. If *Happiness Level > threshold*, it will reach the *Happy Day* node; otherwise, it finds *Gloom and Doom*. Once it gets past these, flow of control moves to the **Log** BTN to print some summary information. Once that BTN is done, control comes back to **Main** and control flows to the final red node at the bottom of the behavior.

3.1 Behavior Nodes

There are four commonly used types of nodes in a BTN:

Node Type	Appearance of Icon on the SimBionic Editor Canvas	Description
Action node	Rectangle [happyCount] Say("Happy Day")	Each action node specifies a single Javascript expression which is evaluated when the node is executed by the SimBionic run-time engine. In the node on the left, *[happyCount]* indicates a variable binding and *Say("Happy Day")* a call to a Javascript method. Each behavior has one initial node (green). This is the start node for the behavior. Each behavior can have one or more final nodes (red) that signify the end of the behavior. Final nodes do not contain expressions.
Compound node	Two Closely Overlapping Rectangles print("" + currentDate.toLocaleString()) print("Local Gloom Count: " + gloomCount) print("Global Happy Count: " + happyCount)	Each compound action node specifies one or more Javascript expressions which are evaluated when the node is executed by the SimBionic run-time engine. Each *print* call refers to a Javascript method. The first two lines use variables that are local to this particular BTN, where currentDate is an instance of *java.util.Date* and *gloomCount* is an integer. The third *print* line references *happyCount*, which is a global variable shared by all BTNs running on the same SimBionic entity.
Behavior	Rectangle with Thick Border	Behavior nodes invoke other behaviors. If

node	 Log(currentDate, gloomCount)	the behavior being called accepts parameters, the behavior node specifies the values of the parameters to be used when invoking the behavior.
Condition node	Oval HappinessLevel() > threshold	Each condition node specifies a Javascript expression that is evaluated to determine whether a transition should occur between action nodes or sub-behavior nodes.

3.2   Actions and Predicates

A SimBionic **predicate** node is a JavaScript expression, JavaScript function call, or Java method call that returns a value. Predicates which return Boolean values are useful for testing whether a specified condition is true. However, predicates can also be used to obtain a numeric or String value or a Java object. A SimBionic **action** is a Javascript function call or Java method call that does not return a value. SimBionic provides built-in actions and predicates. You can also define your own actions and predicates and associate them with Javascript functions that are loaded at run-time from a Javascript file. Defining actions and predicates is a syntactic sugar for creating an empty node and typing in the Javascript expression and is convenient for invoking expressions that are frequently used.

Empty action and condition **nodes** can be created with the right-click popup menu in a BTN. Defined action and condition nodes are added to the BTN via drag-and-drop from the catalog of defined actions and predicates.

3.3   Connectors

Each connector drawn between nodes in the BTN is numbered. These numbers specify the order in which the SimBionic run-time engine tries to transition from a node. For example, in Figure 5, the green node has two connectors leaving it, so the SimBionic engine first tries to evaluate the predicate within the oval condition node that is associated with the connector labelled "1." If the value of the predicate is true (or if there had not been a condition node associated with the connector), execution transitions to the action, compound action, or behavior node that follows the condition node. If the condition's predicate is false, the engine tries to transition from the node using the connector labeled "2."
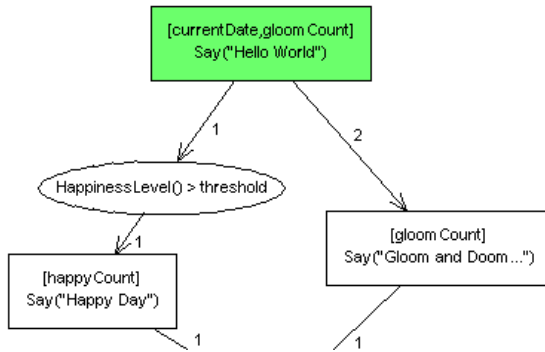
Figure 5. Numbered connectors in a behavior.

Connectors are created by holding down the control key while clicking-and-dragging from one node to another. Selecting a connector will enable visualization of the start node (green circle) and end node (red circle) as shown in Figure 6. Connectors that are not attached to nodes will be visualized with yellow circles.
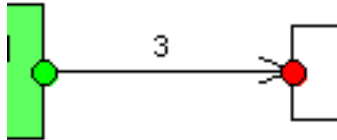
Figure 6. Selecting a connector visualizes the connection between the start node (green circle) and end node (red circle).

3.4   Local Variables

SimBionic local variables are accessed only from within each BTN in which the variable has been declared (Figure 7). Each local variable exists only while an entity is executing the associated BTN. If two or more entities are executing the same BTN at the same time, each combination of entity and executing BTN will have its own set of local variable values.
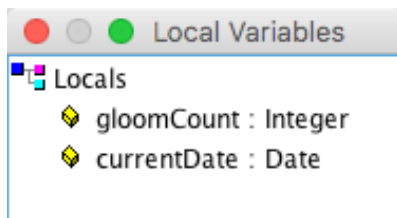
Figure 7. Local Variables dialog.

3.5   Global Variables

BTNs within the same SimBionic entity can share data by reading and writing global variable values (Figure 8). Each entity has its own set of global variables which it can access from within any BTN. For example, if a SimBionic project defines a global variable named *happyCount*, each entity will have its own version of these variables. The global variable named *happyCount* that is accessed by one entity can contain different values than the global variable named *happyCount* that is accessed by a second entity.

To share data between entities, you can write BTNs that send SimBionic *messages*, using the actions and predicates in the *Messages* folders in the *Catalog* pane. Or, your BTNs can share information by calling methods of shared Java objects.
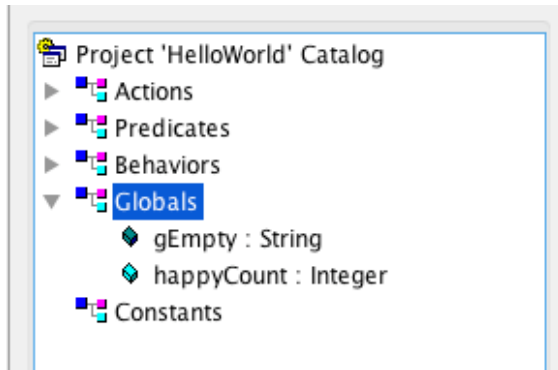
Figure 8. Global Variables in the catalog.

3.6   Expressions

Double-clicking on an action or predicate node brings up the JavaScript expression editor shown in Figure 9. An expression for a predicate may contain multiple JavaScript statements and may include reading from local and global variables. The last statement must evaluate to a Boolean. An expression for an action may contain multiple statements and may include reading from local and global variables.
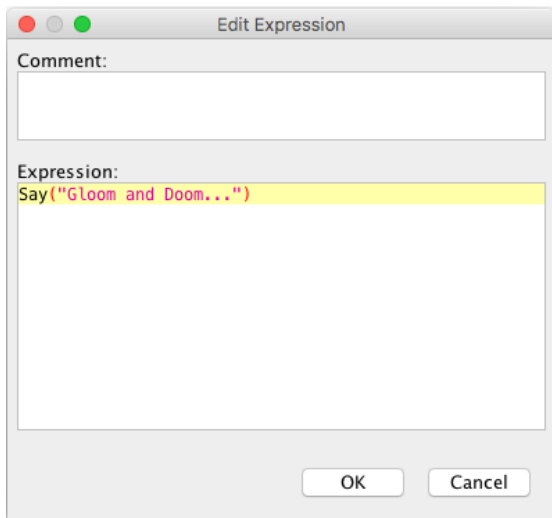
Figure 9. Editing an expression for an Action or Predicate node.

3.7   Bindings

A binding is the assignment of a local or global variable to the value of a Javascript expression. You can specify bindings within action nodes, compound nodes, and condition nodes as well as on connectors. Figure 10 shows bindings of *currentDate* to a newly created Java class and the

integer *gloomCount* to 0. The Edit Bindings dialog is accessed by right-clicking on a node or connector and selecting Edit Bindings.
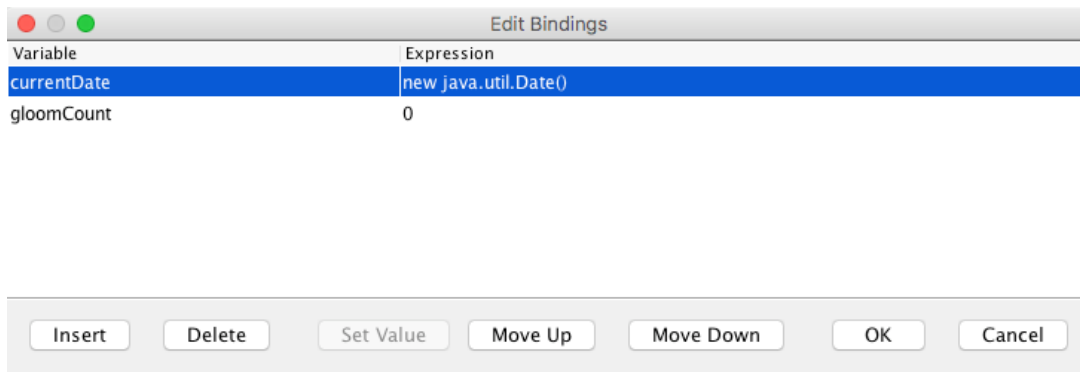
Figure 10. The Edit Bindings dialog. This is accessed by right-clicking on an object and selecting Edit Bindings.

4. Using the SimBionic Editor

The SimBionic Editor contains five main components: the Project, Canvas, and Console views, a menu bar, and a toolbar. The **Project View** on the left of the screen provides access to the actions, predicates, behaviors, global variables, and polymorphic descriptors. The **Canvas View** in the center is the graphical component that uses the project window to develop BTNs. The **Console View** at the bottom display the results from checking for errors in the BTNs, debugging, and the Find command.

4.1   Project View

The sub-window that runs across along the left side of the SimBionic editor is called the *Project View*. It houses two panes, called the *Catalog* and the *Descriptors pane*, which you use to set up all the actions, predicates, behaviors, global variables, constants, and descriptors. Click on the tab labeled Catalog to see the Catalog pane. When the Catalog is selected, the Project View looks something like Figure 11. The Descriptor pane is described in 7.1 Descriptors and Polymorphism.
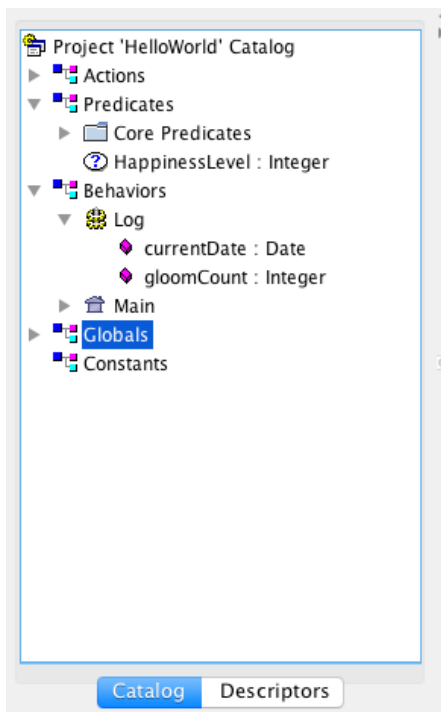
Figure 11. Project View with the Catalog displayed.

The Catalog displays an icon for every action, predicate, behavior, global variable, and constant (program construct) used in your project. These icons are organized within folder icons. For example, the folder icon labeled Actions contains two sub-folders labeled Core Actions and Messages. The first sub-folder contains icons for core actions, and the second sub-folder contains icons for actions related to message sending. You can right-click on any icon to display a context menu of operations that can be applied to the folder or program construct. For example, if you right-click over the action icon labeled DestroyEntity, the SimBionic Editor displays a context menu with choices: Insert Parameter, Rename, Delete, Set Description, and a toggle button that indicates that the action is Reserved. Reserved actions are those that are

11

included as part of the coreActionsPredicates.js. These should not be edited unless the corresponding JavaScript file is edited.

You can double-click on an icon to invoke the default operation, which is usually displaying its description. You can also drag an action, predicate, or behavior icon to the Canvas to quickly add it to the BTN currently being edited. If you need to refresh your memory about the meaning of a particular item in the Catalog, simply hover your mouse cursor over its name. After a moment, a tooltip with the description of that item will appear.

Action folder context menu options:

Insert Action	Adds a new action with a default name, such as *NewAction*, to the list of actions within the selected action folder.
New Folder	Adds a new action folder with a default name, such as *NewFolder*, to the list of subfolders within the selected action folder.
Rename	Makes the name of the selected action folder editable. To rename the folder, type the new name and then press *return*.
Delete	Deletes the selected action folder.

Action context menu options

Insert Parameter	Adds a new parameter with a default name, such as *NewParameter*, to the list of the currently selected action's list of parameters.
Rename	Makes the name of the selected action editable. To rename the action, type the new name and then press *return*.
Delete	Deletes the selected action.
Set Description	Displays a popup window that prompts you to enter a new description of the action.

Action parameter context menu options:

Rename	Makes the name of the selected action parameter editable. To rename the parameter, type the new name and then press *return*.
Delete	Deletes the selected action parameter.
Set Type	Sets the data type of the parameter (e.g., String, Integer, Boolean, Float, etc.).
Move Up	Moves the parameter up (earlier) in the ordered list of the action's parameters.
Move Down	Moves the parameter down (later) in the ordered list of the action's parameters.

Like the context menu for action folders, the context menu for **predicate folders** contains the options: New Folder, Rename, and Delete. The predicate folders context menu also contains the option:

Insert Predicate	Adds a new predicate with a default name, such as *NewPredicate*, to the list of predicates within the selected predicate folder.

The context menu for **predicates** contains the same options as the context menu for actions, as well as the following menu option:

Set Return Type	Sets the data type of the value returned by the selected predicate (e.g., String, Integer, Boolean, Float, etc.).

The context menu for **predicate parameters** contains the same options as the context menu for action parameters.

Like the context menu for action folders, the context menu for **behavior folders** contains the options: New Folder, Rename, and Delete. The behavior folders context menu also contains the option:

Insert Behavior	Adds a new predicate with a default name, such as *NewBehavior*, to the list of behaviors within the selected predicate folder.

Like the action context menu, the **behavior** context menu contains the options: Insert Parameter, Rename, Delete, and Set Description. The behavior context menu also contains the following additional options:

Duplicate Behavior	Creates a new behavior with the same parameters and other attributes as the selected behavior.
Set Execution	Sets the execution mode of the behavior to one of the following: • Run multi-tick • Run in one tick • Run until blocked These options are described in the Advanced Topics section.
Set Interruptibility	Sets the interruptibility of the behavior to one of the following: • Interruptible • Non-Interruptible These options are described in the Advanced Topics section.

Global variables folder context menu options:

Insert Global	Adds a new global variable with a default name, such as *NewGlobal*, to the list of global variables within the selected folder. The name of the global variable is highlighted and editable. By convention, names of SimBionicglobal variables start with "g."

Like the action parameter context menu, the **global variables** context menu contains the options: Rename, Delete, Set Type, Move Up, and Move Down. The global variables context menu also contains the option:

Set Initial Value	Sets the initial value of the global variable.

Constants folder context menu options:

Insert Constant	Adds a new constant with a default name, such as *NewConstant*, to the list

	of constants. The name of the constant is highlighted and editable.

Like the global variables context menu, the context menu options for **constants** contains the options: Rename, Delete, Set Type, Move Up, and Move Down. The context menu for constants also contains the option:

Set Value	Sets the permanent value assigned to this constant.

4.2 Canvas View

The Canvas is the large white area on the right side of the Editor window as shown in Figure 12. The Canvas displays one BTN at a time, selected in the Catalog pane. The Canvas is used to edit nodes, connectors, and local variables.
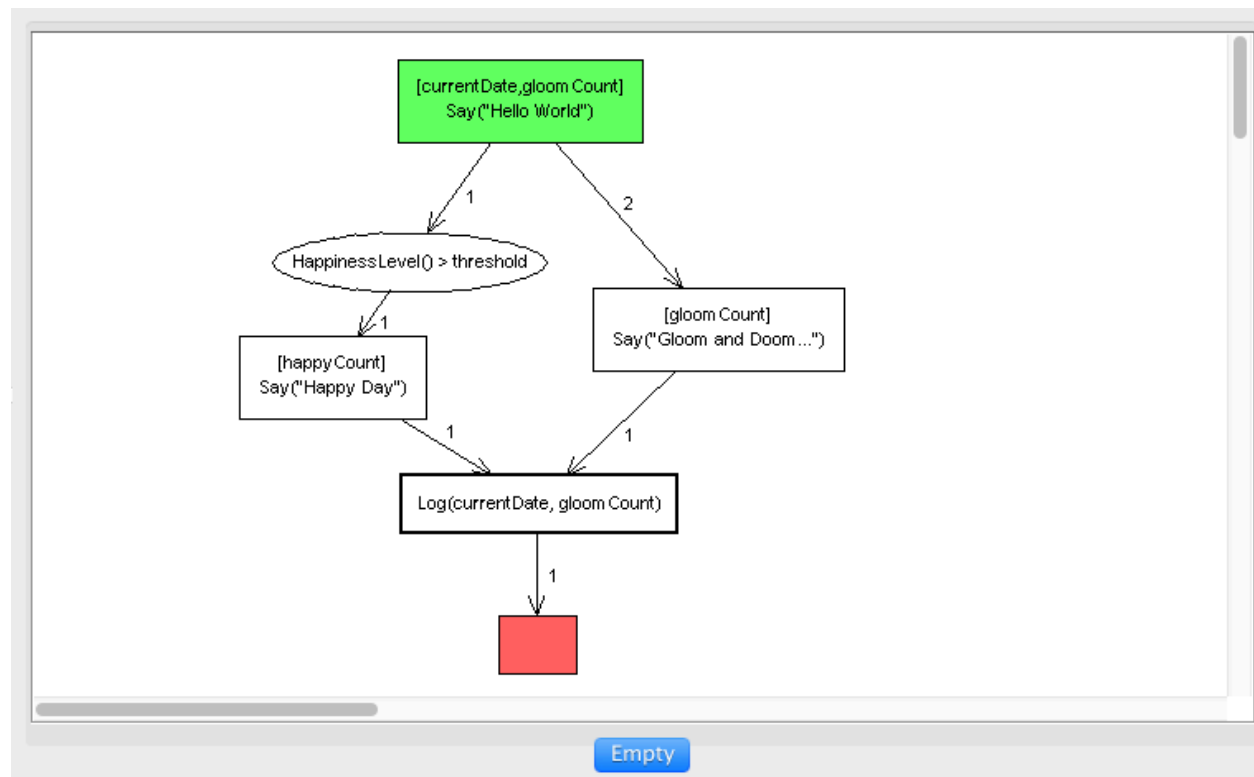
Figure 12. Canvas View of the Main BTN in the Hello World sample.

4.2.1 Nodes

You can add a node to the behavior as follows:
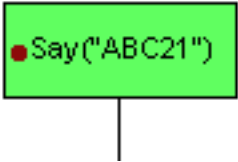
Node Type	User Interaction
Action node	• Drag the name of an action from the Catalog onto the Canvas, or • Right-click over the canvas and select *Insert Action* from the context menu.
Compound action node	• Right-click over the canvas and select *Insert Compound Action* from the context menu.
Condition node	• Drag the name of a predicate from the Catalog onto the Canvas, or

	• Right-click over the canvas and select *Insert Condition* from the context menu.
Behavior node	• Drag the name of a behavior from the Catalog onto the Canvas, or • Right-click over the canvas and select *Insert Action* from the context menu.

After you have added a node to the canvas, double-click on the node. The SimBionic Editor will then display a dialog that enables you to edit the details of the node, as shown in Figure 9.

Click to select a node. Press control-click to select multiple nodes. Click, drag, and release to draw a rectangle that lassos a set of nodes and connectors to select. The Editor will then display the node's expression in the text box labeled E.

Right-click on a node to display its context menu. The context menu for all four types of nodes contains the following options:

Cut Node	Cuts the selected node(s) and/or connector(s) to the SimBionic clipboard so it can be pasted elsewhere.
Copy Node	Cuts the selected node(s) and/or connector(s) to the SimBionic clipboard so it can be pasted elsewhere.
Delete Node	Deletes the selected node.
Edit Bindings	Displays the Edit Bindings dialog that enables you to add, delete, modify, and reorder the node's ordered list of bindings.
Set Label	Sets the label that is displayed inside the node icon on the canvas.
Edit Comment	Edits the comment that describes the node.
Go To Declaration	Displays and highlights the node's entry in the Catalog, opening Catalog folders if necessary.
Toggle Breakpoint	Use this command to set or remove a breakpoint on the behavior. The breakpoint is visually indicated by a bullet in the behavior node, as shown below. A breakpoint indicates that the debugger will stop at this behavior element during execution.

The context menus for action nodes, compound action nodes, and behavior nodes contain the following additional options:

Initial	This option sets the node to be an *initial* node. Execution of every behavior begins at the initial node. Each behavior must have one initial node.

Final	This option sets the node to be a *final* node. Execution of every behavior ends at a final node. Each behavior must have at least one final node.
Always	This option sets the node to be an *always* node. If the node is already set to be an always node, selecting this option undoes the node's *always* setting. Always nodes are colored yellow.
Catch	This option sets the node to be a *catch* node. If the node is already set to be a catch node, selecting this option undoes the node's catch setting. Catch nodes are colored blue.

4.2.2   Connectors

Directed connectors are drawn to describe the flow of control between nodes. To create a connector, either press the **+** bar in the toolbar or hold down the CTRL button while clicking and dragging from one node to another. While selected, correctly placed connectors will have a visible green circle (start) and red circle (end). If a connector is not properly attached to a node, it will show a yellow circle at the unattached end.

Right-click on a connector to display its context menu with the following options:

Cut Link	Cuts the selected node(s) and/or connector(s) to the SimBionic clipboard so it can be pasted elsewhere.
Copy Link	Cuts the selected node(s) and/or connector(s) to the SimBionic clipboard so it can be pasted elsewhere.
Delete Link	Deletes the selected link.
Set Priority	Sets the priority of the link.
Interrupt	Sets whether or not the link is an interrupt.
Edit Bindings	Displays the Edit Bindings dialog that enables you to add, delete, modify, and reorder the node's ordered list of bindings.
Set Label	Sets the label that is displayed inside the node icon on the canvas.
Edit Comment	Edits the comment that describes the node.

4.2.3   Local Variables

To edit the BTN's set of local variables (Figure 7), click on the button labeled Locals in the upper right corner of the SimBionic Editor Window. The Editor will display a dialog that lists the names of the local variables. To add a local variable, right-click on the entry labeled Locals and select *Insert Local…* from the context menu. Right click on the name of a local variable to display the local variable context menu:

Rename	Makes the name of the selected local variable editable. To rename the local variable, type the new name and then press *return*.
Delete	Deletes the selected local variable.
Set Type	Sets the data type of the local variable (e.g., String, Integer, Boolean, Float,

	etc.).
Move Up	Moves the local variable up (earlier) in the ordered list of the behavior's local variables.
Move Down	Moves the local variable down (later) in the ordered list of the behavior's local variables.

4.3   Console View

The Console View is at the bottom of the SimBionic Editor window. An example after a search is shown in Figure 13.
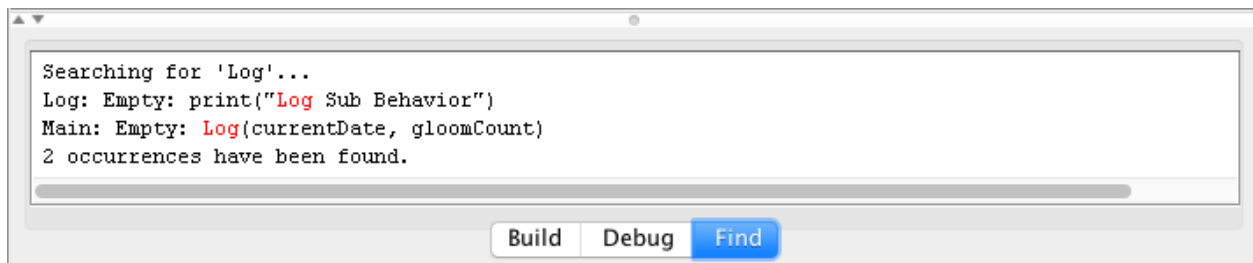
Figure 13. Console View in the SimBionic Editor.

Three tabs control the type of output displayed in the console:

Build	Makes the name of the selected local variable editable. To rename the local variable, type the new name and then press *return*.
Debug	Debugging output is displayed when the Debug tab is selected.
Find	Displays a mouse-sensitive list of nodes and behaviors that contain the user-specified text string. Double-click on the list entry to display the behavior in the Canvas.

4.4   Menu Bar

File menu:

New	Creates a new SimBionic project and refreshes the SimBionic Editor user interface to display this new empty project.
Open	Displays a file selection dialog that prompts the user to select the name of a SimBionic project file. Then, opens the selected project file and refreshes the SimBionic user interface to display the project.
Save	Saves the currently open SimBionic project file, using the current file name. If the SimBionic project file does not yet have a name, the SimBionic Editor will prompt the user for a file name.
Save As…	Prompts the user for a file name and saves the currently open SimBionic project file using this filename.
Exit	Exits from the SimBionic Editor application.

17

Edit menu:

Undo	Undoes a canvas-related operation.
Redo	Redoes the undone operation.
Cut	Cuts selected elements from the displayed behavior.
Copy	Copies selected elements from the displayed behavior.
Paste	Pastes the object that was most recently cut or copied.
Delete	Deletes the selected object.
Select All	Selects all of the elements in the current behavior.
Find	Finds the given text, searching through all of the behavior elements.
Replace	Replaces the text string found in an object with a new text string.

View menu:

Go Back	Goes to the previously displayed canvas.
Go Forward	Goes to the canvas displayed after the current canvas (following a Go Back command).
Previous Error	Displays the previous error in the console window.
Next Error	Displays the next error in the console window (following a Previous Error command).

Project menu: Provides overview of debugging support in Editor. The Project menu includes the following options for debugging:

Check for Errors	Examines each behavior for errors. Any errors (i.e., serious problems) and warnings (i.e., minor or potential problems) encountered are listed in red in the Build pane of the Output window at the bottom of the editor window. Double-click on the error to jump to the appropriate behavior.
Toggle Breakpoint	Select a canvas element and then use this command to set or remove a breakpoint on the behavior.
Connect to Execution Engine	This option initiates a connection between the Authoring tool and a SimBionic run-time engine. If the connection is successful, the Authoring tool will switch to the visual debugger perspective. See Section 8, Using the SimBionic Debugger, for information on how to do this.
Connection Settings	This option sets the IP address to use when searching for a SimBionic run-time engine to connect with when using the visual debugger.

The Project menu also provides access to the Javascript Settings dialog (Figure 14).

Javascript Settings	This menu option enables you to specify: a) The Javascript files that define Javascript functions that are associated (directly or indirectly) with actions and predicates used by the SimBionic

	project and
	b) The names of imported Java classes that can be called from within the Javascript functions.
	When the SimBionic run-time engine initializes, it creates a Javascript engine embedded within the Java run-time system, it loads the specified Javascript files, and it imports the specified Java classes into the Javascript engine.
	• SimBionic depends upon Java 8 and uses the Nashorn Javascript Engine provided as part of Java 8. • See Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM for information on writing Javascript for Nashorn.

Figure 14. Javascript settings dialog.

4.5 Toolbar

The SimBionic Editor provides two rows of toolbar buttons for quick access to commonly used functions. To see a description of a toolbar button, hover your mouse over the button.

In the top toolbar row (Figure 15), the first eight buttons correspond to File and Edit menu options. The central three buttons and drop-down selection support quick navigation through the available BTNs. The **+** button will create a connector from the currently selected node (if any). Pressing the button labeled Locals causes the SimBionic Editor to display a dialog that enables you to edit the local variables for the currently edited behavior. The remaining four buttons correspond to Project menu options.
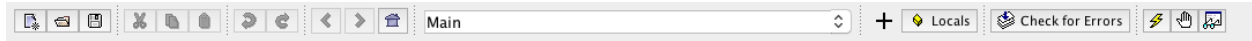
Figure 15. Top toolbar row.

In the bottom toolbar row (Figure 16), the text edit box labeled *E* enables you to view and edit the Javascript expression associated with the currently selected action node, condition node, or behavior node. The pulldown menu labeled *B* enables you to edit the set of bindings associated with the selected node.
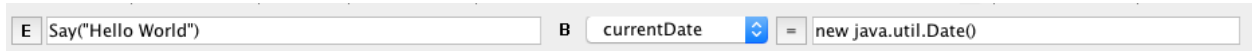
Figure 16. Bottom toolbar row.

5. Embedding the SimBionic Run-Time System

This section describes how to embed the SimBionic run-time engine within a software application. It is taken directly from the HelloWorld.java file.

The first step is to create and fill out an SB_Config object. In this case the debugger is turned off and the behavior model to load is HelloWorld.sbj.

```java
SB_Config myConfig = new SB_Config();
myConfig.debugConnectTimeout = 90;
myConfig.debugEnabled = false;

try
{
    String filename = "samples/HelloWorld/HelloWorld.sbj";
    myConfig.fileURL = new URL("file", "localhost", filename);
}
catch(Exception e)
{
  fail("Exception creating URL from the sim file name.");
  return;
}
```

After that, the engine is created and logging streams are set up.

```java
// creates an instance of the engine interface object
SB_Engine engine = new SB_Engine();

        // specify the log file
PrintStream logPrintStream = null;
try
{
  logPrintStream = new PrintStream(
                 new FileOutputStream(
                 "samples/HelloWorld/HelloWorld.log") );
}
catch(Exception ex)
{
  fail(ex.getMessage());
  return;
}

if( logPrintStream != null )
{
  engine.registerLogPrintStream( logPrintStream );
}
```

With the desired configuration and logging in place, the SimBionic engine is initialized.

```java
if (engine.initialize(myConfig) != SB_Error.kOK)
{
  fail("Engine failed to initialize: " + engine.getLastError());
  return;
}
```

Next, the entity 'Friendly Guy' is created and set to an initial behavior of 'Main.' The Main behavior expects a single parameter of an integer, which is created and added to an array of SB_Param objects.

```
long friendlyGuy = engine.makeEntity( "Friendly Guy" );
if (friendlyGuy == -1 )
{
  fail("Entity creation error: " + engine.getLastError());
  return;
}

String behavior = "Main";
ArrayList<SB_Param> params = new ArrayList<SB_Param>();
params.add(new SB_Param(new Integer(52)));
SB_Error errCode = engine.setBehavior( friendlyGuy, behavior, params);
if (errCode != SB_Error.kOK)
{
    System.out.println("Error setting behavior: " + engine.getLastError());
    return;
}
```

At this point, the SimBionic engine has been created and initialized and a single entity exists running the behavior Main. The engine is then updated 30 times so that the entity can step through the entire Main and Log BTNs. After 30 updates, the engine is shut down and the memory used by the engine is released.

```
for (int tick=0; tick < 30; tick++)
{
  if (engine.update() != SB_Error.kOK)
  {
    fail("Update error: " + engine.getLastError());
    return;
  }
}

engine.terminate();      // shut down the engine
```

This example is meant to highlight how to use the SimBionic runtime engine, adding one entity at the beginning and performing 30 updates. An actual application would likely add and remove entities over time and would constantly update the engine on a timer while the game or simulation is running.

6. Advanced Topics: Flow of Control

This section covers topics that are useful for understanding the flow of control within SimBionic BTNs.

6.1   Typical flow of execution

The run-time engine typically moves forward an entity's behavior one action every time *update* is called. The engine starts by executing the bindings (if any) attached to the current rectangle of the topmost behavior on the entity's execution stack; and then executes the current rectangle itself (which we'll call rectangle A).

If the rectangle is an action, the engine simply executes the action.

If the rectangle is an invoked behavior, then the engine pushes that behavior onto the stack and sets the current rectangle for the newly invoked behavior to be its initial rectangle.

The engine then evaluates the connectors coming out of rectangle A in order of priority. What the engine does next depends on the circumstances:

- If rectangle A is an action and its connector leads directly to another rectangle, SimBionic assigns execution for the next clock tick to flow to that second rectangle.

- If rectangle A is an action and its connector leads to a condition or series of conditions that all evaluate as true, SimBionic assigns execution for the next clock tick to flow to the rectangle that directly follows the condition or series of conditions.

- If rectangle A invoked a behavior that hasn't yet completed execution and its connector leads directly to another rectangle, SimBionic assigns execution to remain within the invoked behavior for the next clock tick. This cycle will repeat until the behavior has completed (i.e., reached a final action), at which point execution will flow to the second rectangle.

- If rectangle A invoked a behavior that hasn't yet completed execution and its connector leads to a condition or series of conditions that all evaluate as true, SimBionic aborts the invoked behavior (i.e., pops it from the stack) and assigns execution for the next clock tick to flow to the rectangle that directly follows the condition or series of conditions.

- If none of the situations above apply, and none of rectangle A's connectors lead to conditions that all evaluate as true, *and* if there are any behaviors above this one on the stack, SimBionic will proceed to check the connectors coming out of the current rectangle in the next behavior on the stack, according to the rules described above.

Otherwise, SimBionic assigns execution for the next clock tick to remain within the current rectangle. Specifically, if the rectangle is a behavior that hasn't been completed, the behavior will continue executing during the next clock tick; while if the rectangle is an action, or is a behavior that *has* been completed, then no further execution will take place in the rectangle during the next clock tick. In any case, the rectangle's connectors will be evaluated again during the next tick. The cycle just described continues until one of the connectors evaluates as true.

Note: If there are any bindings attached to the rectangle, they aren't re-executed during subsequent clock ticks.

Tip: If you'd prefer that an action continue being executed over and over until one of its current connectors evaluates as true (as opposed to doing nothing after the first clock tick), you can add a last connector that leads back to the rectangle. To do so, press the Crtl key and, while keeping it held down, click the rectangle and drag your mouse outward until you see a green starting point, and then drag your mouse back to the rectangle until you see a red endpoint. When you release your mouse button, the connector curves to both begin and end in the rectangle.

Again, the above represents the engine's typical flow of execution. There are some exceptions to these general rules when different *execution modes* and *interrupts* are used.

6.2   Behavior Execution Modes

Typically, the application program first creates a SimBionic engine object, creates one or more SimBionic entity objects, specifies an initial behavior for each entity, and then repeatedly calls the engine's *update* method. When the engine is no longer needed (i.e., when the application is finishing), the application calls the SimBionic engine's *finish* method.

Each call to the update method is called a *clock tick*. At each clock tick, the SimBionic run-time engine continues to execute each entity's behavior, starting where it left off the last time the *update* method was called. The amount of computation that is carried out during each call to *update* is controlled by the *execution mode*. By default, behaviors are *multi-tick*. Only a single action node or compound action node is executed at each clock tick. However, you can increase the amount of computation that is carried out at each clock tick for a BTN by setting the execution mode to one of the following. Note that each BTN can be assigned its own execution mode.

Execution Mode	Description
Run multi-tick	During each clock tick, **execute one rectangle** (i.e., action or invoked behavior) of the behavior, and then evaluate the connectors coming out of that rectangle. If the conditions on any of those connectors evaluate as true, move to the rectangle at the end of that connector **and then stop**. Repeat this cycle every clock tick until the behavior reaches a final action or is aborted or suspended by another behavior. This option is the default, because it allows you to abort or suspend a behavior when new circumstances arise, and it also provides SimBionic maximum flexibility in juggling the various demands on its execution time.
Run in one tick	**Execute the entire behavior** (i.e., from its initial rectangle to its final action) in a single clock tick. However, after executing each rectangle of the behavior, evaluate the connectors at lower levels of the execution stack in case at any point conditions are met that should cause the behavior to be aborted or suspended. This option is appropriate for a behavior that requires quick execution (e.g., reloading ammunition during a battle).
Run until blocked	During each clock tick, **execute one rectangle of the behavior**, and then evaluate the connectors coming out of that rectangle. If the conditions on one of those connectors evaluate as true, **move to the rectangle at the end of that**

| | **connector and execute it** in the same clock tick. **Repeat this cycle** until none of the conditions leading out of the current rectangle evaluate as true, and then stop. Repeat every clock tick until the behavior reaches a final action or is aborted or suspended by another behavior. |
| | This option is appropriate for behaviors that are primarily driven by outside events (e.g., a gun turret that is only active when an enemy is in view) because it effectively allows the behavior to switch between multi-tick and one-tick modes depending on how much is going on in your application. |

Note: Some of these modes take precedence over others: run until blocked mode overrides multi-tick mode, and one-tick mode overrides both other modes. If a behavior with a higher-precedence execution mode invokes a behavior with a lower-precedence mode, the invoked behavior will use the higher-precedence mode. For example, if a one-tick behavior invokes a multi-tick behavior, the invoked behavior will execute in one tick as well.

6.3   Behavior Interruption Modes

By default, a behavior can be aborted or suspended by behaviors below it on the execution stack, as discussed in the topic, Understanding SimBionic's flow of execution. In some cases, however, you may want a behavior to execute without being interrupted by any other behaviors—for example, if it is performing a critical task. You can control this aspect of a behavior's execution by modifying its *interruptibility mode*.

There are two interruptibility mode options available for behaviors:

Interruptibility Mode	Description
Interruptible	Evaluate connectors and conditions in behaviors that are below this one on the execution stack. If a condition on one of these lower-level connectors evaluates as true, abort this behavior (or suspend it if the connector is an interrupt connector).
	This option is the default, because it allows you to abort or suspend a behavior when new circumstances arise. It is appropriate when a behavior may need to be overridden by a more urgent activity (e.g., evading flying shrapnel).
Non-interruptible	Do not evaluate connectors and conditions in behaviors below this one on the execution stack. This behavior thus cannot be aborted or suspended.
	This option is appropriate for top-priority behaviors and/or behaviors that won't be effective if interrupted (e.g., transmitting a critical message).

6.4   Interrupt Connectors

If a connector coming out of an interruptible behavior leads to a condition that evaluates as true, then SimBionic will normally abort the behavior on the spot and redirect execution flow to the rectangle (i.e., action or behavior) directly following the condition.

However, if you don't want the original behavior to be abandoned, just temporarily suspended while the second behavior executes, you can designate the connector coming out of the behavior to be an *interrupt* connector. When the behavior following such a connector is triggered, it takes temporary control of execution until its job is done, and then returns execution to the original behavior so it can pick up where it left off.

To create an interrupt connector, right-click the connector you want to affect and select the Interrupt option from the menu that appears; or click the connector to select it and then click the *Interrupt* button, which is directly to the right of the Elbow buttons. The connector changes from a normal line to a broken line to indicate that it is now set to interrupt the current behavior (as opposed to aborting it). If you later change your mind, you can select the Interrupt option again or click the Interrupt button again to revert the connector to its normal state.

Important Note: The behavior at the end of the interrupt connector *must* end in a <u>final action</u>. Otherwise, the interrupting behavior will never be able to return control to the original behavior, and your program will freeze.

6.5 Checking for completed behaviors

When a standard (i.e., interruptible) behavior is invoked, during every clock tick SimBionic executes a rectangle within that behavior and then evaluates the connector(s) coming out of that behavior.

If any of the behavior's connectors leads to a condition that evaluates as true, the behavior will be aborted mid-stream. SimBionic's execution will then be redirected to flow down the path following the condition.

In some cases, this is precisely what you want to occur—for example, when a new situation arises that makes the current invoked behavior superfluous.

In many cases, however, you'll want a behavior to entirely finish before execution flows down the path of one of its connectors. To ensure this occurs, you must insert an additional condition that checks on whether the behavior is done executing—i.e., has reached a final action. This checking is performed by the *IsDone* predicate, which evaluates as false when a behavior is still running and true when the behavior is completed. IsDone is available in every SimBionic project; it is automatically declared in the Catalog, with corresponding code built into SimBionic's engine.

Using IsDone is only necessary when a behavior's connector leads to a condition. That's because the condition allows you to test for a change in your virtual world which, if it becomes true, may make you want to abort the behavior.

By contrast, if a behavior's connector leads to a rectangle, SimBionic assumes that you don't want to transition out of the behavior until it's completed—because you aren't checking for any changed situation that might impel you to abandon the behavior.

As a result, when a behavior's connector leads to a rectangle, SimBionic *automatically* inserts an *invisible* IsDone condition between the behavior and the rectangle. This is referred to as an *implicit IsDone*, because it is always built into a behavior-to-rectangle connection for you; and even though you can't see it, you can assume it's there.

Therefore, you never need to insert an IsDone condition between an invoked behavior and a rectangle; but you *do* need to use IsDone whenever you want to ensure a *condition* doesn't abort a behavior.

6.5.1   An IsDone example

To help you visualize the practical usage of IsDone, assume your primary behavior on the Canvas is named Main, and within Main you invoke another behavior called DoSomething. Also assume that coming out of DoSomething are three connectors that are respectively designed to handle emergency orders, new orders, and standard orders, as shown in Figure 17:
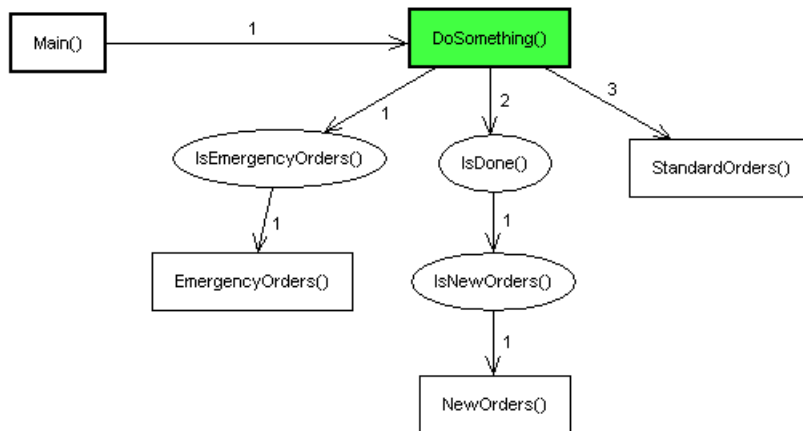
Figure 17. An example if using IsDone in a BTN.

After executing one of DoSomething's rectangles, SimBionic evaluates the first connector, which leads to a condition that checks on whether emergency orders have been received. If they have, the DoSomething behavior is immediately aborted, and SimBionic redirects execution to the EmergencyOrders action. Otherwise, SimBionic proceeds to evaluate the middle connector.

The second connector leads to two conditions. The first is an IsDone condition that checks on whether DoSomething has completed executing. If IsDone evaluates as true, then a second condition checks on whether new orders have been received. If this condition *also* evaluates as true, then SimBionic directs execution to the NewOrders action. Otherwise, SimBionic proceeds to evaluate the final connector.

The last connector leads to a default StandardOrders action. No IsDone condition is visible between DoSomething and the action; but because this is a behavior-to-rectangle connection, it automatically includes an implicit IsDone. As a result, the StandardOrders action won't be triggered unless DoSomething has completed execution by reaching an internal final action. Otherwise, SimBionic realizes that it has run out of connectors and returns the flow of execution to DoSomething.

If DoSomething has been declared as a multi-tick behavior, SimBionic will execute the cycle above once every clock tick, until DoSomething is either aborted or completed.

Alternatively, if DoSomething has been declared as a one-tick interruptible behavior, SimBionic will execute the cycle above over and over again during the same clock tick, until DoSomething is either aborted or completed.

Note #1: If DoSomething had been declared as a one-tick non-interruptible behavior, then the description above wouldn't apply. Instead, SimBionic would skip evaluating any of DoSomething's connectors until the behavior had entirely finished executing. As a result, you never need to use an IsDone condition with a non-interruptible behavior, because the behavior is guaranteed to complete execution before it encounters any conditions that might otherwise abort it.

Note #2: In the example above, all of the connectors are standard ones. Alternatively, however, we could have set one or more of the connectors to be an *interrupt*. In that case, whenever all of the conditions (both explicit and implicit) on an interrupt connector evaluated as true, DoSomething would *not* be aborted, but just temporarily *suspended* while execution flowed to the connector's rectangle. After the rectangle finished executing, control would return to DoSomething, which would continue executing where it left off. The interrupt option therefore provides you with a way of responding to new situations rapidly without permanently disrupting a current behavior.

7. Advanced Features: Communication, Polymorphism, & Exceptions

This section covers features that are useful in developing more complex BTNs.

7.1 Communication Among Entities

SimBionic's **Group Messaging** feature enables any entity to join a named group of entities and receive every message directed at that group. Group Messaging also allows any entity to send a message to a named group, regardless of whether the entity belongs to the group. Every group message is sent to the first-in, first-out message queue of each entity in the group. Each message specifies the name of the entity that sent the message (data type entity), an integer that identifies the message type, and the actual message, which may be of any data type.

Virtual Blackboards are storage areas that enable any entity to share information with any or all other entities in the program. Each blackboard is divided into sections; and each section holds a piece of information of any data type. Any entity that "knows" the board's name can place information on it by specifying the name, the name of the section of the board that will store the information, and the actual information (e.g., "Target_Locations," "Enemy_Headquarters," location). Similarly, any entity that "knows" the board's name and the section's name can specify those names to read the information they store (e.g., "Target_Locations," "Enemy_Headquarters"). Blackboards are accessed via SimBionic's built-in actions and predicates, which are automatically included in every project.

7.1.1 Group Messaging

Group messaging allows an entity to join a named group of entities and receive every message directed at that group. It also allows any entity to send a message to a named group, regardless of whether the entity belongs to the group. Group messaging therefore allows any entity to communicate directly with a select group of other entities.

Every group message is sent to an entity's <u>message queue</u>, which is a personal storage area SimBionic automatically creates for each entity, and which holds onto each message until the entity is ready to access it, optionally read it, and discard it. Messages are accessed in first-in, first-out order—e.g., if an entity receives three messages, it must access and discard the first one before it can access the second one, and it must access and discard the second one before it can access the third one.

Each message contains three components an entity can read:

- **The ID of the entity who sent the message**, which is an Integer. This information can be used to provide context to a message—or determine whether the message needs to be read at all. (E.g., a soldier entity might be instructed to only read messages from its commander, in which case it would check the sender of each message and simply discard any message not from its commander.)
- **A number code associated with the message**, which is an Integer. SimBionic doesn't ascribe any meaning to this code, so what number is attached to a particular message and what that number represents are entirely up to you. For example, you might assign each message a priority number from 1 to 5, with 1 meaning *Urgent; read immediately!* and 5 meaning *Light news; read at your leisure*. Or you might assign each entity a unique number and then use the numbers to flag messages directed exclusively at a particular entity. Or you

might use number codes to indicate different ways a particular message should be interpreted (e.g., 1 might mean *Read as text string*, 2 might mean *Read as map coordinates*, 3 might mean, *Read as direction and velocity changes*, etc.).

- **The actual message**, which is of whatever data type you select. For example, you can choose to make all messages text strings; or you can opt to send messages consisting of a variety of data types and identify each type by a number code (as described in the previous paragraph).

For an entity to directly receive messages, it must join at least one named group. If the entity tries to join a group that doesn't exist, SimBionic will automatically create the group, using the name the entity specified and then make the entity the first member of that group. Any entity that subsequently joins will simply be added as another member of the group. As a result, a group's membership can be as small as one.

An entity doesn't have to be a member of a group to send a message to that group; it simply needs to "know" the name of the group. The advantage of belonging to a group is that an entity receives all the messages sent to that group. But if, say, a leader entity needs to send frequent messages to subordinate entities, it might opt to not join the subordinate's group, so that its message queue doesn't become cluttered with its own messages. Further, the leader might create another group consisting solely of itself, so that subordinates can send messages to the leader that only the leader can read.

There's no limit to the number of groups that can be created; the number of groups a particular entity can join; the number of entities that can belong to a particular group; or the number of messages that can be sent to a particular group.

7.1.2 Virtual BlackBoards

Virtual blackboards are storage areas available to all entities for sharing information. A blackboard is divided into sections; and each section holds a piece of information in the form of a Java object. Your entities can create an unlimited number of blackboards, and they can also create an unlimited number of sections within any board.

Once information is placed on a section of a board, it remains stored there unless an entity overwrites it by posting new information to that same section. The only other way to eliminate the information is for an entity to destroy the entire board that is storing it. (This is also useful for eliminating obsolete boards and freeing up memory.) An entity can also effectively "erase" a board by destroying it and then immediately creating a new, blank board with the same name.

7.2 Descriptors and Polymorphism

A *descriptor* is an identifier or attribute describing a possible state of an entity. A descriptor can represent a dynamic physical or mental attribute of an entity such as *happy*, *sad*, *healthy*, *armed*, or *hungry*. It can also refer to the static characteristic of an entity, such as its role on a team (e.g., *leader*, *scout*, *medic*) or its nationality (e.g., *Russian*, *Australian*, *American*).

Descriptors are hierarchical. Specifically, you first create a top-level descriptor, called a descriptor category, which represents a single aspect or "dimension" of an entity's state. You can then create second-level descriptors branching from that category. You can additionally create

third-level descriptors that branch from a second-level descriptor, fourth-level descriptors that branch from a third-level descriptor, and so on.
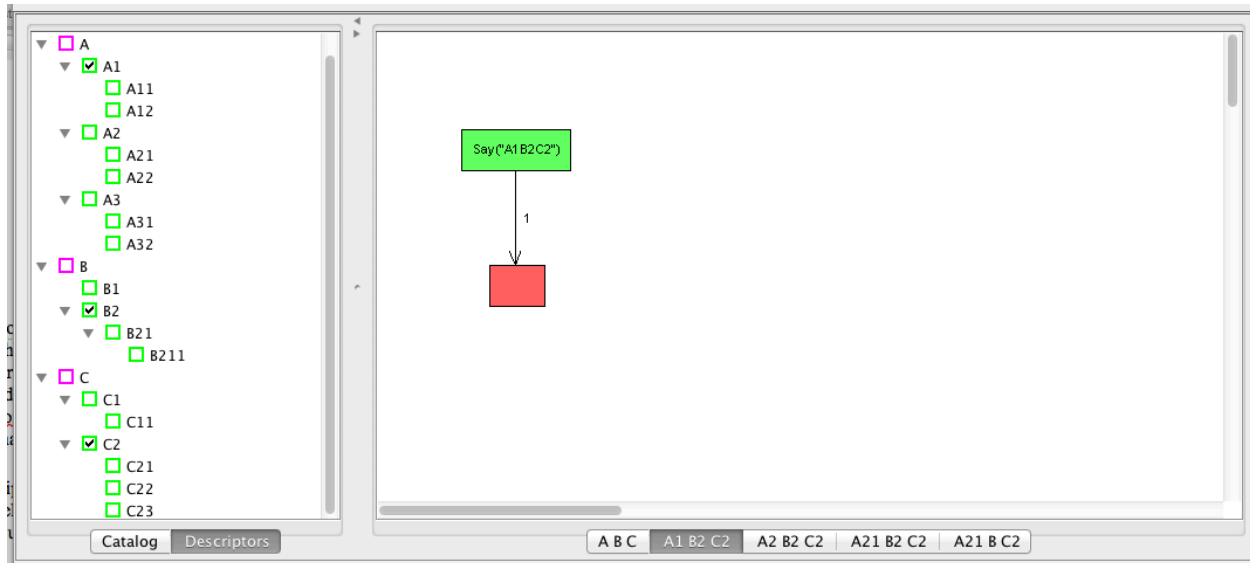

Figure 18. Example descriptors shown in the catalog on the left and multiple versions of the behavior shown on the right. The selected behavior will be selected when the global variables are set. To run the selected behavior, gA is set to "A1," gB is set to "B2," and gC is set to "C2."

You can associate descriptors with BTNs to define polymorphic BTNs. When your program runs, SimBionic will always look for the lowest-level descriptor you've specified that matches the entity's current state, in order to execute the most specific behavior available. If that descriptor has no behavior associated with it, however, SimBionic will then try to execute the behavior associated with the next-highest level in the descriptor's hierarchy. If that behavior doesn't exist either, SimBionic will again go to the next-highest level, until it reaches a descriptor in the category that does have a behavior associated with it.

Note: You should, at minimum, associate a behavior with any top-level descriptor or combination of top-level descriptors. This ensures that SimBionic will always be able to locate a behavior to execute regardless of an entity's descriptor status. Otherwise, SimBionic may occasionally be unable to locate a behavior to execute.

For every descriptor category you create, the SimBionic editor automatically adds a corresponding global variable of type string to the *Polymorphic* folder in the *Globals* list in the Catalog. These global variables (which you cannot delete or rename) are used to store the current state of an entity for each of the various descriptor categories. By default, each variable is set to the topmost descriptor in the hierarchy for the corresponding category.

When an entity invokes a behavior (by executing a behavior node or through the engine API), the run-time engine dynamically selects the polymorphism of that BTN that is most appropriate for the entity based on its current state. To determine what the current state of the entity is, the engine examines the values of the special global variables in the Polymorphic folder. You can therefore change an entity's state by simply binding a new value to one of these global variables. Keep in mind that the new value should also be one of the descriptors in the hierarchy for that category. Binding a non-descriptor value to these special global variables will cause errors when

that entity attempts to invoke a new behavior, since the run-time engine will be unable to find a matching polymorphism.

You should note that descriptors and polymorphisms could alternatively be implemented using ordinary global variables and lots of conditions and behavior rectangles. Under this approach, each time you invoked a behavior (e.g., Attack), you would need to check the value of the global "state" variables and then execute the behavior rectangle for the appropriate version of that behavior (e.g., Attack_Scout_Injured). The key differences between this approach and SimBionic's approach are:

- Descriptors are clearly associated with particular behaviors (because the bottom tab of the Canvas always displays the descriptors linked to the displayed behavior).
- Polymorphisms conceal the "machinery"—i.e., the extra conditions and behavior rectangles—needed to select and invoke the appropriate version of a behavior based on an entity's current state. They also neatly group all variations of a behavior in one place for easy comparison and editing.
- Descriptors are hierarchical, so even if there's no behavior that matches a specified combination of descriptors, SimBionic will be able to identify and execute a more generic version of the requested behavior.

These descriptors represent different states that your entities may attain as conditions change over the course of your simulation or game. These attribute-based descriptors can therefore help you create nuances and subtleties in the way your entities behave.

7.2.1   Descriptors Pane

The Descriptors pane displays the hierarchy of top-level descriptor categories and (lower-level) descriptors that have been defined for the project. In Figure 18, descriptor categories are displayed with violet checkboxes, and descriptors are displayed with green checkboxes. Checkboxes show the descriptor categories and descriptors that are associated with the polymorphism currently displayed in the Canvas.

Tip: If you want to temporarily create more room for other sections of the editor, such as the Canvas, you can hide the Project window by clicking the View menu and selecting the Project option (which turns the checkmark to its left *off*). To bring back the Project window, simply repeat this step; selecting the Project option again turns the checkmark to its left *on* and makes the window visible again.

Top-level Descriptor context menu:

Insert Descriptor Category	A top-level descriptor is a single aspect or "dimension" of an entity's state.

Descriptor context menu:

Insert Descriptor	Adds a new descriptor with a default name, such as *NewDescriptor*, to the list of descriptors that are children of the currently selected descriptor. The name of the descriptor is highlighted and editable.
Rename	Makes the name of the selected descriptor editable. To rename the descriptor,

	type the new name and then press *return*.
Delete	Deletes the selected descriptor.
Select Descriptor	Associates the descriptor with the current polymorphism. Removes any associates with (less specific) ancestors of the descriptor.
Move Up	Moves the descriptor up (earlier) in the descriptor folder.
Move Down	Moves the descriptor down (later) in the descriptor folder.

7.2.2 Polymorphism Tabs

SimBionic displays one or more tabs below the canvas. Each tab corresponds to a polymorphism for the currently selected behavior that is displayed in the canvas. To display the polymorphisms context menu, right-click in the area below the canvas. This context menu contains the following options:

Menu Option Label	Description
Insert polymorphism	Creates a new polymorphism for the current behavior.
Delete polymorphism	Deletes the selected polymorphism.
Duplicate polymorphism	Creates a new polymorphism for the current behavior that contains the same attributes as the selected polymorphism.

After you create or duplicate a polymorphism, use the Descriptors pane to select the descriptors associated with the polymorphism.

7.2.3 Example

As an example, the Hello World behavior can be changed to be polymorphic, where the BTN Main changes based on the mood of the entity.

First add a Descriptor Category called Mood on the Descriptor tab, followed by inserting two Descriptors: Good and Bad.
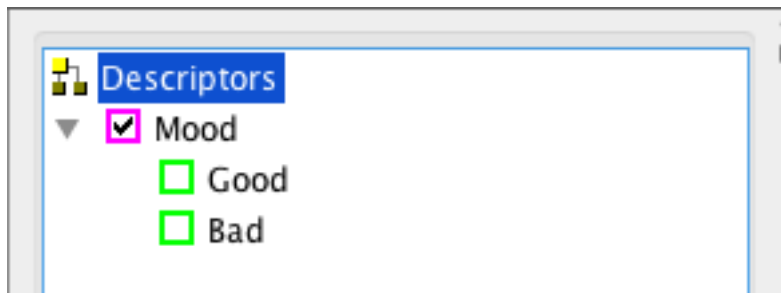
Creating the Descriptor Category Mood automatically creates a corresponding global variable called gMood. To set the descriptor, *gMood* is the variable that must be set either through a variable binding or by setting global variables via the engine API.
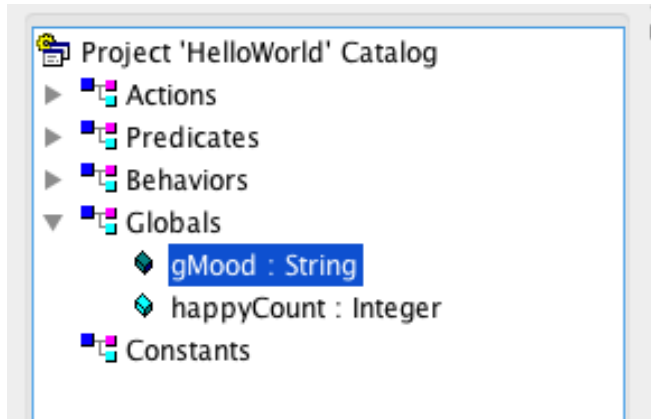
Second, right-click on Mood underneath the Canvas and select Duplicate Polymorphism. This will create a second copy of the Mood BTN as shown by the second button beneath the canvas reading Mood.
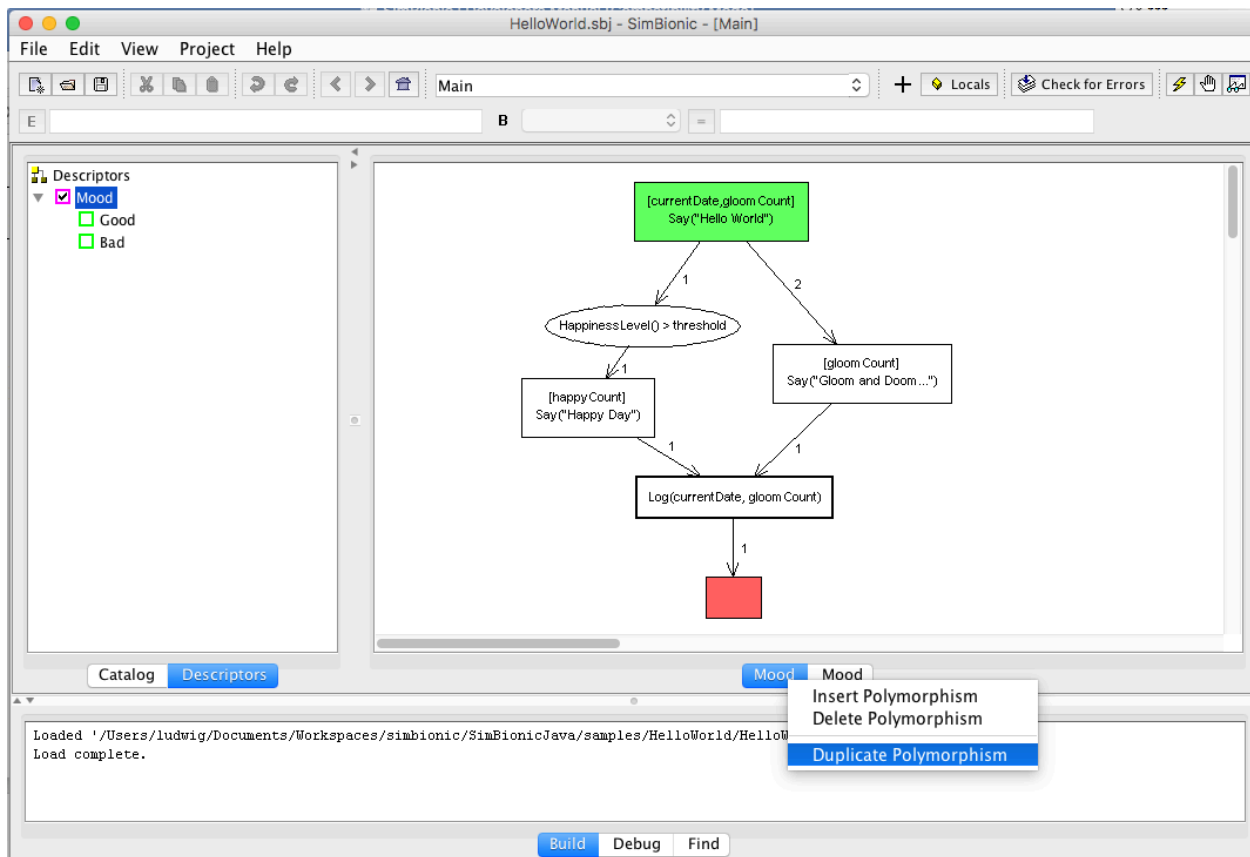
Third, select the second Mood polymorphism. Then click on the Good descriptor. We now have an alternate version of the Main behavior that will be invoked when Mood is set to Good.
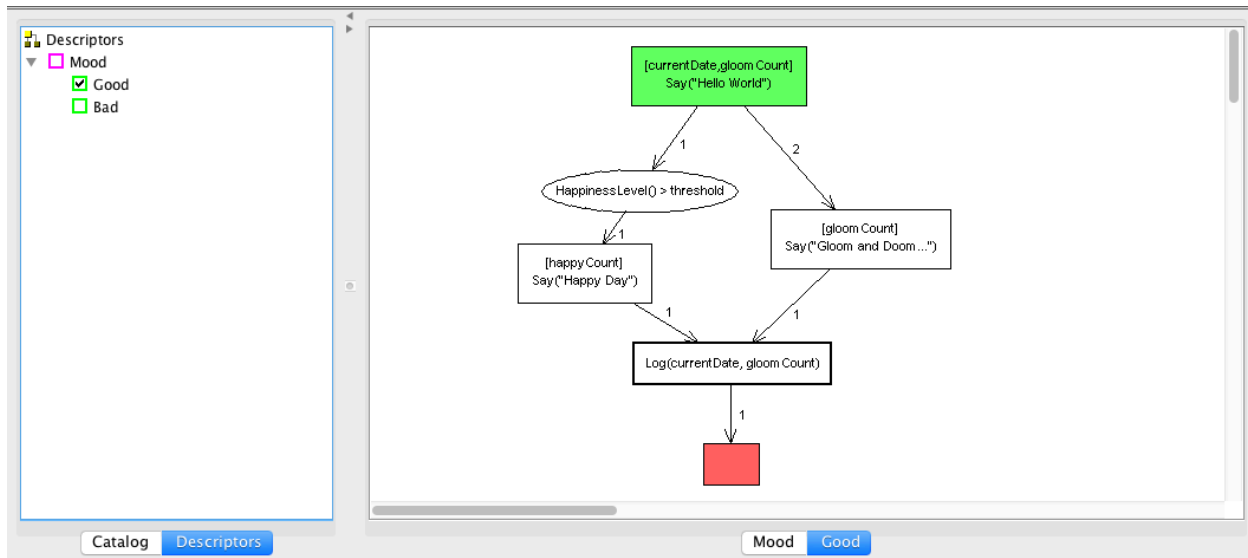
Fourth, update the Good version of the behavior. For example, two changes below update the Hello World print statement [Say("Hello World – it's a Good Day")] and make it so that HappinessLevel predicate will usually be true [HappinessLevel() > 1].
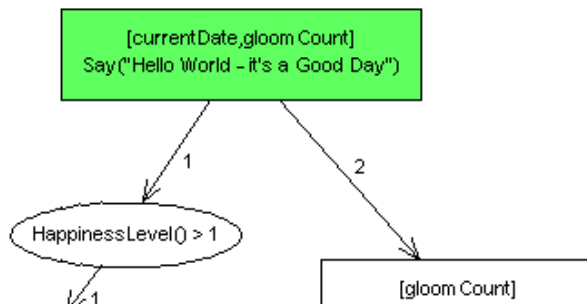
Fifth, if the Hello World test is run now, the default version of the Main BTN (Mood) will continue to be run. To see the Good version of the behavior, the global variable gMood must be set to "Good." Typically, a BTN would set the global variable gMood and then call a sub-behavior which would load the best version of the behavior. However, in this case, the Main behavior is the first BTN to be run, so the value of gMood must be set through the engine API before the Main behavior is loaded. To do this, the following two lines of code are inserted into the HelloWorld.java file:

```
// create an entity
long friendlyGuy = engine.makeEntity( "Friendly Guy" );
if (friendlyGuy == -1 )
{
  fail("Entity creation error: " + engine.getLastError());
  return;
}

SB_Param value = new SB_Param("Good");
engine.setEntityGlobal(friendlyGuy, "gMood", value);

// set Friendly Guy's initial behavior
String behavior = "Main";
```

7.3   Exception Handling

Sometimes errors occur in the underlying Java code that are called by an action node, compound action node, or predicate node. SimBionic provides a built-in facility, modeled after Java's exception-handling mechanism, for handling these run-time errors in a graceful and consistent fashion within your behaviors. To make use of this facility, you must ensure that your action and predicate code throws the special SimBionic exception SB_Exception when it encounters an error. SimBionic will not catch other kinds of exceptions. SimBionic provides two error-handling constructs and three core actions devoted to error recovery.

7.3.1   Catch Nodes

You can assign any action node, compound action node, or behavior node to be a **catch node.** A catch node in a behavior is like the catch block of a try-catch statement in Java. When a Java exception is thrown during the execution of a behavior containing a catch rectangle, the catch rectangle becomes the current rectangle for that behavior, and the execution mode for the behavior is set temporarily to non-interruptible (because error recovery should be treated as a critical section). Execution continues normally from that point, executing the catch rectangle and any subsequent rectangles (also known as a *catch block*). Typically the catch block is used to log and possibly clean up after the error, but you can do whatever you wish there. A behavior may have at most one catch rectangle.

Catch nodes are otherwise identical to normal rectangle nodes (action, compound action, and behavior nodes). Note that catch nodes are optional. Suppose that behavior A invokes behavior B, and behavior A has a catch rectangle, but B does not. If an exception is thrown by an action node in behavior B, behavior B will throw the exception to its invoking behavior, A, and pop itself from the stack. Behavior A's catch node will catch the exception, and execution will continue from that node.

If no catch node is provided in the behavior that originated the exception or in any of the ancestor behaviors, then the SimBionic *update* Java API method will return the constant *kFailure*, and the SimBionic *GetLastError* Java API method will return the message string for the exception.

Note: If a new exception is thrown during the execution of a catch block, the current exception's information is lost, and control returns to the catch rectangle at the start of the current catch block.

7.3.2   Always Nodes

The *always* rectangle is a special construct that allows you to ensure that a particular sequence of actions is always executed at the termination of a behavior, regardless of how that termination occurs. It is very similar to the finally block of a try-catch statement in Java. Whenever a behavior is about to be popped from the stack for any reason, the always rectangle becomes the current rectangle of the behavior, and the execution mode for the behavior is set temporarily to non-interruptible (because it should be treated as a critical section). Execution continues normally from that point, executing the always rectangle and any subsequent rectangles (known as an *always block*). A behavior may have at most one always rectangle.

Always rectangles are primarily useful for cleaning up any resources that might have been used by a behavior. Typical actions for an always block include deallocating client-allocated memory and relinquishing control over resources (e.g., files, devices). Note that there are a variety of ways in which a behavior can be popped from the stack, and the always rectangle will execute for all of them:

Normal Completion	The behavior has executed a final node.
Disruption	A non-interrupt connector in a behavior lower on the stack has been followed.
Unhandled Exception	An exception was thrown in the behavior or one of its child behaviors, and the behavior contains no catch rectangle.
Rethrown Exception	An exception was caught in this behavior, but the Rethrow action was invoked.
Cleared	The client application called a Java API method (e.g., SetBehavior) that cleared the entity's execution stack, including this behavior.

7.3.3   Recovery Actions

SimBionic provides three actions that can be used to handle exceptions in a graceful manner: Retry, Resume, and Rethrow. These actions can all be found in the Catalog action folder named Core Actions. They are described in Section 9.1.

8. Using the SimBionic Debugger

The SimBionic editor also performs as the debugger. There are three steps to starting the debugger. First, the debug jar file must be on the build path for both the editor and the application that is creating the SimBionic runtime engine. Second, the runtime engine needs to start with debugging enabled. Third, the authoring tool needs to connect to the waiting runtime engine.

8.1 Setting up the Build Path

The SimBionic distribution includes two jar files. To use the debugger, the dev version must be the one on the project build path or classpath for both the editor and the engine.

- SimBionic-3.x.x.jar – deployment jar file
- **SimBionic-dev-3.x.xjar – debugging jar file**

Alternately, if SimBionic is being run from source files, then the following two variables must be set in SIM_Constants to enable debugging:

- **public static boolean *DEBUG_INFO_ON* = true;**
- **public static final boolean *AI_DEBUGGER* = true;**

8.2 Setup the Runtime Configuration

To enable the debugger for the runtime, set the following items in the SB_Config. For example, in HelloWorld.java, setting myConfig.debugEnabled to true will cause the runtime engine to try to connect to the Authoring tool at startup.

```
SB_Config myConfig = new SB_Config();
myConfig.debugConnectTimeout = 90;
myConfig.debugEnabled = true;
…
if (engine.initialize(myConfig) != SB_Error.kOK)
{
  fail("Engine failed to initialize: " + engine.getLastError());
  return;
}
```

8.3 Connecting to the Engine to the Visual Debugger

Once the code is set up as shown, the steps are:

1. Run the authoring tool and open the behavior file that will be executed. Set breakpoints in the behaviors as needed.

2. Start the runtime engine, which will wait for the debugger to connect for the number of seconds provided in the SB_Config (90 seconds in this example).

3. Select **Project**, then **Connect to the Execution Engine** in the authoring tool.

4. At this point, the debugger view will be shown.
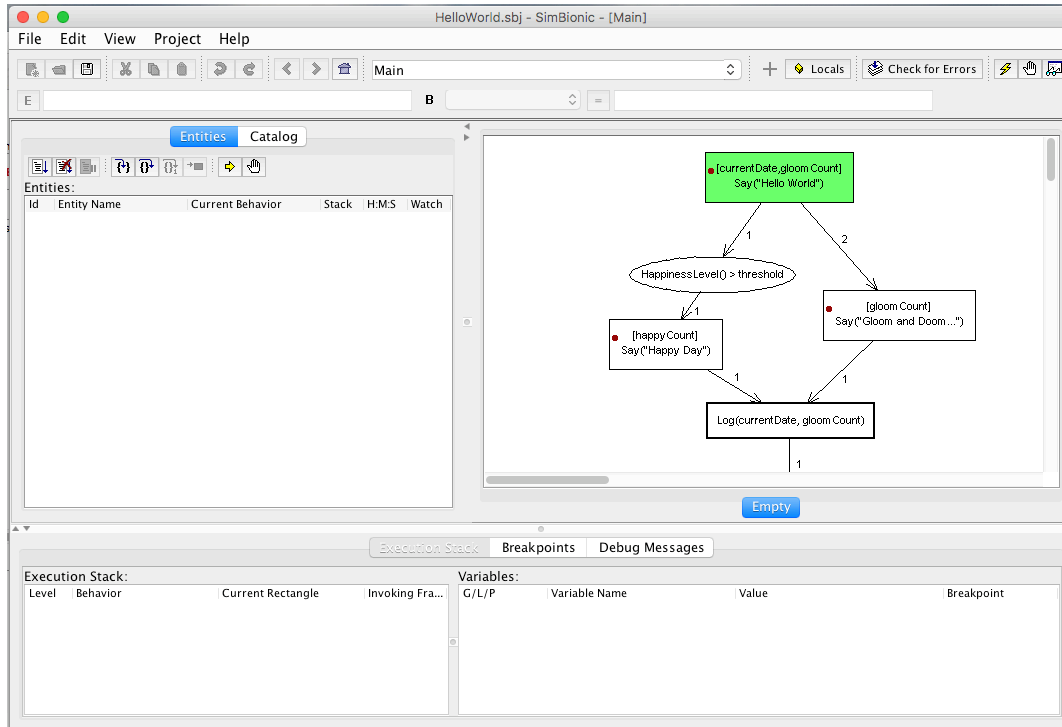
Figure 19. Step 1 - Load in the file and set breakpoints.

8.4   Using the Debugger

When the debugger is activated, new windows will be available as shown in Figure 19. An Entities tab will share space with the Catalog in the upper left. The bottom portion of the UI is replaced with the Execution Stack Tab, Breakpoints Tab, and Debug Message Tab.

8.4.1   Entities Window
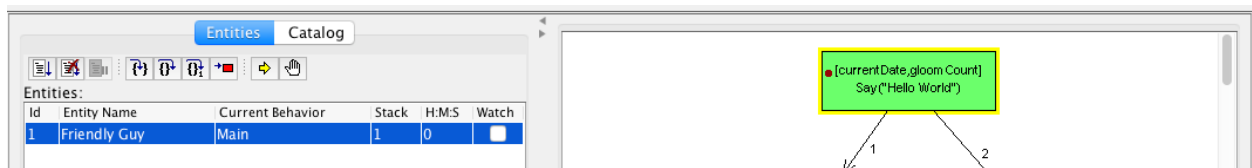
Figure 20. Entities window in the debugger.

The toolbar at the top of the Entities Window allows you to start, stop, and pause behavior execution. It also allows you to step into the current node, over the current node, one tick, or run to the final node in the behavior. Figure 20 illustrates that the Main behavior in Hello World has been started and is currently at the highlighted initial node.

The window also includes a list of all the entities currently being run within the SimBionic engine. The window devotes a row to each entity and provides data about the entity in the following six columns:

- **ID:** Displays the entity's ID number. This number is assigned on the fly by SimBionic's run-time engine when your program creates the entity via the CreateEntity or MakeEntity API methods.

39

- **Entity Name:** Displays the text name you optionally assigned to the entity via the CreateEntity or MakeEntity API methods. If you skipped creating a text name for the entity, this cell is blank.
- **Current Behavior:** Displays the text name of the behavior (i.e., the name you declared in the Catalog and which is displayed on the Canvas) that the entity is currently executing.
- **Stack:** Displays the level of the frame currently being executed on the entity's stack. For example, if the entity has just been created, the stack level will be 1, because the entity will be in the bottommost frame of the stack; but the first time a behavior is invoked, the level will be 2, because the invoked behavior will cause a second frame to be placed onto the stack and executed.
- **H:M:S:** Displays the number of hours, minutes, and seconds the entity has been executing.

8.4.2   Execution Stack Tab

The Execution Stack window in this tab within the interactive debugger is a visual representation of the execution stack that SimBionic's run-time engine has assigned to the entity that is currently selected in the Entities window. Every entity is automatically assigned its own individual stack to track its particular behaviors.

Every behavior that's run for the entity is implemented by an execution frame that the engine pushes onto the entity's stack. Therefore, this window devotes a row to each execution frame, and provides data about the frame via the following five columns:

- **Level:** Displays the level of the execution frame on the stack. The first behavior will be level 1, the next invoked behavior will be level 2, and so on.
- **Behavior:** Displays the name of the behavior being run via the execution frame.
- **Current Rectangle:** Displays the name of the action or behavior rectangle currently being executed in the behavior.
- **Invoking Frame:** Displays the stack level of the behavior that invoked the current behavior. In almost all cases, this number will be one less than the current frame—i.e., the invoking behavior will be directly under the current behavior on the stack. The one exception is when the current frame is an interrupt behavior, in which case the invoking frame may be at any level below the current one. To help you easily pick out these exceptions to the rule, this window displays the text representing interrupt behaviors in cyan (as opposed to standard black text). *Note:* The initial behavior will have a dash (-) instead of a number in this column, because it wasn't invoked by any previous behavior.

Note: Selecting a row (i.e., execution frame) in the Execution stack window will cause the Variables Windows and Canvas to update to show the current behavior in that frame as well as its local variables and parameters.
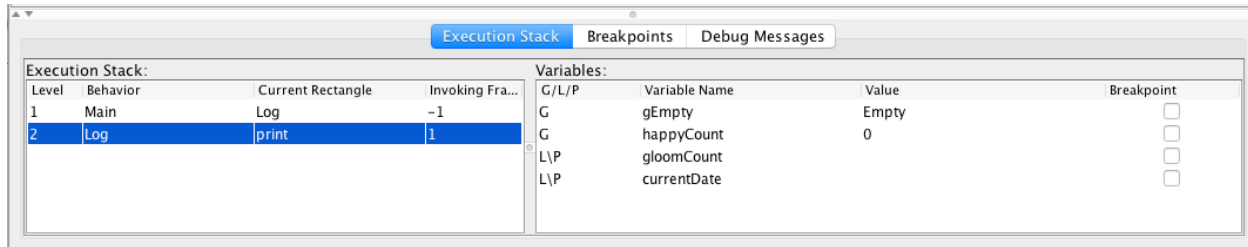
Figure 21. Execution Stack and Variables display for HelloWorld, with a breakpoint set at the initial node in the Log sub-behavior.

The Variables window in the Execution Stack tab allows you to track your program's use of global variables, local variables, and behavior parameters. The window devotes a row to each variable or parameter, and provides data about the variable or parameter via the following five columns:

- **Global/Local/Parameter:** Displays the letter *G* if the item is a global variable, *L* if it's a local variable, or *P* if it's a behavior parameter.

- **Variable Name:** Displays the text name you declared for the variable or parameter.

- **Type:** Displays the data type you declared for the variable or parameter. Specifically, displays the letter *I* for integer, *F* for float, *S* for string, *V* for vector, *B* for Boolean, *E* for entity, *A* for array, *T* for table, *D* for data, and INV for Invalid. (For more information, see the Choosing a data type topic.)

- **Value:** Displays the value currently stored in the variable or parameter. If the value is longer than the column width, displays the initial data followed by an ellipsis (…) to let you know that you can view more of the value by widening the column (by clicking and dragging a column border).

- **Breakpoint:** Allows you to set a breakpoint that pauses your program whenever the value in the variable or parameter changes.

8.4.3   Breakpoints Tab

The Breakpoint List window allows you to track and control your use of breakpoints in the interactive debugger. The window devotes a row to each breakpoint you've created, and provides data about the breakpoint or parameter via the following five columns:

- **Breakpoint:** Displays the name of both the behavior and the Canvas element or Catalog item to which the breakpoint is attached.

- **Entity:** Allows you to set whether the breakpoint applies to All entities (represented by *A*, which is the default); or just the entities in the Entities window that you've set to be Watched (represented by *W*); or just one particular entity (represented by that entity's ID number).

  - To use this feature, click in the Entity cell of the breakpoint you want to adjust. A text box appears that lets you type an entity ID number (e.g., 0, 1, 12) or W or A.

    - NOTE: This does not work in the current implementation. Once an entity ID number is selected, it cannot be changed back to A or W.

- **#:** Allows you to set how many times your program must flow to the breakpoint before the breakpoint actually pauses execution. The default setting is 0, meaning the breakpoint will pause execution every time your program encounters it.

  - For example, if you only wanted to pause execution every other time the breakpoint was reached, you'd set this value to 1; if you wanted to pause every third time the breakpoint was reached, you'd set this value to 2; and so on.
  - To use this feature, click in the # cell of the breakpoint you want to adjust. A text box appears that lets you type a whole number (e.g., 1, 2, 12). Type the number of times you want the breakpoint to allow execution to proceed before it's triggered, and then click outside of the text box to save your setting.

- **Constraint:** Allows you to enter an expression that evaluates as true or false (e.g., *Count>5*, or *X==Y*). The breakpoint will only be active when the expression you specify evaluates as true. The default is to specify no expression, which places no constraint on the breakpoint being active. To use this feature, click in the Constraint cell of the breakpoint you want to adjust. A text box appears that lets you type any expression that evaluates to true or false. Type the expression and then click outside of the text box to save it.

- **Enabled:** Allows you to turn the breakpoint off. This is useful if you don't want to remove the breakpoint but just temporarily disable it, allowing execution to flow past the breakpoint without pausing.

8.4.4   Debug Message Tab

This tab contains low-level debug information, primarily of use to developers maintaining SimBionic.

9. SimBionic Actions, Predicates, and Standard Variable Types

This section summarizes predefined actions and predicates.

9.1 Predefined Actions - Core Actions Folder

DestroyEntity	Destroys the entity that is identified by its entity ID.
PushBehavior	Pushes the specified behavior onto the entity's behavior stack. This is an advanced API method. The method SetBehavior is more typically used.
SetBehavior	Sets the specified entity's current base-level behavior, initializing it with the specified parameters. The polymorphism for the behavior is chosen based on the current values of the entity's global variables.
SetEntityGlobal	Sets a global variable to the specified value for the given entity.
SetUpdateFrequency	Sets the update frequency for the specified entity. Lower-frequency numbers will be updated more frequently than higher-frequency numbers, with zero being updated on each tick. Entities with negative frequencies are placed on hold and will not be updated until their frequencies change.
SetUpdatePriority	Sets the update priority for the specified entity. This priority is used to determine the order in which entities are updated within a single update. Entities with lower-priority values will be updated before those with higher values.

This folder also contains actions for exception handling:

Retry	Causes behavior execution to jump to the rectangle where the exception was thrown *in this behavior*, executing that rectangle exactly as if it had just become the current rectangle in normal fashion—that is, the bindings for that rectangle will be evaluated, and then the action or behavior will be invoked. Note that this behavior might not be the one that threw the original exception (because that behavior may have been popped off the stack). If this is the case, then the rectangle that *invoked* the behavior that threw the original exception will be retried. Retry is generally invoked after attempting to diagnose and fix whatever error condition caused the exception to be thrown, in the hopes that the second attempt will be more successful.
Resume	Causes execution to resume at the rectangle *in this behavior* where the exception was thrown, without attempting to retry execution of that rectangle. That rectangle becomes the new current rectangle, but its bindings are not evaluated and its action or behavior expression is not executed. However, the conditions leading out of the rectangle will be evaluated normally. Resume is typically invoked when it is not possible to fix the error condition (so retrying the offending action or predicate would be fruitless), but it is still possible to clean up after the error and continue with the behavior's normal course of execution.

Rethrow	Rethrows the current exception down the stack to the current behavior's invoking behavior and then terminates the current behavior just as if a final rectangle had been executed. The invoking behavior then becomes the new current behavior and must attempt to handle the exception exactly as if it had been thrown in that behavior. Rethrow is called when a behavior is unable to recover from an error and needs to pass responsibility for error recovery on to another behavior.

9.2   Predefined Actions – Messages Folder

DestroyGroup	Deletes the message group, identified by its group name.
JoinGroup	Enrolls this entity in the named group. If that group does not exist, it is created.
Resume	Causes execution to resume at the node where the exception was thrown without attempting to retry execution of that node.
NextMsg	Discards the topmost message in the queue.
QuitGroup	Removes this entity from the named group.
SendMsg	Sends a message to the specified group, where a message consists of a numerical type and an arbitrary variable value.

9.3   Predefined Actions – Blackboards Folder

CreateBBoard	Creates a Blackboard with the provided name.
DestroyBBoard	Destroys named blackboard.
PostBBoard	Writes (key, value) to the named blackboard.

9.4   Predefined Predicates – Core Predicates Folder

CreateEntity	Destroys the entity that is identified by its entity ID.
GetEntityID	Returns the integer entity ID of the current entity.
GetEntityName	Returns the SimBionic name of the current entity.
IsDone	Indicates whether an invoked behavior has completed execution. Returns true if behavior is finished, false otherwise. The typical use case is as part of a condition on a connector out of a behavior rectangle.
IsEntityFinished	Returns true if the specified entity, identified by entity ID, currently is not executing any behaviors, and returns false if otherwise. Also returns false if the entity does not exist.

9.5   Predefined Predicates – Messages Folder

GetMsgData	Reads the message at the top of the entity's message queue. The message is unaffected by being read, and will remain stored in the queue until the entity discards it (via the NextMsg action) to access the next message in the queue.
GetMsgSender	Identifies which entity sent the message at the top of its message queue.
GetMsgType	Returns the number code that's attached to the current message in its message queue. The meaning of the type is author-defined in the behaviors. For example, the behavior may handle message type = 1 differently than message type = 2.
HasMsg	Checks whether the entity currently has any messages stored in its message queue.
NumMembers	Returns the number of entities that belong to a named group.

9.6   Predefined Predicates – Blackboards Folder

IsBboard	Verifies a named blackboard exists—i.e., was created previously via the CreateBBoard action and hasn't been permanently eradicated via the DestroyBBoard action
ReadBBoard	Accesses any blackboard and reads any information on the board (placed previously via the PostBBoard action) by specifying the board's name and the section of the board storing the information. ReadBBoard therefore allows any entity to obtain information from any other entity in your program.

9.7   Standard Variable Types

String	A Java String.
Boolean	A Java Boolean.
Integer	A Java Integer.
Float	A Java *Float*.
Vector	A Java Vector<Object>.
Table	The *table* data type is used to store a two-dimensional array. While you can implement a table via the array data type, using the table data type is more convenient when your information is logically organized into columns and rows. See the Javadoc or Java code for: *com.stottlerhenke.simbionic.common.Table*.
Object	A Java Object.
ArrayList	A Java ArrayList<Object>.

10.   SimBionic Java API

SimBionic's application program interface, or *API*, provides you with a range of API methods that let you easily control the functioning of the run-time engine via your Java code. These methods allow you to perform such fundamental tasks as start up the engine, create entities, set the frequency and priority with which entities are updated, select behaviors, set the values of global variables, retrieve error messages, insert comments to be recorded along with the engine's activities in a log file, and shut down the engine.

The API methods for the Java run-time engine are as follows:

- **Administrative methods:** initialize, getLastError, getVersion, swapProject, terminate
- **Communication methods:** sendMsg, createBBoard, destroyBB, postBBoard, readBBoard
- **Entity management methods:** createEntity, destroyEntity, getEntityGlobal, isEntityFinished, makeEntity, setBehavior, setEntityGlobal, setUpdateFreq, setUpdatePriority
- **Execution control methods:** update, updateEntity
- **Logging methods:** log, registerLogPrintStream

Controlling the entity scheduler is described in more detail below. For more information on the rest of the engine API, review the SimBionic Javadocs or Java code for SB_Engine and other classes in the package: com.stottlerhenke.simbionic.api.

10.1 Controlling the entity scheduler

Each time your application calls SimBionic's <u>Update</u> API method, the run-time engine consults its built-in scheduler to determine which entities should be updated on the current clock tick. Because updating an entity uses a bit of valuable CPU time, SimBionic's scheduler attempts to evenly distribute these updates across multiple clock ticks. This *load-balancing* process ensures that the run-time engine maintains a steady level of CPU usage, without peaks or valleys that might adversely affect the performance of your application.

You can control how a particular entity is scheduled via two parameters: its *update frequency* and its *update priority*. You can set these parameters when the entity is initially created, or you can change them later using the <u>setUpdateFreq</u> and <u>setUpdatePriority</u> API methods or core actions.

- **Update frequency:** This is the minimum frequency with which you want the entity updated, specified via an integer from -1 to 100. A setting of 0 tells the engine to update the entity precisely once every clock tick, and is the *default*. Higher number settings tell the engine that it can update the entity less frequently; e.g., a setting of 1 means to update at least every other clock tick, a setting of 2 means to update at least every third clock tick, a setting of 3 means to update at least every fourth clock tick, and so on. If it has the resources available, the engine will update an entity more frequently than the setting you've specified; but it will never update less frequently. Alternatively, if you want to temporarily prevent an entity from executing, a setting of -1 bars the entity from performing additional behaviors during *any* clock tick, effectively freezing it until you change its update frequency back to a positive number.
- **Update priority:** This is the execution priority you want to assign the entity *within* any given

clock tick, specified via an integer from 0 on up through the highest integer allowed by your compiler. For example, a setting of 0 (which is the *default*) tells the engine to execute the entity's behavior before entities scheduled to execute during the same clock tick that have settings of 1 or higher; a setting of 1 means to execute the entity before entities scheduled to execute during the same clock tick that have settings of 2 or higher; and so on.

For example, suppose you create four entities:

- **A** has update frequency 0 and update priority 3.
- **B** has update frequency 0 and update priority 2.
- **C** has update frequency 1 and update priority 1.
- **D** has update frequency 1 and update priority 4.

The run-time engine's scheduler will schedule these entities as follows:

- **Clock tick 1:** entity C, then entity B, then entity A
- **Clock tick 2:** entity B, then entity A, then entity D
- **Clock tick 3:** entity C, then entity B, then entity A
- …

Note: You can change the maximum value allowed for an entity's update frequency by setting the value of the maxUpdatePeriod variable in the configuration object you pass to the run-time engine on startup. The default value is 100.