

REPAST STATECHARTS GUIDE

JONATHAN OZIK, NICK COLLIER - REPAST DEVELOPMENT TEAM

0. BEFORE WE GET STARTED

Before we can do anything with Repast Symphony, we need to make sure that we have a proper installation of the latest version. See the [Repast Requirements Web Page](#) for instructions on downloading and installing Repast Symphony and Java.¹

1. GETTING STARTED WITH STATECHARTS

Agent states and transitions between states are an important abstraction in agent-based modeling. While it is possible for Repast Symphony users to create their own implementation of state-based agent behaviors (e.g., by adapting the State pattern in Gamma et al. 1994²) and even agent state visualizations, the effort involved in doing so is usually prohibitive. By integrating an agent statecharts framework into Repast Symphony, we made it easy for users of all levels to take advantage of this important modeling paradigm. Statecharts are visual representations of states and the transitions between those states³. Statecharts can be very effective in visually capturing the logic within agents and quickly conveying the underlying dynamics of complex models.

Figure 1 shows a simple example of a statechart created with the Repast Symphony statecharts framework. The logic embedded in the diagram is mapped directly to the execution logic of an agent-based model. The benefits of the Repast Symphony statecharts framework include: improved clarity of a model's logic for model design, improved turnaround times for developing complex state based agent models, and the ability to convey in a compelling manner the internal state of agents as a simulation evolves to both experienced agent-modelers and to non-modelers alike. In the rest of this guide we will present the Repast Symphony statecharts framework.

Date: November 13, 2018.

¹ <https://repast.github.io/requirements.html>

²Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. illustrated ed. Addison-Wesley Professional, 1994.

³Statecharts were first proposed by Harel in Harel, D., 1987. Statecharts: A visual formalism for complex systems. Sci. Comput. Program., 8(3), pp.231-274.

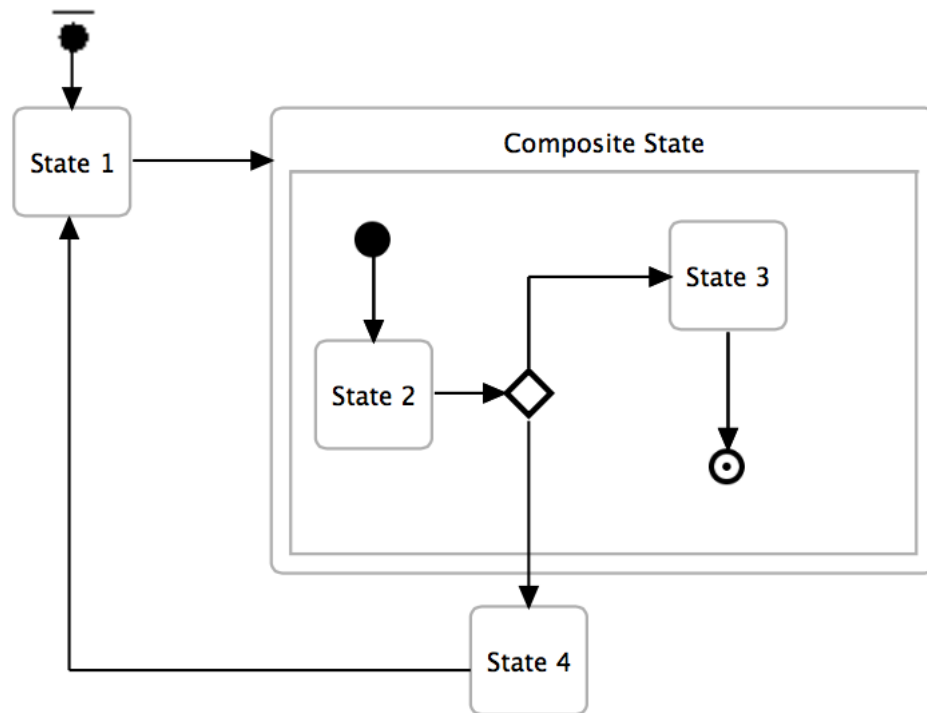


FIGURE 1. An example statechart created with the Repast Symphony visual statecharts editor.

1.1. Adding Statecharts. A statechart can be added to any Java, Groovy or ReLogo class⁴. Right-clicking the class of interest and selecting New → Statechart Diagram (Figure 2) will bring up the new statechart wizard (Figure 3). The editable new statechart wizard elements are:

File Name: This is the .rsc statechart file name that will be edited with the visual statecharts editor.

Name: The display name of the statechart.

Class Name: The name of the statechart class which will be generated.

Package: The package name within the `src-gen` source folder where the statechart class source code will be generated.

Agent Class: This is the agent class that will be associated with the statechart.

⁴For example, any of the agent classes that are created in the other Repast Symphony Getting Started guides can have statecharts added to them.

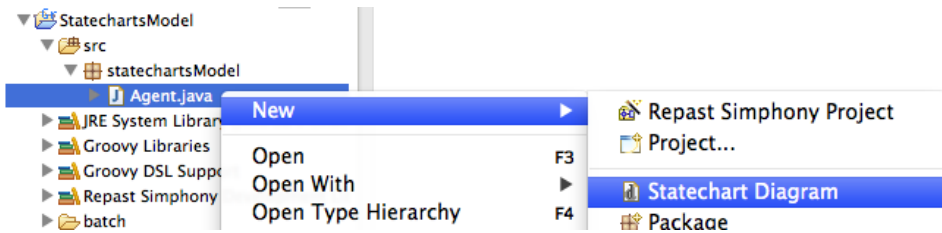


FIGURE 2. Creating a new statechart.

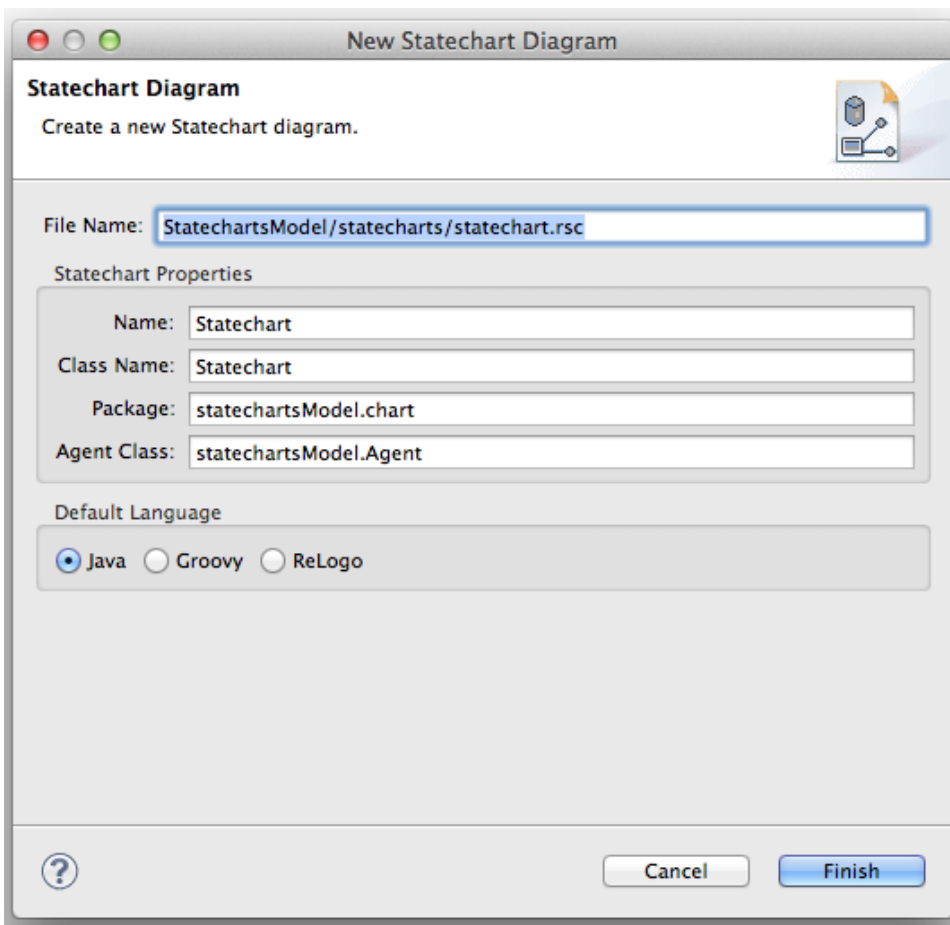


FIGURE 3. New statechart wizard.

After accepting or modifying the defaults, click the *Finish* button. This injects a new statechart field into the selected agent class (see Listing 1). The `@ProbedProperty` annotation (Line 8 in Listing 1) connects the *Name* element of the statechart to the display name that will be shown in the probe panel in the simulation runtime GUI (Section 5). Line 9 in Listing 1 calls the `Statechart.createStateChart(Agent, double)` method with 0 as the second argument. This will schedule an `Agent` class's statechart to begin as soon as the `Agent` instance is created. Alternatively, you can specify a numerical value greater than 0 (in units of simulation *ticks*) to delay the statechart activation for that number of ticks. If there is a need for a statechart to begin not at a predetermined time but, for example, based on some model logic, Line 9 would be modified to read:

```
Statechart statechart = Statechart.createStateChart(this);
```

which just instantiates the statechart but doesn't schedule it to begin at any time. Then, when the conditions for the statechart to begin are met, the statechart can be scheduled to begin with:

```
StateChartScheduler.beginNow(statechart);
```

for immediate scheduling or:

```
StateChartScheduler.beginLater(later, statechart);
```

for delayed scheduling at a time `later`. The statechart field does not have to be instantiated upon agent class creation. So if there is a situation when not all of a simulation's agents will be needing their statecharts and there is a desire for conserving the simulation's memory footprint, the statechart instantiation (i.e., `Statechart.createStateChart`) can be delayed until needed.

When a statechart is no longer needed, it can be stopped via:

```
statechart.stop();
```

This will deactivate all scheduled transition resolution activities. If there is a need for memory conservation, simply doing:

```
statechart = null;
```

after stopping the statechart, will remove the reference to the statechart, freeing it up for garbage collection.

```
1 package statechartsModel;
2
3 import statechartsModel.chart.Statechart;
4 import repast.simphony.ui.probe.ProbedProperty;
5
6 public class Agent {
7
8     @ProbedProperty(displayName="Statechart")
9     Statechart statechart = Statechart.createStateChart(this, 0);
10
11 }
```

LISTING 1. Agent class with the default injected statechart field.

1.1.1. *A note on statechart scheduling.* Statecharts have two associated scheduling behaviors. The first is the one described above which deals with when a statechart should begin being active. If a time tick is specified, a statechart is scheduled to be active at the time tick with *highest* model action priority. That is, the statechart is activated before any other scheduled model behaviors are run. In this way all scheduled actions that occur during a time tick occur after the statechart is active.

The second, and more prevalent scheduling activity, is that which governs the resolving of statechart triggers. This dictates when a statechart will check for transition conditions and, consequently, when associated state changes can occur. This resolve scheduling occurs with the *lowest* model action priority, meaning that all other scheduled activities for a particular time tick will run before any statechart resolutions occur. This allows any state based activities to depend on the model state after all other possible model changes have already occurred.

1.2. Statecharts Editor. Creating a new statechart will bring up the newly created blank statechart editor, which we describe next. The statecharts editor is divided into three main panels (Figure 4). The first (Figure 4a) is the statecharts workspace area. This is where the visual elements of a statechart are created and arranged. The palette panel (Figure 4b) shows the available statechart elements that can be used in the statecharts workspace⁵. Clicking on an element in the palette panel and then clicking on the statecharts workspace will create an instance of that element in the workspace. To create transitions between elements, you can click on the Transition arrow in the palette and then click and drag from a source element to a target element. The properties panel (Figure 4c) shows the properties of the element selected in the statecharts workspace⁶. If, as in Figure 4, no element is selected, the properties of the statechart itself are shown. In addition to displaying element properties, the panel is also where the properties of elements can be edited. A statechart has a priority which indicates the order in which it will be resolved with respect to other statecharts (see red box in Figure 5). So, for example, if an agent has two statecharts (A and B) and statechart A should be resolved before statechart B, giving statechart A a higher priority will ensure that this occurs.

There is a contextual menu approach for adding elements to the workspace as well. Simply hovering over an area in the workspace will reveal a contextual menu of the available elements appropriate for the region pointed to. If the mouse pointer is on an empty space in the workspace background, you will see the contextual menu in Figure 6. If the pointer is on a state, you will see transitions shortcuts like in Figure 7, the left symbol indicating a connection to this state and the right symbol a connection from this state. Finally, if the pointer is inside a composite state⁷, you will see the contextual menu in Figure 8.

⁵Sections 2 and 3 will cover these elements in detail.

⁶If the properties panel is not showing, right-click on the canvas and choose “Show Properties view” or double-click on any statechart element.

⁷A composite state is a state that contains other states.

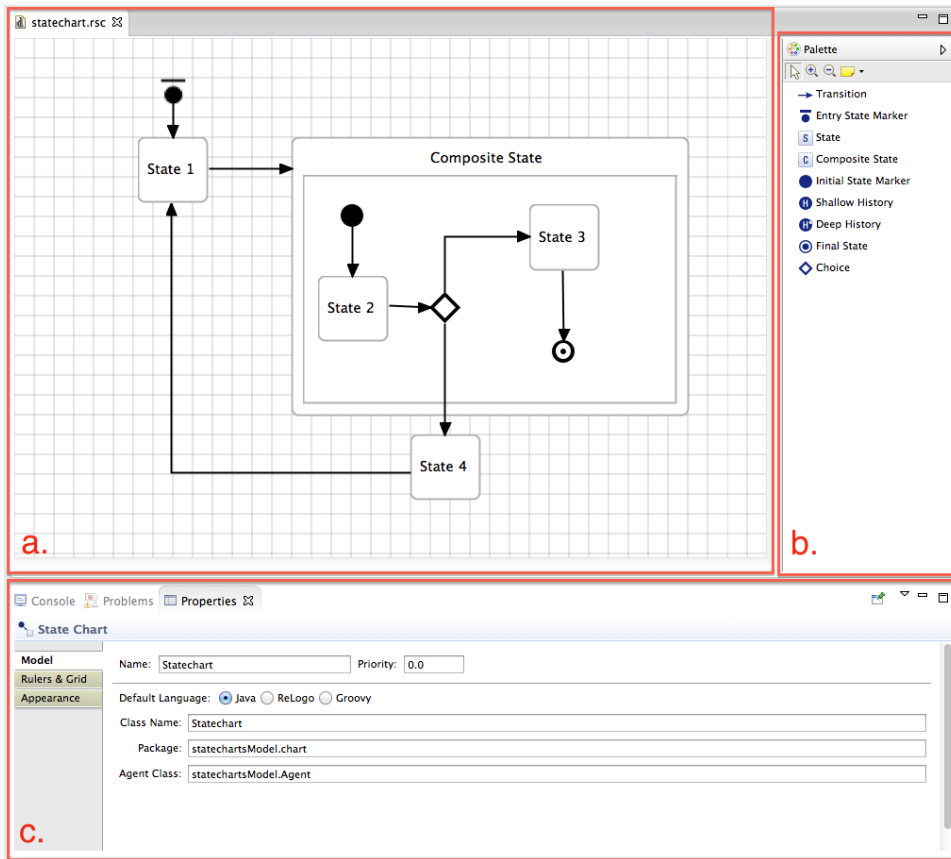


FIGURE 4. The components of the statecharts visual editor. a) The workspace area where the statechart elements are created and arranged. b) The palette panel of available elements, including selection, zoom and note tools. c) The properties panel of the selected statechart element in the workspace (a). If no element is selected, the properties of the statechart itself are displayed.

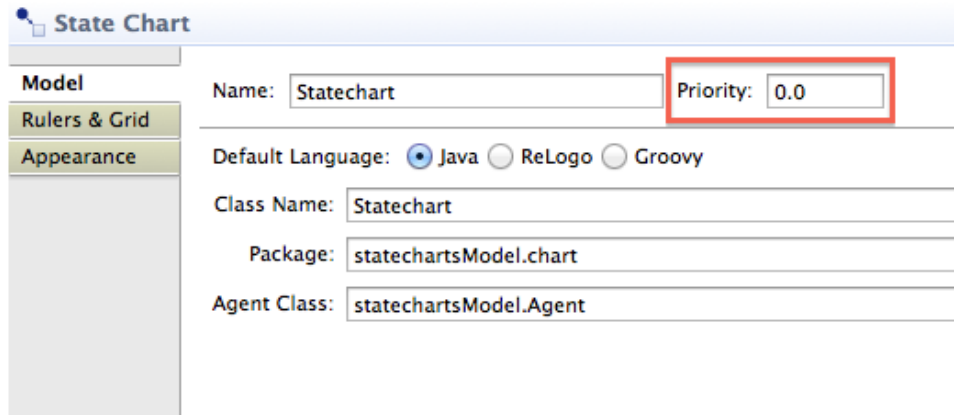


FIGURE 5. The statechart properties panel with the priority element indicated by a red box.

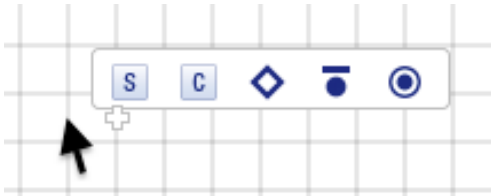


FIGURE 6. The default contextual menu when the pointer is on a blank area of the workspace.

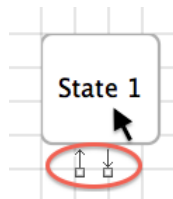


FIGURE 7. The transitions shortcuts, circled in red, for making connections to (left) and from (right) the state.

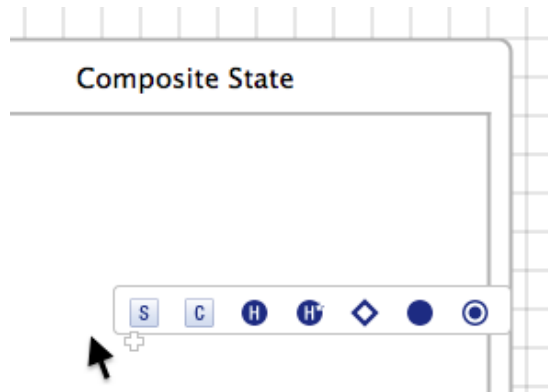


FIGURE 8. The contextual menu when the pointer is inside a composite state.

2. STATES

One of the fundamental building blocks of statecharts are states. Here we introduce the different types of states that exist within the Repast Symphony statecharts framework.

2.1. Entry State Marker.

Every statechart must have an entry state marker. This defines point of entry into a statechart takes when the statechart is activated.

2.2. Simple State.

A simple state looks like Figure 9. At any one point in time within an active statechart, one and only one of the simple states will be active. In addition to their *ID*, simple states can have *On Enter* and *On Exit* actions defined, as seen in the simple state properties panel in Figure 10. These actions are triggered when entering or exiting the simple state, respectively. The keywords available within the two action blocks are:

- agent:** This is the agent that contains the statechart. Any method (e.g., `customMethod`) defined on the agent can be invoked through this reference (e.g., `agent.customMethod()`).
- state:** This is the state itself. For example, the state's *ID* can be accessed via `state.getId()`.
- params:** This is the model's `Parameters` object. As an example, a double valued parameter `dParam` can be retrieved with: `params.getDouble("dParam")`⁸.

The action block editor supports syntax highlighting, code suggestion and auto completion. So, for example, entering "agent." in the editor will trigger a pop-up window displaying all the methods defined on that agent. Figure 10 illustrates this pop-up behavior when invoked on an example agent. The pop-up window can also be triggered via a CTRL+Space key shortcut.

As is the case with all types of action blocks, their logic can be specified using Java, Groovy or ReLogo. All action blocks also statically import `repast.simphony.essentials.RepastEssentials` and `repast.simphony.random.RandomHelper`, making the many convenience methods directly callable (i.e., `nextDouble()` instead of `RandomHelper.nextDouble()`). If additional imports are required, these can be specified in the imports tab. Imports should be specified one-per line, ending with a semi-colon. Specifically, any Java, Groovy or ReLogo code can be used to express the behavior that should be executed upon entry to or exit from the state⁹.

⁸See the source or JavaDoc for `repast.simphony.parameter.Parameters` for all of the available methods.

⁹When using the ReLogo option, the `agent` parameter is implicit so writing `customMethod()` is equivalent to `agent.customMethod()`.

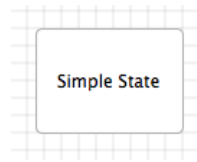


FIGURE 9. Simple state.

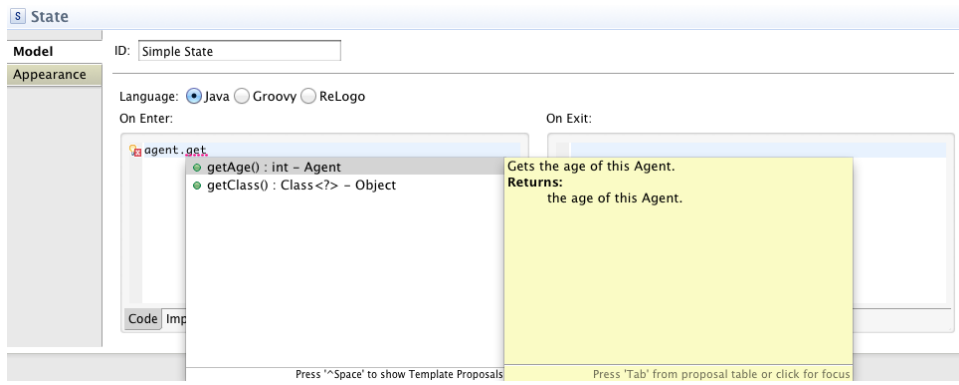


FIGURE 10. Simple state properties.

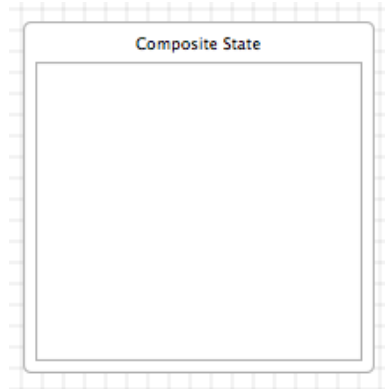


FIGURE 11. Composite state.

2.3. Composite State. C

Composite states are used to nest elements within a statechart. Figure 11 shows an empty composite state and Figure 12 shows the properties panel for composite states, which is identical to that of the simple states in that *On Enter* and *On Exit* actions can be defined. The difference between composite and simple states lies in the fact that composite states can include the following elements as sub-elements:

- Simple state (Section 2.2)
- Composite state (Section 2.3)
- Initial state marker (Section 2.4)
- History state (Section 2.5)
- Final state (Section 2.6)
- Branching state (Section 2.7)

Whenever a sub-element is active, the composite state containing that sub-element will be active as well. If a transition is made from outside of a composite state directly to a sub-element, the composite state will be entered prior to its sub-elements. In a similar manner, if a transition is followed from a sub-element out of the composite state, the composite state will be exited after the sub-elements are exited.

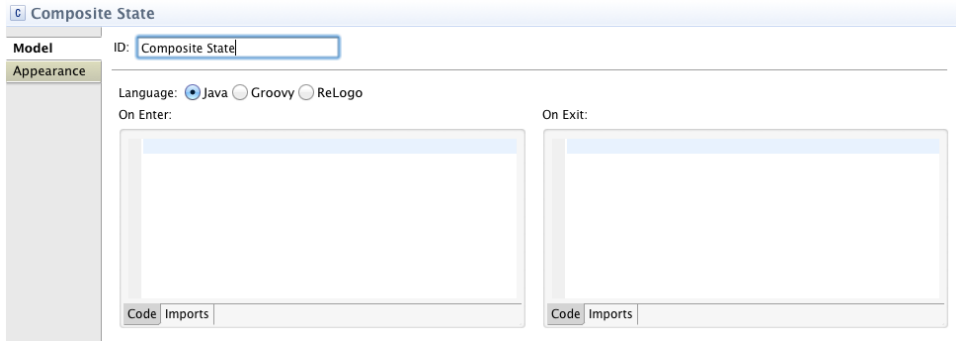


FIGURE 12. Composite state properties.

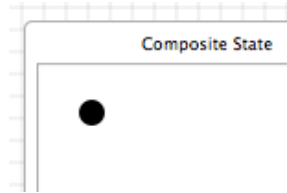


FIGURE 13. Initial state marker (within a composite state).

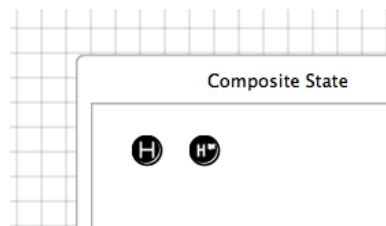


FIGURE 14. Shallow (left) and deep (right) history states (within a composite state).

2.4. Initial State Marker. ●

Any composite state that has a transition ending at it or contains history states must define an initial state marker. The initial state marker points to the element within the composite state that should be entered upon entering the composite state.

2.5. History State. H H*

There are two types of history states, shallow and deep (Figure 14). When a shallow history state is entered, the last active element within the enclosing composite state at the same hierarchical level of the history state is re-entered. For a deep history state, the last active *simple state* within the enclosing composite state, no matter at what level of the nesting hierarchy, is entered. In both cases if there was no previously active state, the state pointed to by the initial state marker is entered. Figure 15 shows the properties panel for a (shallow) history state. Only an *On Enter* element can be defined since history states are never directly exited.

Figure 16 shows an example statechart with shallow and deep history states defined. If the transition going from State 2 to External State 1 is followed, the Base Composite State will be re-entered via the shallow history state. Since shallow histories only track the last active state at the same hierarchical level within the enclosing composite state (Base Composite State), the shallow history state will indicate that the Internal Composite State

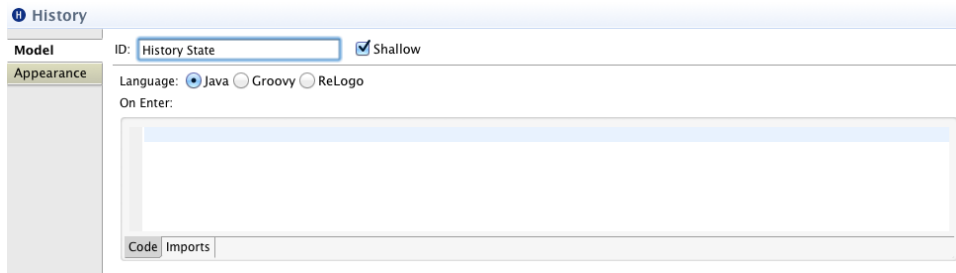


FIGURE 15. The properties panel for a shallow history state. A deep history state would have the same properties panel except that the *Shallow* element would be unchecked.

was the last active element and so State 1 will be entered. On the other hand, if the State 2 to External State 2 transition is followed and the Base Composite State is re-entered via the deep history state, State 2 will be entered, since this was the last active simple state within the Base Composite State.

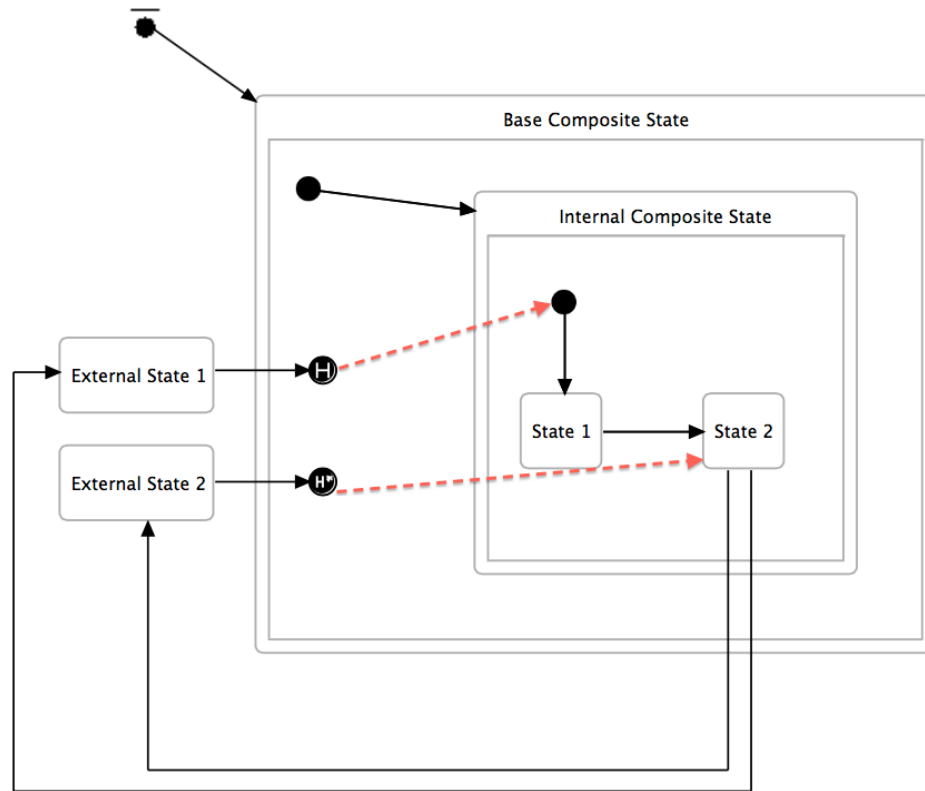


FIGURE 16. An example statechart demonstrating the differences between shallow (top) and deep (bottom) history states. The dashed red arrows indicate the paths taken from each of the history states.

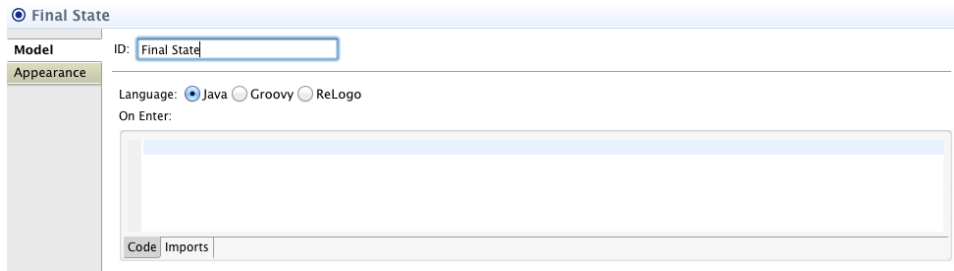



FIGURE 17. The properties panel for a final state.

2.6. **Final State.**  A final state marks the end of all activities for a statechart. When a final state is entered, no further states will be visited and no transitions will be triggered. Figure 17 shows the properties panel for a final state. Only an *On Enter* element can be defined since final states are never exited.

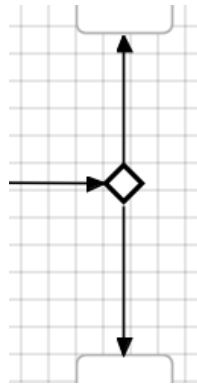


FIGURE 18. A branching state with one incoming and two outgoing transitions.

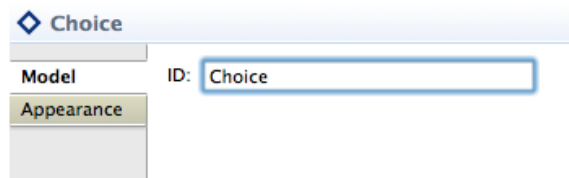


FIGURE 19. The properties panel for a branching state.

2.7. Branching State.

Branching states represent choices, or logical branching, within statecharts (Figure 18). Every branching state must define one outgoing *Default* transition, where the rest of the outgoing transitions are *Condition* transitions (Section 3.4). The *Condition* transitions are checked for validity and, if valid conditions are found, the transitions' priorities dictate the transition that is followed. If no valid transitions are found, the *Default* transition is followed. Figure 19 shows the properties panel for a branching state. Since a branching state is entered and immediately exited, nothing other than the state's *ID* can be specified.

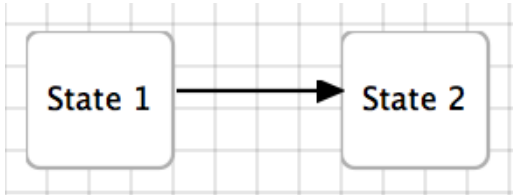


FIGURE 20. Regular transition between states 1 and 2.

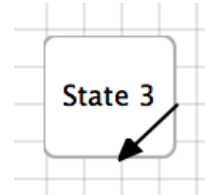


FIGURE 21. Self transition internal to State 3.

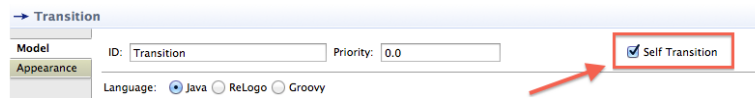


FIGURE 22. The transition properties panel Self Transition check box, marked with a red rectangle.

3. TRANSITIONS

Transitions between states make up the other fundamental building block of statecharts. In this section we introduce the different types of transitions that can be used within the Repast Symphony statecharts framework.

There are two overall types of transitions, *regular transitions* (Figure 20) which connect different states and *self transitions* (Figure 21) which are internal to a state¹⁰. There are a number of different transition trigger types, demonstrated in the transition properties panel in Figure 23 (these will be discussed below in further detail).

For any transition an *On Transition* action can be defined (see Figure 24). This action will be executed whenever the transition is traversed. The code editor for transition action blocks works the same way as the state's *On Enter* and *On Exit* action blocks. Autocomplete, syntax highlighting, and so forth are all supported. The keywords available within the *On Transition* action block are:

agent: This is the agent that contains the statechart.

transition: This is the transition itself. For example, the transition's source state can be accessed via: `transition.getSource()`¹¹.

¹⁰You can also define a *regular transition* that begins and ends at the same state. Unlike the *self transition* case, each time the *regular transition* is taken, the state will be exited and subsequently re-entered. You can toggle between a *self* and *regular* transition by toggling the Self Transition check box in the transitions properties panel (see Figure 22).

¹¹See the source or JavaDoc for `repast.simphony.statecharts.Transition` for all of the available methods.

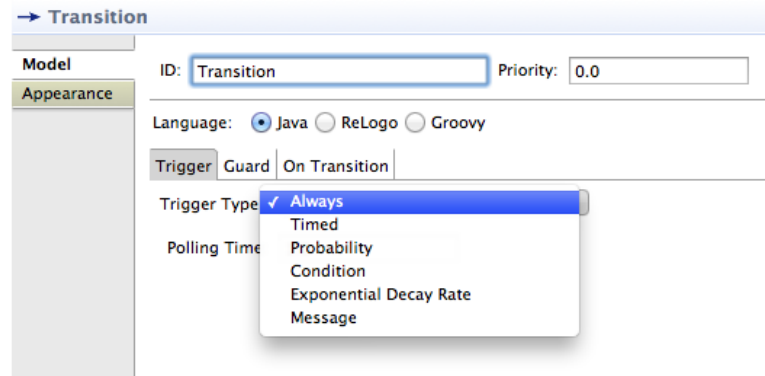


FIGURE 23. The properties panel showing the different types of transitions that are available.

params: This is the model's `Parameters` object.

For almost all types of transitions¹² a *Guard* condition can be defined (see Figure 25). A *Guard* condition is an additional boolean condition that has to be satisfied for a transition that is valid to be actually considered as a candidate for traversal. This condition is specified by a block of code that returns a boolean and the keywords available in a *Guard* condition are the same as those in an *On Transition* action block.

When there is more than one valid transition ties are broken using the priority of the transition. If the priorities of valid transitions are equal then one of the transitions will be chosen with a uniform random probability. The priority of a transition can be specified in the transition's properties panel.

Regular transitions can be divided into zero time transitions and non-zero time transitions¹³. For zero time transitions when a new state is entered, if there is a valid zero time transition out of it, that transition is followed immediately (with ties broken via priorities as usual). Always (Section 3.1), Condition (Section 3.4) and Message (Section 3.6) transition triggers are zero-time transitions.

Within a state, transitions are resolved starting with *self transitions* and proceeded by *regular transitions*.

Every transition has a *polling time* associated with it. This indicates the frequency (in units of simulation *ticks*) with which the transition is polled for validity. After the initial polling, the polling time can be modified within any action block with the call to `transition.setNextPollingTime(double)`. This way, a transition can be first polled at one polling time and at different polling time(s) subsequently¹⁴.

¹²All transitions except default transitions out of branching states.

¹³All *self transitions* are non-zero time transitions.

¹⁴This only impacts Always, Condition, Probability and Message triggers.

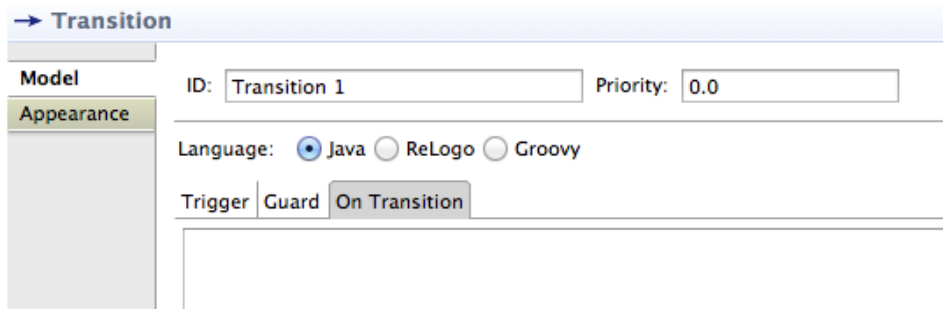


FIGURE 24. Properties panel for a transition showing the *On Transition* action block.

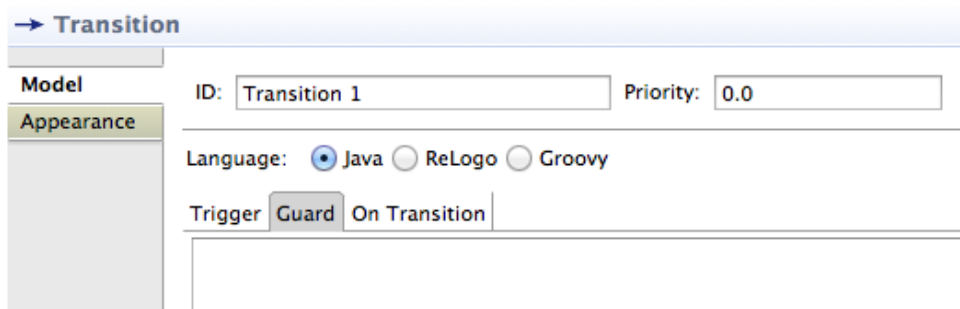


FIGURE 25. Properties panel for a transition showing the *Guard* condition block.

Next we present the different transition trigger types in more detail.

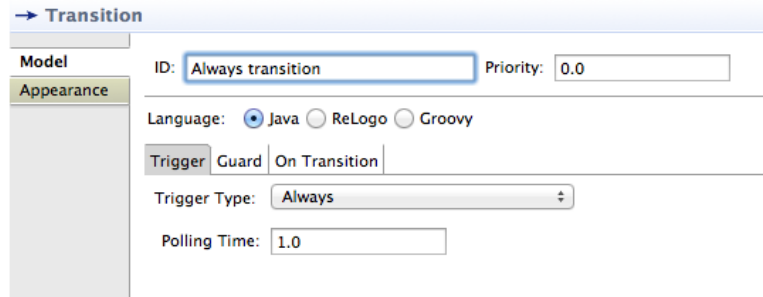


FIGURE 26. The properties panel for an *always* transition.

3.1. Always Trigger. *Always* triggers are always valid. The transition can, however, contain a *Guard* condition which if false, would prevent the transition from triggering. Because this trigger type results in zero time transitions, it is important to make sure that there are no *always* trigger transitions contributing to zero time loops in any statechart you create, since this has the potential to create never-ending loops. The properties panel for an *always* trigger transition is in Figure 26. One use for *always* trigger transitions is as self transitions to execute some action at a set polling time.

The screenshot shows the 'Transition' properties panel in the REPAST Statecharts IDE. The panel is titled '→ Transition' and has a sidebar on the left with 'Model' and 'Appearance' tabs. The 'Model' tab is active, showing the following fields:

- ID:** Timed Transition
- Priority:** 0.0
- Language:** Java (selected), Groovy, ReLogo
- Trigger:** Guard, On Transition
- Trigger Type:** Timed (dropdown menu)
- Time:** A large text area for entering code.
- Code Imports:** A small text area at the bottom of the Time field.

FIGURE 27. The properties panel for a *timed* transition.

3.2. Timed Trigger. *Timed* triggers become valid after some time, measured in simulation *ticks*. The properties panel for a *timed* trigger transition is shown in Figure 27. The *Time* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., 2 or `agent.getDelay()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). If at the time a *timed* trigger is valid a *Guard* condition keeps the transition from being valid, the transition does not get reinitialized and will simply remain invalid.

The screenshot shows a software interface for configuring a transition. The main title is "Transition". On the left, there are two tabs: "Model" and "Appearance".

- Model Tab:**
 - ID: Probability Transition
 - Priority: 0.0
- Appearance Tab:**
 - Language: Java (selected), Groovy, ReLogo
 - Trigger: Guard, On Transition
 - Trigger Type: Probability
 - Polling Time: 1.0
 - Probability: [Large text area]
 - Code Imports

FIGURE 28. The properties panel for a *probability* transition.

3.3. Probability Trigger. *Probability* triggers are evaluated as valid with a specified probability. The properties panel for a *probability* trigger transition is shown in Figure 28. The *Probability* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., 0.2 or `agent.getProbability()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated each time the transition is polled for validity.

The image shows a software interface for configuring a transition. At the top, there is a header bar with a right-pointing arrow and the text "Transition". Below this, there are two tabs: "Model" and "Appearance", with "Appearance" currently selected. The "Model" tab contains two input fields: "ID:" with the value "Condition Transition" and "Priority:" with the value "0.0". The "Appearance" tab contains several settings: "Language:" with radio buttons for "Java" (selected), "Groovy", and "ReLogo"; "Trigger:" with radio buttons for "Guard" and "On Transition"; "Trigger Type:" with a dropdown menu set to "Condition"; "Polling Time:" with an input field set to "1.0"; and a large text area labeled "Condition:" which is currently empty. At the bottom of the "Appearance" tab, there are two sub-tabs: "Code" and "Imports", with "Code" selected.

FIGURE 29. The properties panel for a *condition* transition.

3.4. Condition Trigger. *Condition* triggers are evaluated as valid based on a specified condition. The properties panel for a *condition* trigger transition is shown in Figure 29. The *Condition* element in the properties panel accepts general Java, Groovy or ReLogo code returning a boolean value, including simple boolean entries (e.g., `true` or `agent.getCondition()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated each time the transition is polled for validity.

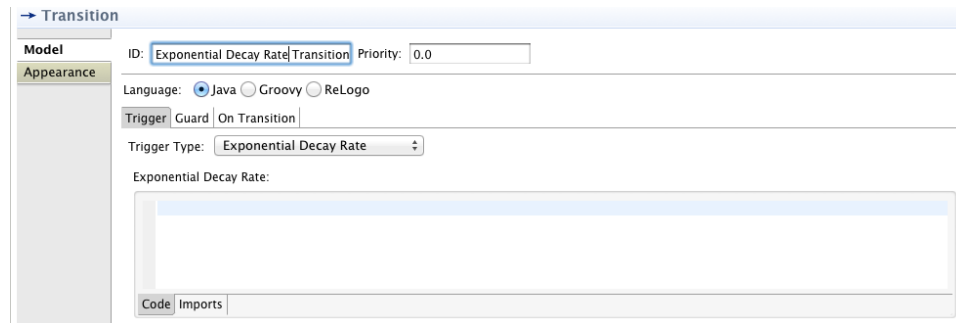


FIGURE 30. The properties panel for a *exponential decay rate* transition.

3.5. Exponential Decay Rate Trigger. *Exponential decay rate* triggers become valid after a random time following the exponential distribution. The properties panel for an *exponential decay rate* trigger transition is shown in Figure 30. The *Exponential Decay Rate* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., 2 or `agent.getDecayRate()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated when the transition is initialized (i.e., when a state is entered that has a possible *exponential decay rate* transition leading out of it). The code block supplies the λ parameter to the exponential distribution specified by the probability density function:

$$(1) \quad f(t) = \lambda e^{-\lambda t}$$

The expected value of an exponentially distributed random variable with parameter λ is $1/\lambda$. So given, for example, a λ of 2, the expected value for the time it would take for an *exponential decay rate* transition to trigger would be 0.5 in units of simulation *ticks*.

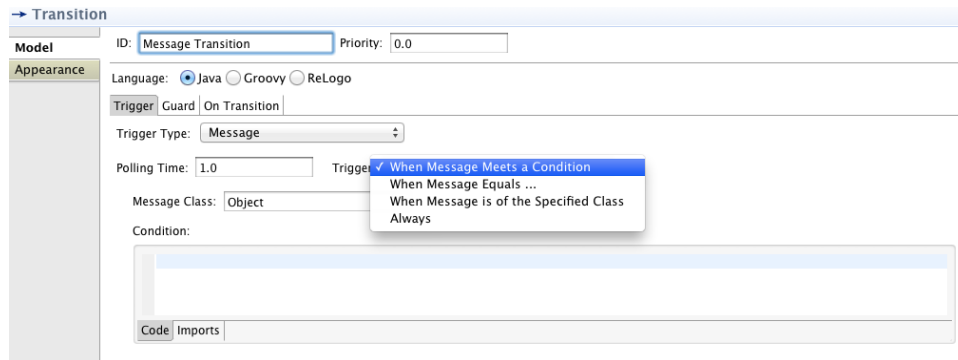


FIGURE 31. The properties panel for a *message* message, with the available trigger types shown under the *Trigger* element.

3.6. Message Trigger. *Message* triggers become valid when a message meeting specific criteria is received by the statechart. A statechart is sent a message when the statechart's `receiveMessage(Object)` method is called. From an agent-based modeling perspective, this would likely occur when an agent is sent a message and then the agent forwards the message to all or a subset of its statecharts¹⁵. Messages from the queue are consumed by message transitions in the order they were received. If there are message transitions to check, the queue will be checked until a valid message is found.

There are four different types of *message* triggers, shown in the drop down menu of the *Trigger* element in the *message* trigger properties pane in Figure 31, and we present them next.

¹⁵There is nothing to prevent one agent from directly accessing another agent's statechart if the statechart is visible, but it could be considered not very good practice in an object oriented programming sense.

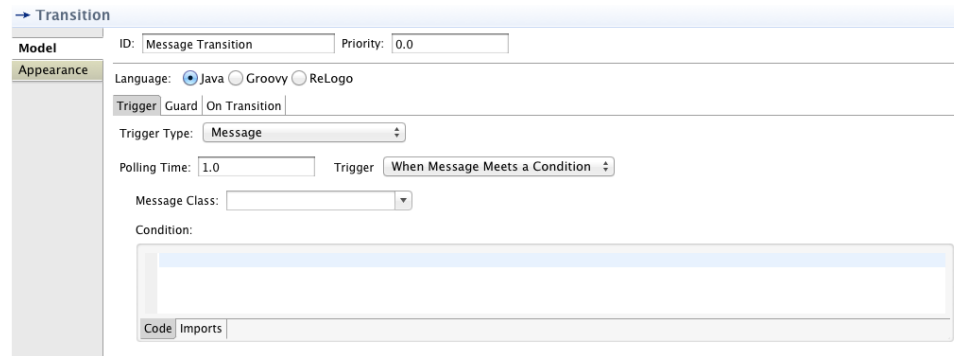


FIGURE 32. The properties panel for a *When Message Meets a Condition* message trigger.

3.6.1. *When Message Meets Condition*. The *When Message Meets a Condition* message trigger has the properties panel shown in Figure 32. The *Message Class* element specifies the type of the message, which can be any of the basic types in Figure 33 or the fully qualified name of any another type¹⁶. To specify a type not in the list, enter the fully qualified name of the type in the combo box. The *Condition* element in the properties panel accepts general Java, Groovy or ReLogo code returning a boolean value, including simple boolean entries (e.g., `true` or `agent.getCondition()`). The keywords available within the *Condition* action block are:

- agent:** This is the agent that contains the statechart.
- transition:** This is the transition itself.
- message:** This is the message received by the statechart.
- params:** This is the model's `Parameters` object.

The code block is evaluated each time the transition is polled for validity.

¹⁶The default *Message Class* is `java.lang.Object` which will accept any type of message.

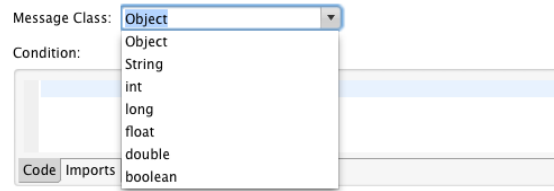


FIGURE 33. The basic types available for the *Message Class* element in the properties panel for the *When Message Meets a Condition*, *When Message Equals*, and *When Message is of the Specified Class* message triggers. Additional types can be specified with their fully qualified names.

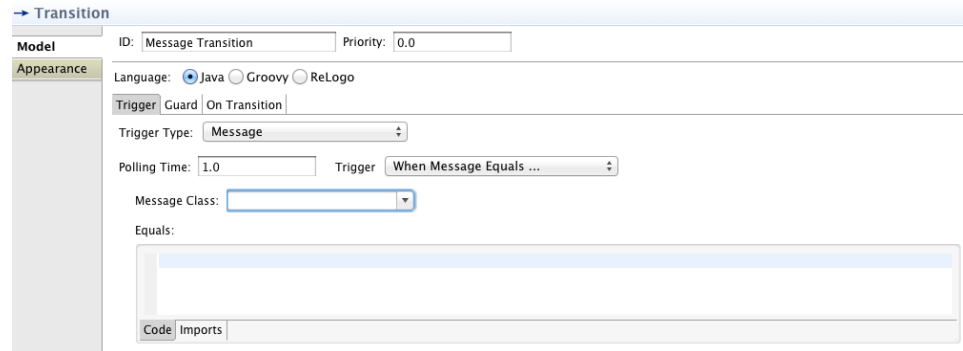


FIGURE 34. The properties panel for a *When Message Equals* message trigger.

3.6.2. *When Message Equals*. The *When Message Equals* message trigger has the properties panel shown in Figure 34. The only difference between this and the *When Message Meets a Condition* message trigger is that instead of a *Condition* element there is an *Equals* element that needs to be defined. The *Equals* element accepts general Java, Groovy or ReLogo code returning any value that will be checked against the received message using the message's `equals(Object)` method. The keywords within the *Equals* block are the same as the *On Transition* action block (i.e., `agent`, `transition`, `params`).

The *When Message Equals* trigger can be considered as a special case of the *When Message Meets a Condition* trigger where, if the contents of an *Equals* element in a *When Message Equals* trigger are denoted by `<Equals Contents>`, then the equivalent *Condition* element in a *When Message Meets a Condition* trigger would be:

```
message.equals(<Equals Contents>);
```

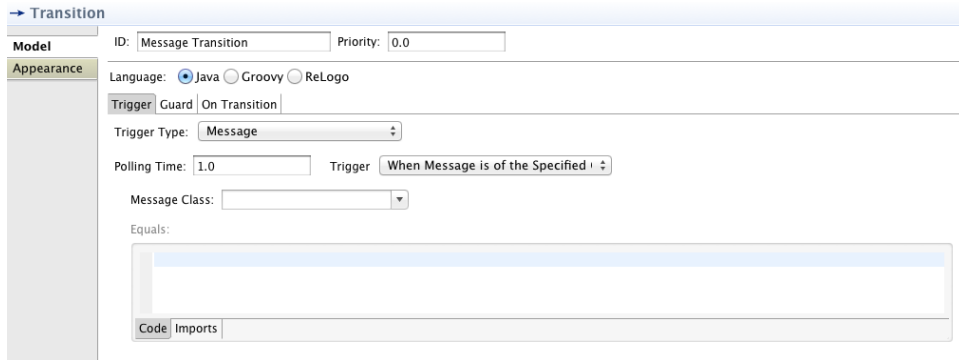


FIGURE 35. The properties panel for a *When Message is of Class* message trigger.

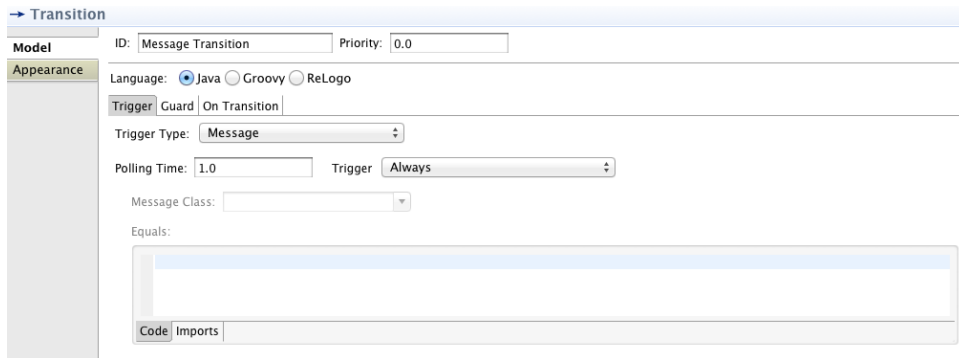


FIGURE 36. The properties panel for an *Always* message trigger.

3.6.3. *When Message is of Class*. The *When Message is of Class* message trigger has the properties panel shown in Figure 35. Any message that is received that is of the specified class type will result in the transition being triggered.

3.6.4. *Always*. The *Always* message trigger has the properties panel shown in Figure 36. Any message that is received will result in the transition being triggered.

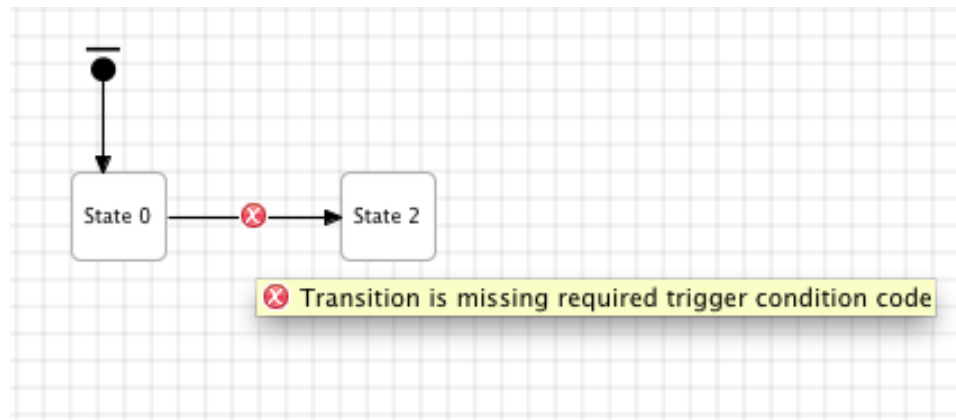


FIGURE 37. An error marker on a transition

4. DEBUGGING STATECHARTS

When a statechart is saved the code for that statechart will be generated in the `src-gen` directory in the statechart's project. At that time the statechart will also be validated and any structural warnings or errors will be displayed in the statechart workspace. Figure 37 is an example of this. The transition between State 0 and State 2 has an error. Moving the mouse pointer over the error marker will display a tooltip with the error message. In this case, the transition has a *Condition* trigger type but no condition has been specified in the transition's properties. The error message will also be displayed in Eclipse's *Problems* view.

If there is an error in the code generated by the statechart, you will see the error marker on the state chart element that contains the error (the state or transition) and on the `src-gen` folder in the statechart's project. This kind of error will occur when any code specified in the statechart's state or transition properties (such as the *On Exit*, *Condition* and so forth code blocks) is erroneous. To fix these kinds of errors, click on the element with the error and examine its action block properties. The error should be flagged in the action block editor as in Figure 38. The error itself is that the `getHealth` method is not defined on the agent. This should be fixed directly in the code block.

In the rare case, the error is not flagged in the action block editor itself, then expand the `src-gen` folder to find the offending file as in Figure 39. Open the file. The comments in the file describe the statechart element that produced the bad code and eclipse will flag errors in the code itself. Figure 40 shows such an error. As before, the error itself is that the `getHealth` method is not defined on the agent. The comments in the code state that this is the code for the "Condition trigger condition for Transition 3, from = State 0, to = State 2." To fix this then, we need to edit the trigger condition code in Transition 3. To find

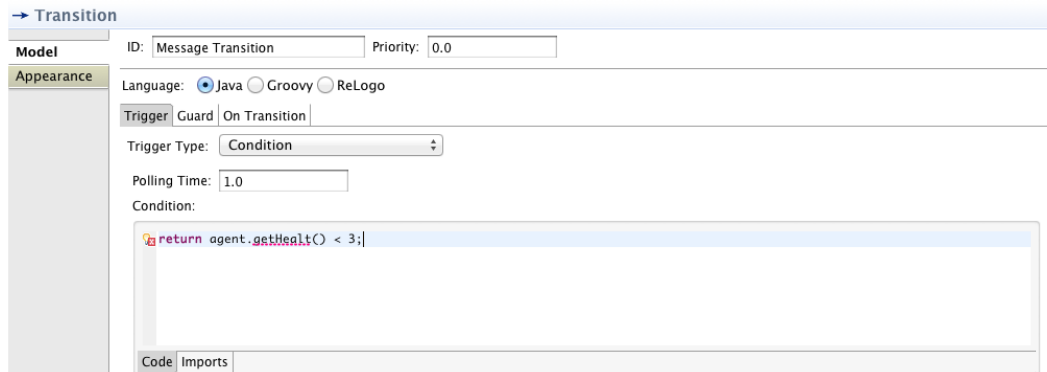
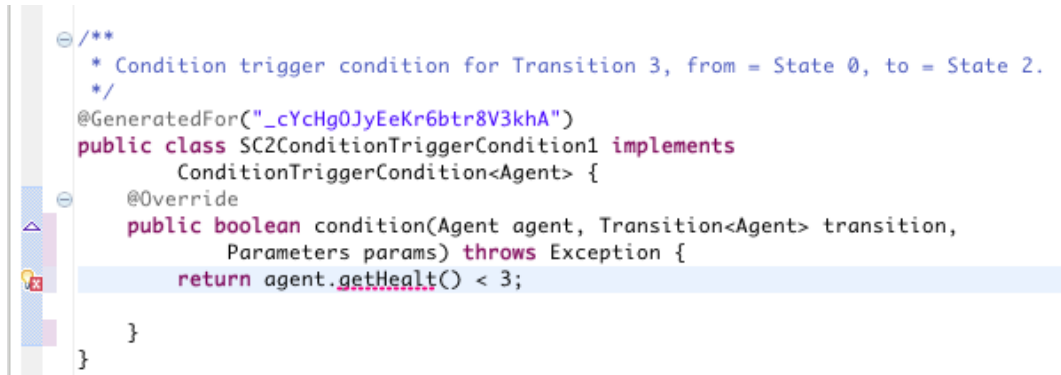


FIGURE 38. An error in the transition condition block flagged by the editor.



FIGURE 39. An example of an error in the code generated from the statechart.

it, we can use the *to* and *from* states mentioned in the comments and select the transition that connects them. Note that editing the code directly in the .java file will remove the error, but will **NOT** fix the problem. The next time the statechart is saved, the code will be regenerated. The code must be fixed in the statechart element that produced the problem.

A screenshot of a code editor showing a Java class definition. The code is as follows:

```
/**  
 * Condition trigger condition for Transition 3, from = State 0, to = State 2.  
 */  
@GeneratedFor("_cYcHg0JyEeKr6btr8V3khA")  
public class SC2ConditionTriggerCondition1 implements  
    ConditionTriggerCondition<Agent> {  
    @Override  
    public boolean condition(Agent agent, Transition<Agent> transition,  
        Parameters params) throws Exception {  
        return agent.getHealth() < 3;  
    }  
}
```

The line `return agent.getHealth() < 3;` is highlighted in light blue. The IDE interface includes a vertical toolbar on the left with icons for collapse, expand, and search, and a vertical scrollbar on the right.

FIGURE 40. An error in the code produced by the statechart. Note that the comments refer to the element that produced the code.

5. STATECHARTS AT RUNTIME

When a simulation is running, an agent's statechart can be visually accessed via the agent's probe panel. An agent's probe panel is launched by double clicking an agent in a display within the Repast Symphony runtime GUI. Figure 41 shows the Repast Symphony runtime GUI with a gridded display on the right and a probe panel on the bottom left (larger red rectangle). The probe panel was launched by double-clicking the visible agent on the right (blue filled circle). Within the probe panel, the button to display the *Statechart* statechart is indicated with the smaller red rectangle (and the red arrow). When this button is pressed, a statechart display is launched (Figure 42). The statechart display highlights active states with a green color¹⁷. Under the statechart display *Options* menu item there is a menu item (*Always On Top*) to always keep the statecharts display on top of other displays. This is enabled by default but can be disabled.

In addition, statecharts can be directly manipulated via mouse clicks, thereby permitting manual activation of inactive states and the forcing of transitions to be followed. This feature allows for experimentation with alternative or rare event paths through agent statecharts. Right click on a state to pop-up the menu to activate it, or right click on a transition to pop-up the menu to follow it. Note that a transition must be active if it is to be followed.

¹⁷As seen in Figure 42, a composite state is active if a sub-element is active (unless the sub-element is a final state).

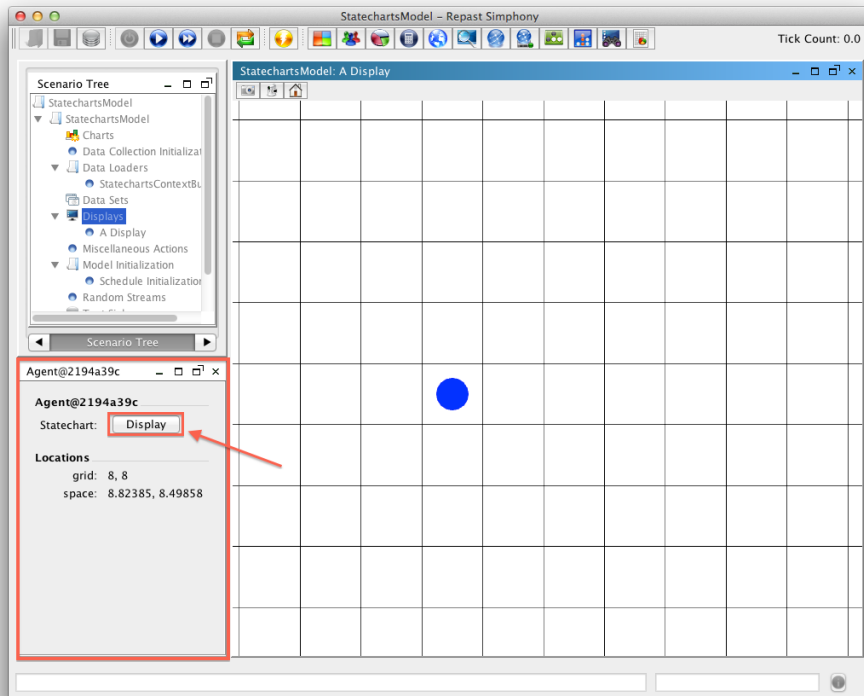


FIGURE 41. Repast Symphony runtime GUI with a probe panel showing (large red rectangle) and with a statechart display button (smaller red rectangle with arrow pointing to it).

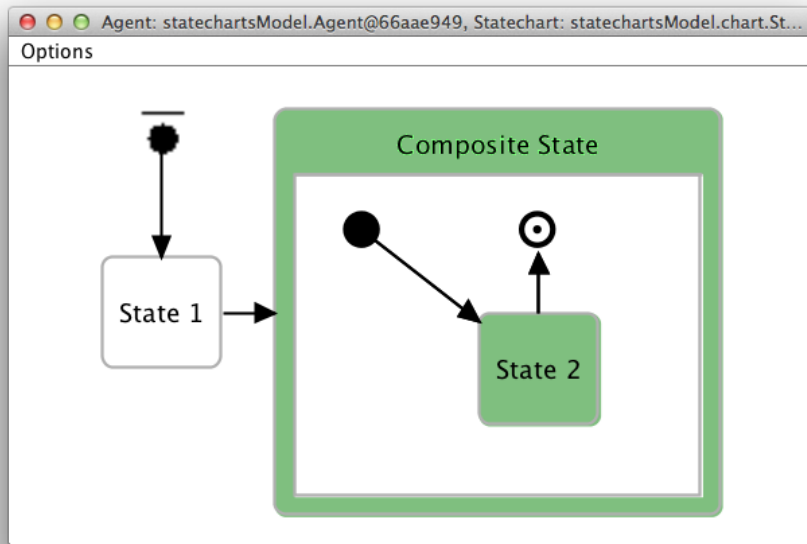


FIGURE 42. The runtime statechart display with State 2 and the Composite State that contains it highlighted as active.

6. PUTTING IT ALL TOGETHER

In this section we provide a few simple use cases for using statecharts in an agent model.

6.1. Scheduling Agent Behavior. Statecharts can make scheduling agent behaviors very easy to do.

Use Case: An agent repeats certain activities for a while and then execute some other action based on a condition, after which, the original activities are resumed.

Figure 43 shows a simple state configuration that can be used to achieve this. The *Go* state has a self transition with an *Always* trigger and a polling time of 1 (see Figure 44) and with the *On Transition* action block as shown in Figure 45. Thus the agent will execute its `doSomething()` method at each simulation *tick* until told otherwise. We can define the transition going out of the *Go* state to the branching state as a *Timed* transition, which will allow the agent to repeat its behavior until a certain amount of time has passed, at which point the statechart will proceed to *Case 1* or *Case 2* based on the condition checked by the branching state's outgoing transitions. This process can be repeated if the transitions leading back to the *Go* state are followed.

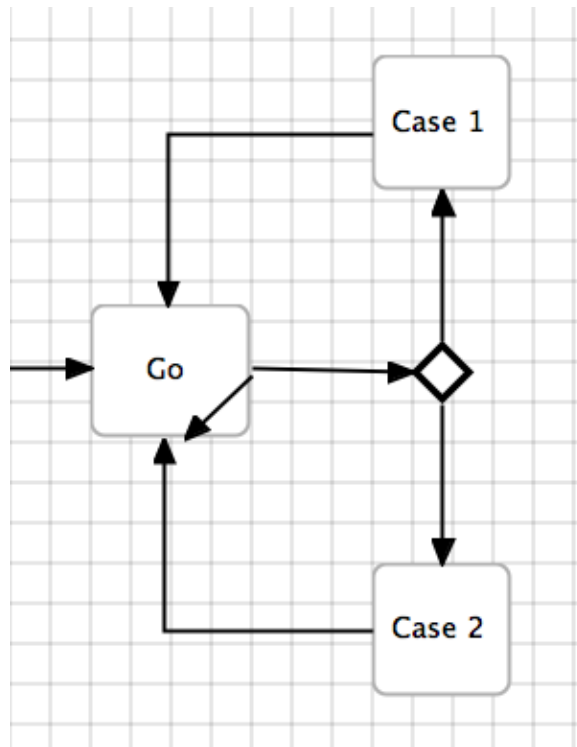


FIGURE 43. State configuration for simple behavior scheduling.

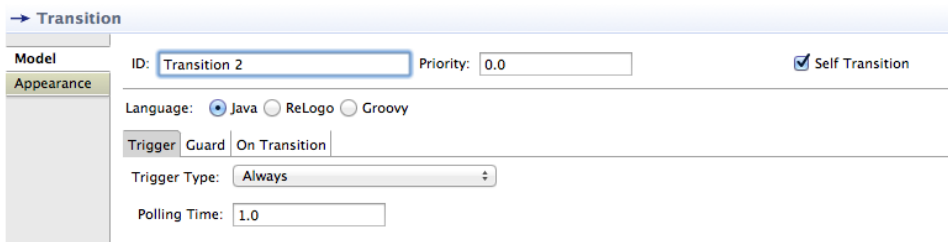


FIGURE 44. The trigger tab of the properties panel for the Go state’s self transition (see Figure 43).

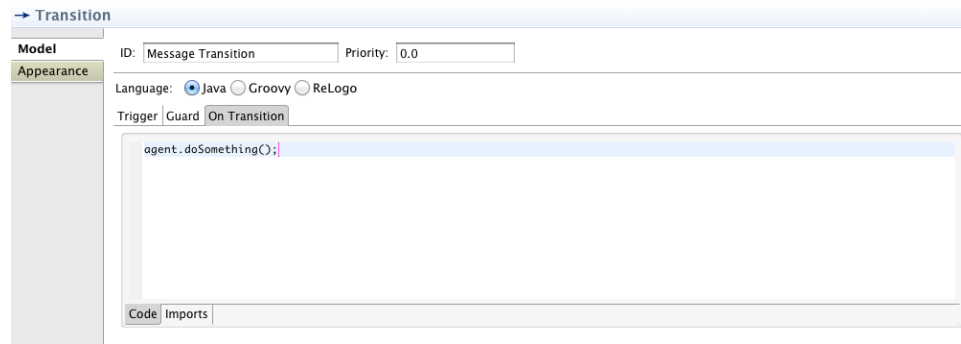


FIGURE 45. The *On Transition* action block for the Go state's self transition (see Figure 43).

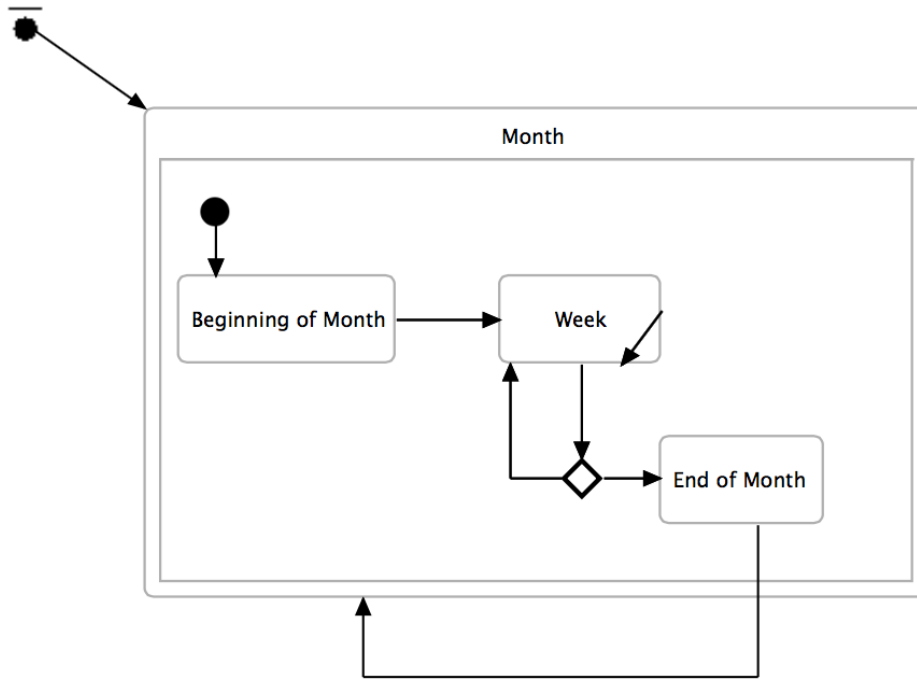


FIGURE 46. A statechart showing an example of how to schedule weekly and monthly activities.

Use Case: Agent behaviors need to be scheduled in a nested temporal structure.

This corresponds to, for example, weekly vs monthly activities. Figure 46 shows a simple example of how you might define a statechart that does this. The general idea here is that at the beginning of each month there are certain activities that need to be taken care of, after which the weekly cycle begins. This can continue until the end of month is reached, at which point any end of month activities can be executed. In this example, the month composite state is then re-entered, resulting in the start of another month.

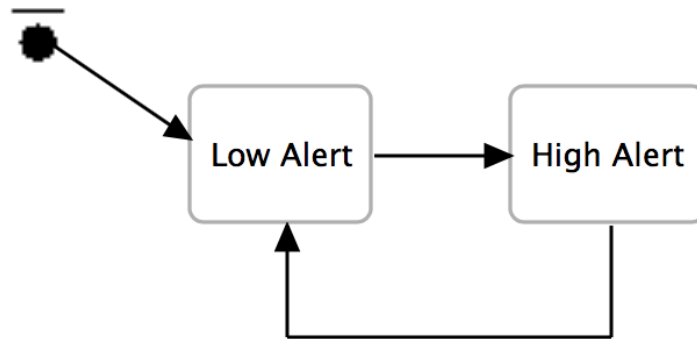


FIGURE 47. A statechart showing an example with “Low Alert” and “High Alert” states. The example uses message based transitions to loosely couple agent behaviors and the triggers for those behaviors.

6.2. Message Based Agent Behaviors. Message based triggers can make distributed, context dependent behavior easy to implement.

Use Case: Agents need to react to changing conditions within a simulation.

Figure 47 shows a simple example where agents change from a “Low Alert” to a “High Alert” state and vice versa. Each of the transitions are *When Message Equals* transitions (see Figure 48). The messages can be broadcast to agents within a certain region of a simulation or based on other criteria, but the important aspect of this example is the dynamic and loose coupling that can be achieved by using message based transitions.

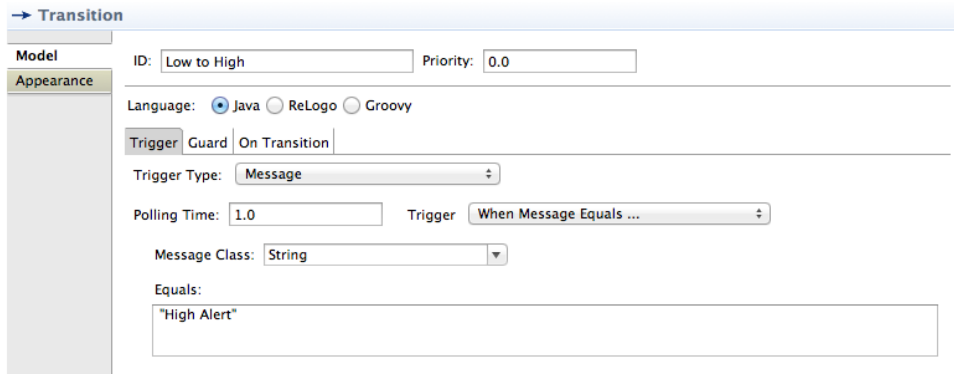


FIGURE 48. The properties panel for the transition from the “Low Alert” state to the “High Alert” state in the statechart from Figure 47. The transition has a *When Message Equals* trigger that looks for messages of type `String` with contents “High Alert”.