

# The Polyglot Tool Chain

---

This document provides information on how to use the Polyglot tool chain and summarizes its design. Readers should be familiar with Java and C++ programming, the modeling concepts of Statecharts and its variants, and the use of JPF: the Java Pathfinder software model checker.

## Contents

Introduction to the Polyglot tool chain .....	3
Objectives .....	3
Tool chain Architecture.....	3
Content of the release .....	5
Using the tool chain.....	5
GUI Driver in MATLAB.....	5
Installation.....	5
Functionality.....	5
MDL2MGA translator.....	7
StateMachineCG code generator .....	8
Running the generated code from the command line .....	9
Running the generated code with a CSV file.....	10
Format of the CSV file .....	11
Using Polyglot with JPF/SPF .....	12
Examples .....	12
Technical Details of the Tool chain.....	19
Specifying Properties in Simulink/Stateflow .....	19
State Machine Models in XML format .....	19
Model in Java format .....	22
The Polyglot Engine .....	24
Semantic modules .....	25

## Introduction to the Polyglot tool chain

### Objectives

We created the Polyglot tool chain to:

- (1) provide a precise, executable definition of semantics for various Statechart variants,
- (2) allow the execution of Statechart-like models using the different semantic variants,
- (3) support the analysis (model checking), symbolic execution, and test vector generation for models expressed in some Statechart-like language.

Statecharts were introduced by Harel, and several semantic variants have been developed. In its current form the toolchain has support for three variants: (1) UML State Machines (based on the UML 2.0 standard), (2) Rhapsody State Machines (based on the IBM Rhapsody tool Version X.XXX), and (3) MATLAB Stateflow (Version R2010b). There are semantics engines in the tool chain for each of the above variants, and there are model translators available for (2) and (3) (i.e. for importing models from Rhapsody and MATLAB).

### Tool chain Architecture

The high level architecture of the tool chain is shown in Figure 1 below. Reading from left to right:

1. Models are created in MATLAB (Stateflow) or in Rhapsody (UML State Machines).
2. These models are translated into an intermediate XML format by two, respective translators: MDL2MGA and Rhapsody2MGA. This XML file contains relevant information about the original models that is sufficient to simulate the models.
3. The XML file is translated by a code generator: StateMachineCG that produces a Java representation of the models.
4. The model (in Java) is then executed by the Polyglot engine that uses one of the three possible 'semantic plugins': sf\_sem (for Stateflow semantics), rh\_sem (for Rhapsody semantics), and uml\_sem (UML semantics).
5. The model can be executed by the engine in different ways (not indicated on the diagram):
  - a. It can read a text file of input events and it produces another text file containing the output events produce by the state machine
  - b. It can run under the control of JPF with non-deterministic choices, backtracking, and the JVM state monitored (as usual for programs executed under JPF)
  - c. It can be run under SPF (the Symbolic Pathfinder of the JPF family of tools) for symbolic exploration of the state space and to generate test vectors.

- On the MATLAB side, there is also a graphical user interface to add specifications to the ‘main’ model to be analyzed that will be translated into a portions of the generated model that in turn will be checked when the model is executed.

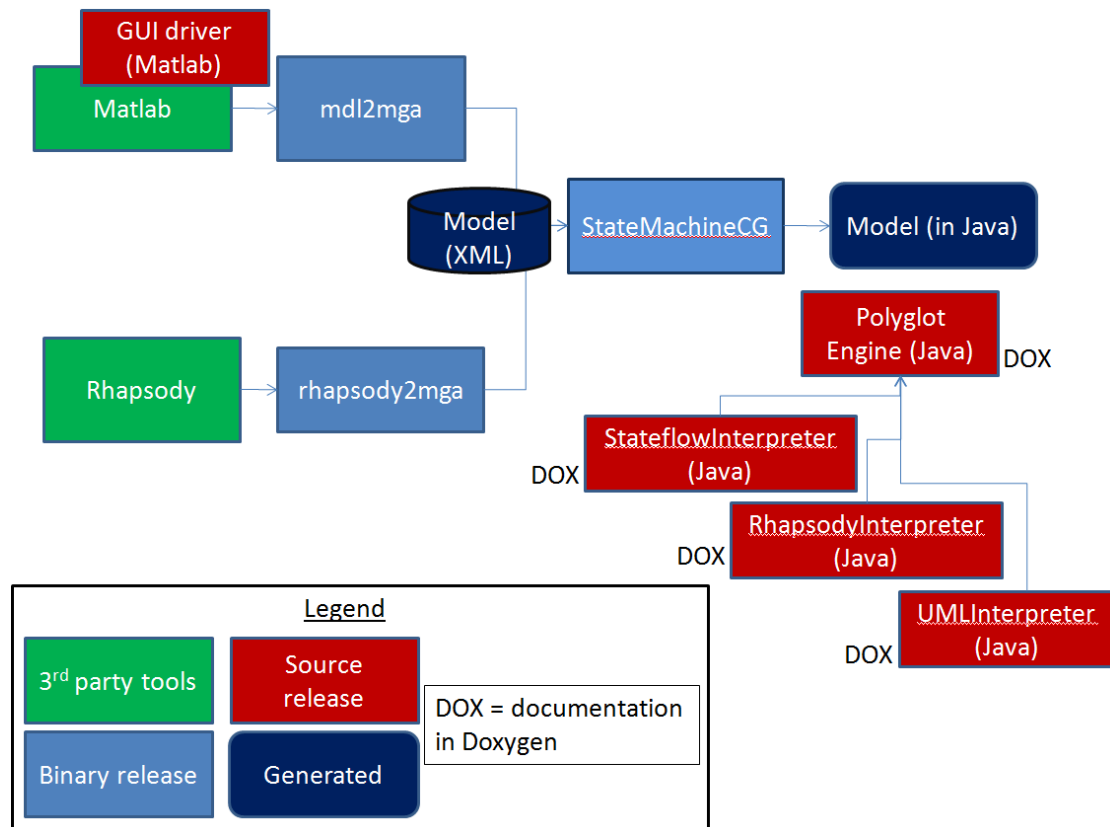


Figure 1 - The Polyglot toolchain.

Note that the core part of the tool chain is the Polyglot Engine (PE) that acts as an interpreter. Its ‘program’ is the state machine model, its input is a sequence of input values to the state machine, and its output is a sequence of output values produced by the state machines. (I.e. input and output traces). The state machine model is implemented as a set of Java objects at run-time – these objects are created by the Java code produced by the StateMachineCG code generator.

The PE can run in standalone mode: one can simply compile and link the ‘model’ Java file with the engine libraries, and then run the resulting executable – against an input file containing input sequences. The PE can also run under the control of JPF, where JPF can monitor the Java code execution. If the PE runs under the control of SPF, specific input values are considered as ‘symbolic’ (e.g. constrained integers), and SPF can be used for generating test vectors that will exercise state machine.

The specifications that can be added to the Stateflow model follow a pattern language (developed by Dwyer, et. al in 1999). These specifications are carried forward through the tool chain, and StateMachineCG will produce Java code from them that will be executed the PE when the state machine

is running. This is for checking temporal logic properties over variable values and states in the state machine.

## Content of the release

- The code generators: MDL2MGA, StateMachineCG, and Rhapsody2MGA are released in binary form. These are rather complicated to build, and they require other libraries, so they are made available in the form of executables (packaged with a few DLL-s) for immediate use.
- The XML schema (and UML metamodel) for the state machine models in XML is included.
- The Polyglot Engine with the three pluggable semantics is in Java, and is included in source code.
- The same Java modules and base classes for the state machine models in Java are documented using the Doxygen tool. The generated documentation is also included.
- The release includes a number of examples: models, data files, and configuration files.

## Using the tool chain

### GUI Driver in MATLAB

The ‘MATLAB GUI Driver’ extends the graphical user interface of the MATLAB/Simulink environment such that specifications can be added to the models. The specifications will be translated by the tool chain, and verified when the model is executed.

### Installation

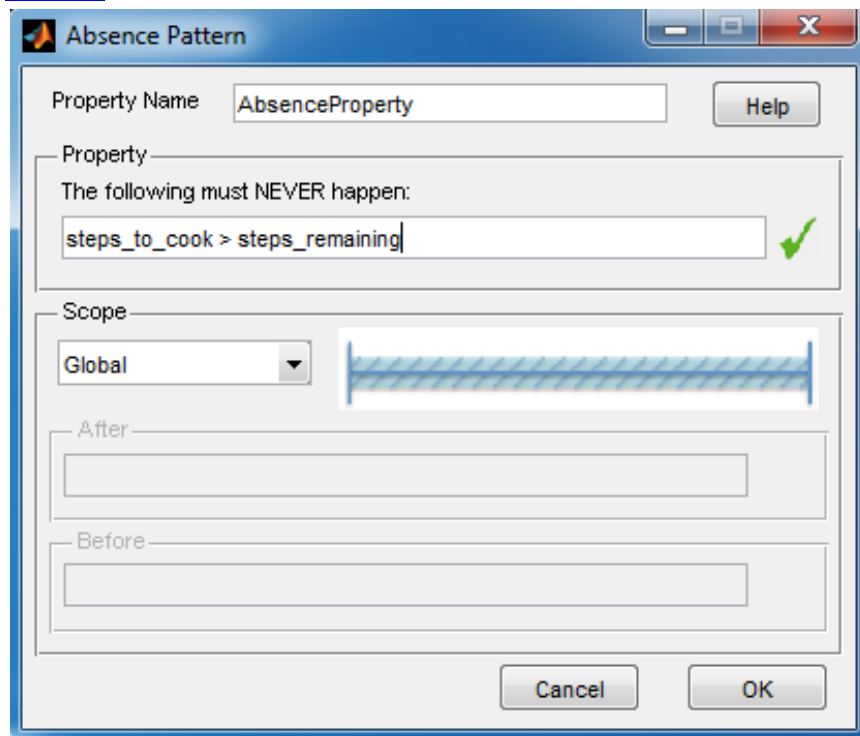
To enable the MATLAB custom extension the following directories in this package have to be added to the MATLAB path:

1. `$(INSTALL_DIR)/src/matlab`
2. `$(INSTALL_DIR)/src/matlab/images`

### Functionality

1. Defining Temporal Properties in Stateflow
  - a. This feature is accessible via the Context Menu > MTV Properties > Add Property. The supported properties include Absence, Existence, Universality, Precedence and Response. These properties can be observed in the scopes Global, Before, After, Between and Until.
  - b. The dialog that is presented after the user picks a pattern can be divided into two sections. The first section addresses the pattern itself; here we can specify a Boolean expression over variables that are defined for the given Stateflow. The green checkmark in the example below indicates that the variable names used are valid. The second section of the dialog addresses the scope of the property; here the expressions are defined in the same manner. To learn more about property patterns, use the “Help”

button, it in turn will open a browser window and navigate it to [KSU's Spec Pattern Website](#).

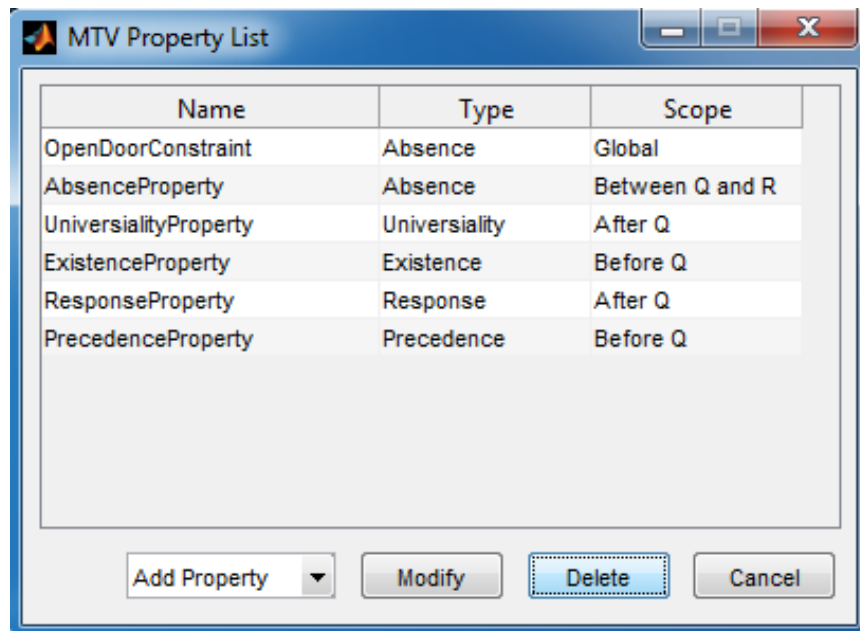


The 'Absence Pattern' dialog box is shown. It has a title bar with a standard icon and window controls. The main area contains the following fields and controls:

- Property Name:** A text box containing 'AbsenceProperty' and a 'Help' button to its right.
- Property:** A section with the text 'The following must NEVER happen:' followed by a text box containing 'steps\_to\_cook > steps\_remaining' and a green checkmark icon to its right.
- Scope:** A dropdown menu set to 'Global' and a blue hatched rectangular area to its right.
- After:** An empty text box.
- Before:** An empty text box.
- Buttons:** 'Cancel' and 'OK' buttons at the bottom right.

## 2. Listing and Modifying Properties in Stateflow

- a. This feature is accessible via the Context Menu > MTV Properties > List Properties. The dialog invoked this way provides access to properties already defined and enables their modification or removal.

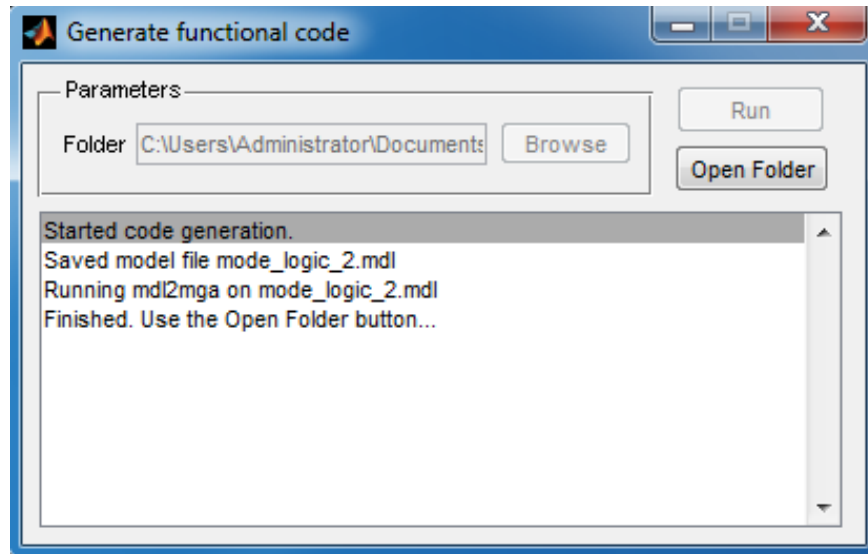


The 'MTV Property List' dialog box is shown. It features a table with three columns: Name, Type, and Scope. Below the table are four buttons: 'Add Property' (with a dropdown arrow), 'Modify', 'Delete' (highlighted with a blue border), and 'Cancel'.

Name	Type	Scope
OpenDoorConstraint	Absence	Global
AbsenceProperty	Absence	Between Q and R
UniversalityProperty	Universality	After Q
ExistenceProperty	Existence	Before Q
ResponseProperty	Response	After Q
PrecedenceProperty	Precedence	Before Q

## 3. Generating Java code

- a. This feature is accessible via the Tools Menu > Model Transformation and verification... > Generate Java Code. This dialog drives the translation process that takes the defined Stateflow model with all the specified properties and produces functional Java code in a given folder. The produced Java code can then be compiled and executed as outlined in the later sections of this documentation.



## MDL2MGA translator

The MDL2MGA translator takes a MATLAB Simulink/Stateflow model in an .mdl file and translates it into an intermediate XML format that can be processed by StateMachineCG. It is important that you have MATLAB installed on the machine on which MDL2MGA is executed, or MDL2MGA will not be able to perform the translation; MDL2MGA actually loads the input model into MATLAB in order to translate it. To invoke the MDL2MGA translator, use the following command format:

```
MDL2MGA -n [options] input_file_name.mdl [output_file_name.xml]
```

Arguments are as follows:

- input\_file\_name.mdl* : (Required) A MATLAB Simulink/Stateflow .mdl (model) file to be used as the input file for the translation.
- output\_file\_name.xml* : (Optional) The name of the output UML file – MUST end in “.xml”. If this argument is not specified, the output UML file has the same name and location as the input file, but with an .xml extension.

Options relevant to this tool-chain are as follows:

- n (ALWAYS USE) Create a new UML .xml output file, as opposed to adding the UML translation of the .mdl file to an already-existing model. This option is explicitly shown in the command format above because, for this tool-chain, you will always want to use it.
- L *directory* Add *directory* to the MATLAB search-path. This is necessary if the input .mdl file

uses libraries or m-files located in *directory*.

**-m *mfilename*** Will cause the m-file specified by *mfilename* to be executed in MATLAB before the input model is loaded into MATLAB for translation. This is necessary if one or more m-files are needed to initialize the workspace for the input .mdl file.

## StateMachineCG code generator

The StateMachineCG translator takes as input an .xml file produced by the MDL2MGA translator and produces as output a number of .java files. These generated .java files describe the structure of a Statechart using the concepts of the Polyglot framework. Also generated is a .java file that implements the logic to read values for the model's input variables. Optionally generated are a number of .java files that implement the logic of any temporal properties defined in the MATLAB model and translated into the intermediate .xml format by the MDL2MGA translator. To invoke the StateMachineCG translator, use the following command format:

```
StateMachineCG.exe input_file_name.xml
```

After StateMachineCG has been run, the generated java code can be run with the following options:

```
java classname [-s <SF | UML | RHAPSODY>] [-d <CSV -c csvfile | CommandLine | Symbolic>] [--checkOutput]
```

The options are:

**-s [--semantics] <SF | UML | RHAPSODY>** Selects the semantic variant for Statechart model execution. Default is Stateflow. Case insensitive.

**-d <CSV -c *csvfile* | CommandLine | Symbolic>** Sets the data provider. If CSV is selected, the -c option should be used to tell the name of the .csv file. Symbolic is for use with JPF/SPF. Default is CommandLine. Case insensitive.

**--checkOutput** Used only in combination with the "-d CSV" option. Says that the outputs of the model should be compared against expected values that are in given in the .csv file passed to the -d option. Case sensitive.

For example, the command:

```
java TopLevelAres -s Rhapsody -d CSV -c csvinputfile.csv --checkOutput
```

executes TopLevelAres:

- With Rhapsody semantics
- using a CSV reader with values coming from the file *csvinputfile.csv*
- checking the output values of the chart against the expected output contained in the .csv file

Another example:



java TopLevelOrion

executes with default arguments:

- Stateflow semantics
- Command line reader (you have to input the current event and input values at each step on the command line)
- Won't check the output (because the checkOutput switch (case-sensitive) wasn't present)

## Running the generated code from the command line

The generated code can be run directly from the command line, in which case input events and data values are given manually by the user. We will show how to do this using an included sample, which is located at \$(INSTALL\_DIR)\samples\commandlineex.

Step 1: Generate the ESMoL model from the Stateflow model.

Open a command prompt and navigate to the \$(INSTALL\_DIR)\samples\commandlineex directory. Run the following command:

```
> MDL2MGA.exe -n commandlineex.mdl
```

Step 2: Run StateMachineCG

Again at the command prompt, run the StateMachineCG code generator on the newly created .xml file:

```
>StateMachineCG.exe commandlineex.xml
```

This will translate the structure of the Stateflow model from the intermediate format into a Java representation used by the Polyglot framework. Two files will be produced: commandlineexReader.java and TopLevelcommandlineex.java.

Step 3: Run the generated code from the command line

Compile the file TopLevelcommandlineex.java (make sure that both Statemachines.jar and JSAP-2.1.jar, both located in the \$(INSTALL\_DIR)\bin directory, are on the classpath). Then, run TopLevelcommandlineex.java in the following way:

```
>java -cp .;$(INSTALL_DIR)\bin\Statemachines.jar;$(INSTALL_DIR)\bin\ JSAP-2.1.jar  
TopLevelcommandlineex
```

After issuing this command, the prompt will wait for user input; the format for the input is:

```
name_of_event <return>  
value_of_input_variable_x <return>  
name_of_event <return>  
value_of_input_variable_x <return>  
...
```

If the empty event is needed, the user can simply press return. In general, after the input event is entered, the input variables are entered, one at a time, in the same order as specified by the model. If the Statechart modeling language used to create the model does not assign an explicit order, the order used by the generated code can be checked by looking at the generated *setData* method contained inside the generated .java file that begins with “TopLevel”; for instance, for this example, the *setData* method is contained in the file TopLevelcommandlineex.java.

For this example, entering the sequence

```
b
1
a
-1
```

Will drive the model from state A to state B and back to state A.

### Running the generated code with a CSV file

The generated code can be given input values from a CSV file, as shown below. We will show this using an included sample, located at \$(INSTALL\_DIR)\samples\csvexample.

Step 1: Generate the ESMoL model from the Stateflow model.

Open a command prompt and navigate to the \$(INSTALL\_DIR)\samples\csvexample directory. Run the following command:

```
> MDL2MGA.exe -n csvexample.mdl
```

Step 2: Run StateMachineCG

Again at the command prompt, run the StateMachineCG code generator on the newly created .xml file:

```
>StateMachineCG.exe csvexample.xml
```

This will translate the structure of the Stateflow model from the intermediate format into a Java representation used by the Polyglot framework. Two files will be produced: csvexampleReader.java and TopLevelcsvexample.java.

Step 3: Run the generated code from the command line

Compile the file TopLevelcsvexample.java (make sure that both Statemachines.jar and JSAP-2.1.jar, both located in the \$(INSTALL\_DIR)\bin directory, are on the classpath). Then, run TopLevelcsvexample.java in the following way:

```
> java -cp .;$(INSTALL_DIR)\bin\Statemachines.jar;$(INSTALL_DIR)\bin\ JSAP-2.1.jar TopLevelcsvexample
-d CSV -c csvexample.csv
```

This command will cause the model to be executed using the values provided in csvexample.csv. If you look inside this file, you will see the expected format of the .csv file: the first line should contain the labels of the input data variables to the chart, and the remaining lines should be comma separated values for the input variables to the chart, followed by comma separated values for the output values of the chart. The output values are used to check the output of a model with the --checkOutput option; for instance, executing the command:

```
> java -cp .;$(INSTALL_DIR)\bin\Statemachines.jar;$(INSTALL_DIR)\bin\JSAP-2.1.jar TopLevelcsvexample  
-d CSV -c csvexample.csv --checkOutput
```

Will cause the expected output values of the model provided in the .csv file to be compared to the output values produced during execution. The result is a generated .txt file named "result.txt" that contains a single line that says either "OK" if the expected values match the produced values, or "FAILED" if the expected values do not match the produced values.

### Format of the CSV file

The generated code can be used with CSV (comma separated values) files to quickly provide inputs and check the values of the outputs against expected values.

The format of the CSV file is the following:

Line 1: comma separated list of the names of the input events, followed by a comma separated list of the names of the input variables, followed by a comma separated list of the names of the output variables.

Subsequent lines: comma separated list of values of the input events, followed by comma separated list of values of the input variables, followed by a comma separated list of values of the output variables.

For example, for a Statechart model with one input event named "command", whose first input is an integer x and whose second input is a Boolean b, and whose first output is a float y and whose second output is an integer z, a .csv file for this model might look like the following:

```
command,x,b,y,z  
on,1,true,3.4,5  
off,2,false,2.6,7
```

If the same Statechart model has no input events, its .csv file would look like the following:

```
x,b,y,z  
1,true,3.4,5  
2,false,2.6,7
```

The exact order in which the input/output values should be given is determined by the Statechart model. If this cannot be explicitly set in the modeling tool and the Java code was automatically generated, the order can be seen by looking at the *setData* method in the generated Java code.

## Using Polyglot with JPF/SPF

Even though the Polyglot framework can be used by itself to simulate a single Statechart using different semantics, it can also be used with Java Pathfinder (JPF), a software model checker for Java, and its symbolic execution engine, Symbolic Pathfinder (SPF). SPF has the useful capability of finding interesting sequences of inputs for a Statechart that cause the Statechart to be driven through many different paths of execution. In other words, SPF can work with the Polyglot engine to create test-vectors (i.e., sequences of bindings to input variables) that drive a Statechart through different combinations of States.

The easiest way to use the Polyglot framework with JPF and SPF is with the Eclipse IDE and the plugins for JPF and SPF.

For information on installing JPF, please see: <http://babelfish.arc.nasa.gov/trac/JPF/wiki/install/start>.

For information on installing SPF, please see: <http://babelfish.arc.nasa.gov/trac/JPF/wiki/projects/JPF-symbc>.

This version of Polyglot has been tested with JPF source version 539 and SPF source version 326.

## Examples

We assume for this example that both Java Pathfinder (JPF), and Symbolic Pathfinder (SPF) are installed according to the directions above, along with the JPF plug-in for Eclipse.

Consider the following Stateflow model of Figure 2:

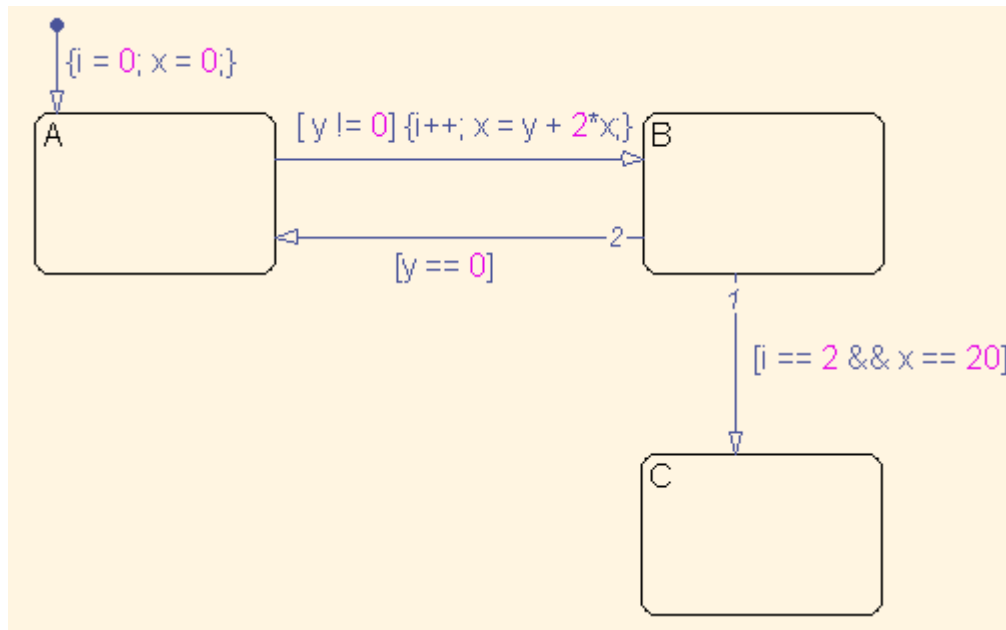


Figure 2 - Example Stateflow model

Assume that  $y$  is an input variable, and  $i$  and  $x$  are internal Stateflow variables. The question we wish to answer is: “Does there exist a sequence of input values for the variable  $y$  such that State C is reachable?” The answer is yes; for instance, with the input sequence  $\{y = 2, 0, 16\}$ . The following steps show how SPF can help us discover this automatically. This example Stateflow model is located at `$(INSTALL_DIR)\samples\symbolicex1\SymbolicEx1.mdl`.

Step 1: Run MDL2MGA

At a command prompt, run the MDL2MGA interpreter on the .mdl file:

```
> MDL2MGA.exe -n SymbolicEx1.mdl
```

This will translate the Stateflow model into an intermediate format, producing the output file `SymbolicEx1.xml`

Step 2: Run StateMachineCG

Again at the command prompt, run the StateMachineCG code generator on the newly created .xml file:

```
> StateMachineCG.exe SymbolicEx1.xml
```

This will translate the structure of the Stateflow model from the intermediate format into a Java representation used by the Polyglot framework. Two files will be produced: `TopLevelSymbExample.java` and `SymbExampleReader.java`.

### Step 3: Create a new Eclipse project with the generated files

Inside Eclipse, create a new project with the two newly generated Java files. You will need to add two external JARs to the dependencies, which are both included in the \$(INSTALL\_DIR)\bin directory: Statemachines.jar and JSAP-2.1.jar.

### Step 4: Create a .JPF configuration file

In order to run SPF with the generated code, a configuration file that contains settings for SPF must be created. Create a file named symbex1conf.JPF (the exact name isn't important as long as the extension is .JPF). The exact contents of your configuration file will depend on your particular directory structure; the following shows example settings:

```
-----  
#Example symbex1conf.JPF file  
target = TopLevelSymbExample  
  
#IMPORTANT: The classpath setting should include all JARs and  
#directories with classes needed to run the example  
#Replace $(INSTALL_DIR) with the path to where Polyglot resides  
  
classpath=.;$(INSTALL_DIR)/samples/symbolicex1;$(INSTALL_DIR)/bin/JSAP-  
-2.1.jar;$(INSTALL_DIR)/bin/Statemachines.jar;  
  
sourcepath = C:/work/NASARelease/samples/symbolicex1  
symbolic.method=TopLevelSymbExample$REGION_T$STATE_T.setData(sym)  
coverage.include=*. *  
coverage.show_methods=true  
coverage.show_bodies=true  
symbolic.minint=-10  
symbolic.maxint=20  
search.depth_limit=80  
  
# The SF option uses Stateflow semantics; can also use UML or Rhapsody  
target_args=-s,SF,-d,SYMBOLIC  
listener =  
gov.nasa.JPF.symbc.sequences.SymbolicSequenceListener,gov.nasa.JPF.lis-  
tener.CoverageAnalyzer  
coverage.show_methods=true
```

---

### Step 5: Explore with SPF

If everything has been entered correctly, you should be able to right click on the file symbex1conf.JPF from inside the Eclipse menu and select "Verify..." from a context menu. Upon doing this, a lot of output from SPF will scroll by on the console window. Some of this output should look like the following:

```
[setData(1), setData(0), setData(18), setData(5), setData(18),
setData(9), setData(18), setData(7), setData(10)]
[setData(1), setData(0), setData(18), setData(1), setData(-1),
setData(16), setData(-8), setData(7)]
[setData(1), setData(0), setData(18), setData(4), setData(18),
setData(15), setData(16)]
[setData(1), setData(0), setData(18), setData(16), setData(8),
setData(5)]
[setData(1), setData(0), setData(18), setData(-4), setData(-1)]
```

Each set of calls to the method `setData(int)` inside the square brackets is a test-vector generated by SPF. Each of the generated test-vectors above cause the Statechart to reach State C. For instance, the last test-vector above causes the following state/variable sequence in the model:

```
Initial condition:
current state = A, value of x = 0
After setData(1):
current state = B, value of x = 1
After setData(0):
current state = A, value of x = 1
After setData(18):
current state = B, value of x = 20
After setData(-4):
current state = C, value of x = 20
```

Thus, SPF has shown that it is possible to reach state C.

## Defining Temporal Properties

As mentioned earlier, Polyglot also has a mechanism for defining and monitoring temporal properties that one wishes to check on their Statechart models. This section provides details of how to: (1) Define these properties in the Stateflow environment, (2) Monitor them with Polyglot, (3) Use Symbolic Pathfinder to search for input sequences that violate the properties.

The example Statechart we will use to demonstrate properties is located at `$(INSTALL_DIR)\samples\cruisecontroller\cruise_control.mdl`. The model is shown below in Figure 3. This is a very simple model of a cruise controller. The variable “command” is an input variable that is given by the user. The model begins in the off state. When the user turns the cruise controller on (`command == 1`) the model transitions to the `CC_disengaged` state. From here, if the user sets the speed on the controller (`command == 3`) the model transitions to the `CC_engaged` state. Braking (`command == 4`) will then transition to the `CC_paused` state. While paused, either setting the speed (`command == 3`) or resuming (`command == 5`) will transition back to `CC_engaged`. Turning the cruise controller off (`command == 2`) will cause a transition to the `CC_off` state.

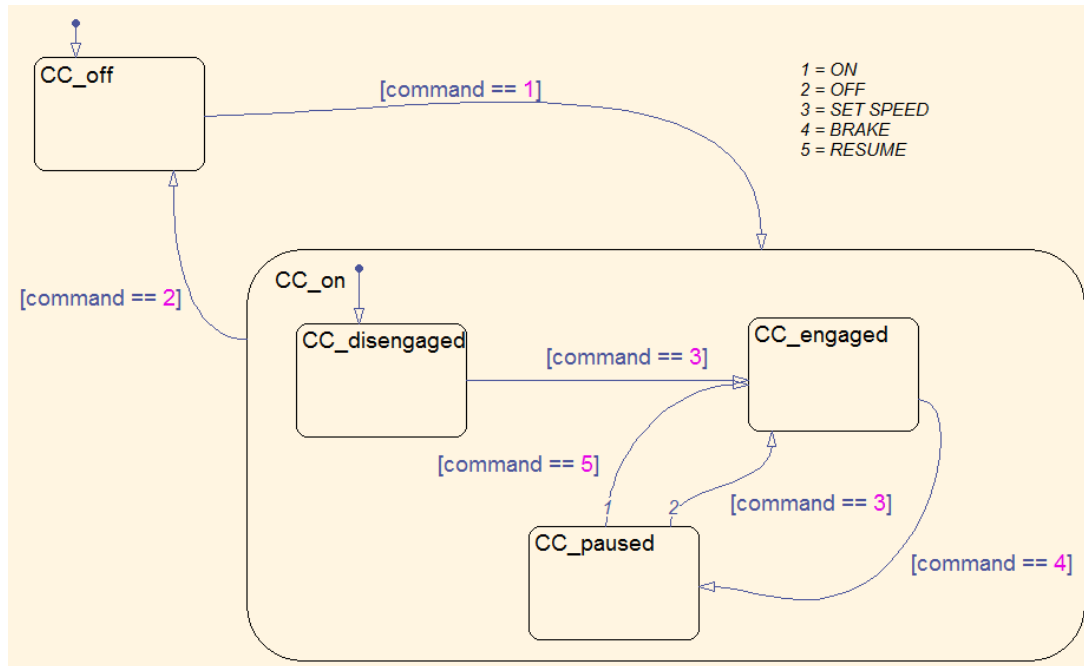


Figure 3 - Cruise Controller example model

Polyglot includes a user interface extension to Matlab that allows properties to be defined easily. Right click anywhere on the Stateflow model, select “MTV Properties...” from the context menu, and then select “Add Property.” There are 5 types of patterns that can be added:

- Absence: an event or configuration should never happen during some scope
- Universality: an event or configuration should always hold during some scope
- Existence: an event or configuration should hold at least once during some scope
- Response: an event or configuration should be followed by another event or configuration during some scope
- Precedence: an event or configuration should be preceded by another event or configuration during some scope

By “configuration,” we mean a valuation of model variables and a set of active states.

In this example, we will define two properties for the model: one that states that the model should never be in the CC\_on state if the value command is OFF (command == 2), and one that states that the model should always enter the CC\_engaged state before entering the CC\_paused state. These two properties are already defined in the included sample, but we will walk through the steps of how to define them.

Step one: define a new absence property.

- Right click on the model.



-From the context menu, select “MTV Properties...” > Add Property > Absence. A dialog box appears as shown below in Figure 4.

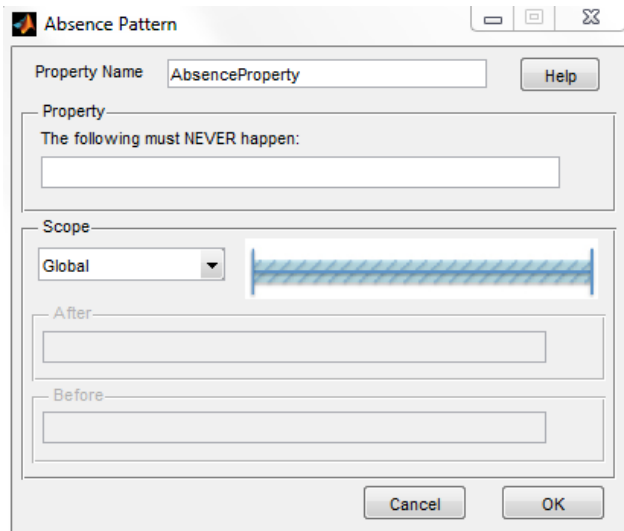


Figure 4 - Absence property dialog box.

-Fill in the dialog box as shown below in Figure 5. This states that the model should never be in state CC\_on if command == 2. The Global scope says that this property should hold at the end of all steps.

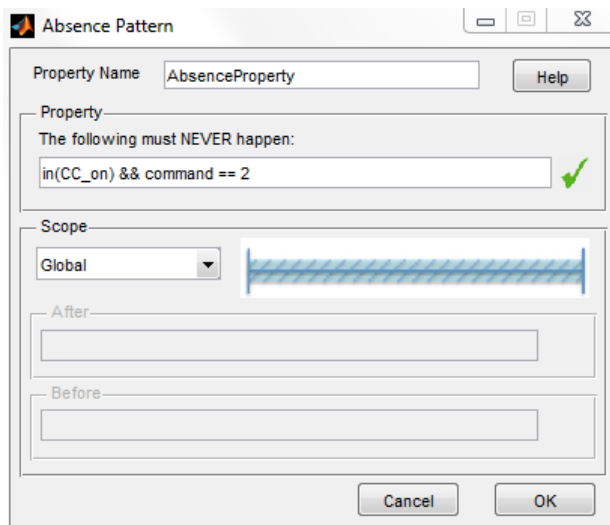


Figure 5 - Absence property stating the model should never be in state CC\_on while command is equal to 2.

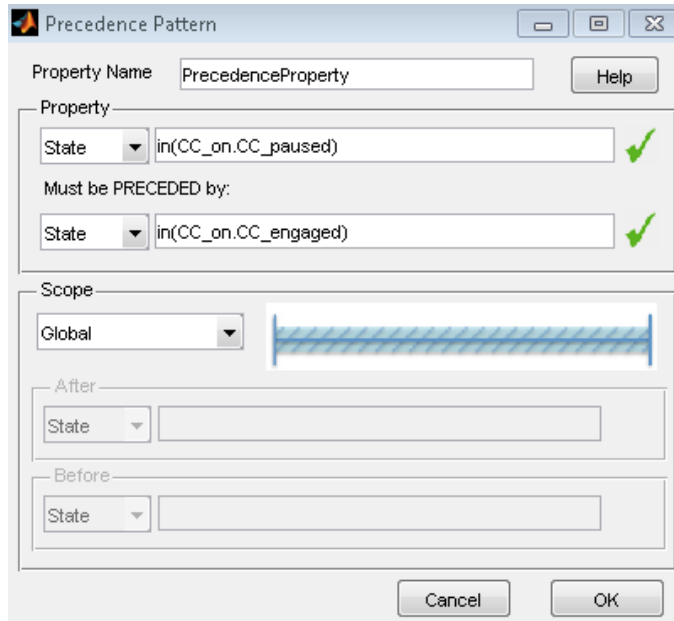
-Press “OK”.

Step two: define a new precedence property

-Right click on the model.

-From the context menu, select “MTV Properties...” > Add Property > Precedence. A dialog box appears.

-Fill in the dialog box as shown below in Figure 6.



**Figure 6 - Precedence property stating that state CC\_paused should always be preceded by CC\_engaged. Notice that state names must be fully qualified.**

Step 3: generate code

At the command prompt, type the following to generate the Java code for this model:

```
> MDL2MGA.exe -n cruise_control.mdl  
> StateMachineCG.exe cruise_control.xml
```

This will generate 4 files:

- Absence1.java
- Precedence2.java
- cruise\_controlReader.java
- TopLevelcruise\_control.java

Absence1.java and Precedence2.java contain the logic of the two temporal properties that were defined above.

Step 4: run the code

The cruisecontroller directory contains a sample .jpf configuration file named "ccconf.jpf" that you can use to run this sample with Symbolic Pathfinder (SPF). You will need to replace the value of the classpath variable in the file with the appropriate values for your system (e.g., give the path to the cruisecontroller class files, the Statemachines.jar file and JSAP-2.1.jar file).

Create an Eclipse project file for the code generated in Step 3 above, and add the .jpf configuration file to the project. You should be able to right click on the .jpf file inside Eclipse and select “Verify...”. After some time, SPF should report that no errors were detected; this is because the properties we defined are always satisfied. To see an example of how SPF can discover a path in which a property is not satisfied, go back to the Absence property defined in Step 1 above and for the property, give the following: “in(CC\_on) && command == 3”. Here we are purposely giving a property that is not always satisfied; the Statechart can be in the CC\_on state and the value of command can be 3 (SET\_SPEED). Repeat step 3 to generate code, and run it again with SPF inside Eclipse (right-click on the .jpf file and select Verify...). SPF should report that an error was found. The generated unit tests and method sequences will show the sequence of input values that led to the property violation.

## Technical Details of the Tool chain

### Specifying Properties in Simulink/Stateflow

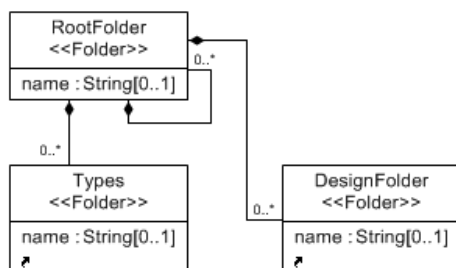
The Property Specification is based on the related work of Matthew B. Dwyer; the dedicated website is <http://patterns.projects.cis.ksu.edu>. Our GUI however does not support all the patterns listed on the website. Currently we only support the Absence, Existence, Universality, Precedence and Response patterns. Our solution does support all the Scopes listed on the website, which are: Global, Before, After, Between and Until. The specified properties are saved in the description field of the Stateflow model as an XML string and thus do not interfere with normal behavior of MATLAB.

### State Machine Models in XML format

The state machine models (derived from Stateflow or Rhapsody) are stored in an XML file. The schema for the XML was automatically generated by the UDM tools<sup>1</sup>. Here we include the UML class diagrams that capture the concepts used by the translators and the appendix contains a copy of the XML schema. The XML schema has many additional entries that are not used by the translator.

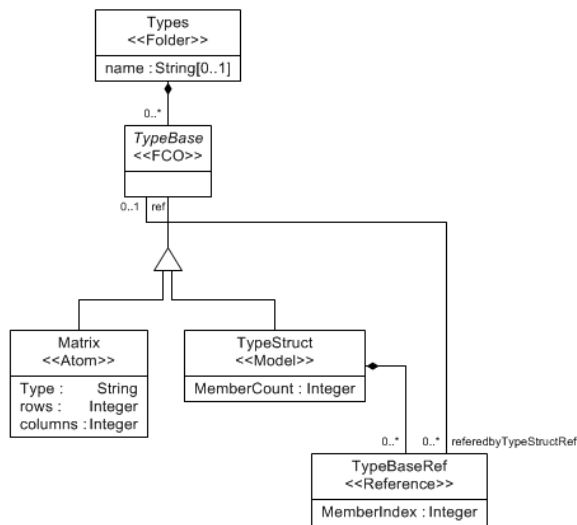
*While the XSD-compliant XML files can be created using any XML tool, we recommend to use the UDM toolsuite. The tools will generate C++ code for creating, accessing, and navigating in the XML files.*

The root of the XML file is a RootFolder:

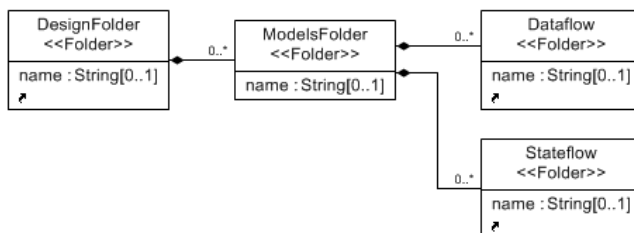


<sup>1</sup> The UDM tools are available from <http://repo.isis.vanderbilt.edu>. In the class diagrams the stereotypes shall be ignored, and the optional attributes are denoted by a ‘name : type[0..1]’ syntax. Connections typically represent association classes, while ‘...Refs’ are pointer-like classes that refer to other classes.

The Types folder contains type definitions: matrices and structures, the latter may contain references (that refer to other types, i.e. matrices, and structures).



The Design folder holds Models folders that contain the Dataflow and Stateflow models.



Dataflow models are for holding Simulink models -- these are not used by the subsequent StateMachineCG. The class diagrams up to this point are for representing model containment. The most important class diagram for the tool chain is the next: the class diagram for state machine models.

Stateflow models hold the state machines (show on the next full page). The key container here is the 'Stateflow' folder that contains States. States contain Data or Events (both of which may contain a reference to a TypeBase). States may also contain embedded MATLAB functions (EMFunction) and TruthTables, as well as Transitions that link TransConnectors. TransConnector is a base class for States and state-like elements, including: Junctions, starting points for transitions (TransStart), History (which is sub-classed into DeepHistory and ShallowHistory), Join, Fork, Choice, Entry, Exit, Terminate and Conditional elements. TransConnectors can also be referenced by ConnectorRefs.

This class hierarchy was reverse engineered from the Stateflow modeling language, and the Rhapsody and UML standard State Machines. The class hierarchy is capable of holding models of any of these three state machine modeling language variants.

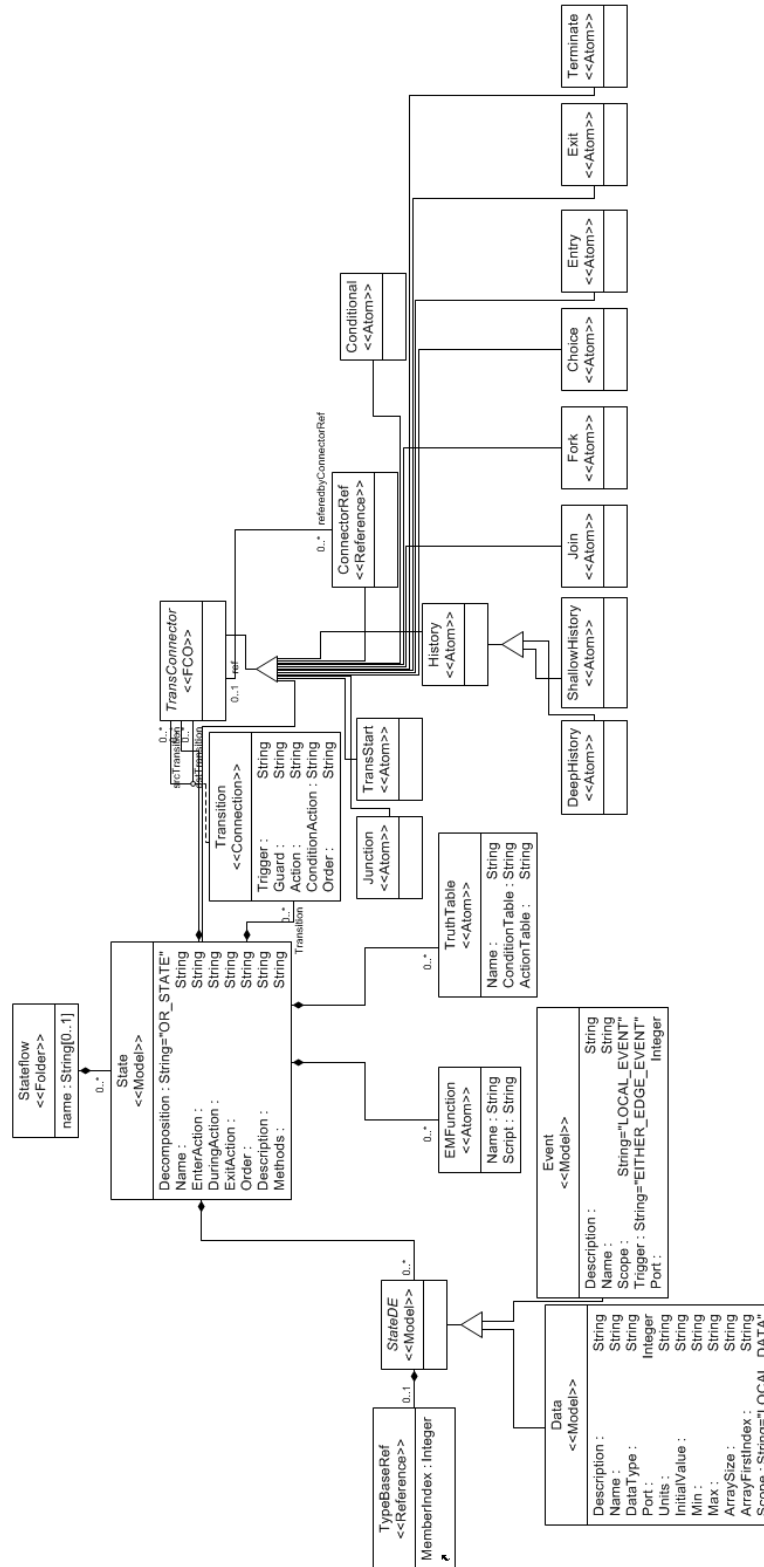


Figure 7: UML class diagram for state machines.

The release includes the UML class diagram model for the XML format used<sup>2</sup>, as well as the generated XML schema file. However, we suggest using the UDM libraries if one wants to generate (or read) XML files of the desired format.

## Model in Java format

Statechart models can be coded by hand in Java using the concepts of the Polyglot framework, but this is a tedious process. The best approach is to create the models using the intermediate ESMoL language and then use the StateMachineCG translator to automatically generate the Java representation of the Statechart. This also allows the Java code that is responsible for setting the inputs of the Statechart model to be automatically generated. However, Statecharts can also be created and/or modified on the fly, as outlined below.

### Class Hierarchy

The following is a list of the Classes in the Polyglot framework arranged in their class inheritance hierarchy. For specific details on any of the Java classes, please consult the documentation in \$(INSTALL\_DIR)\doc\html\index.html.

```
edu.vanderbilt.isis.sm.Event
edu.vanderbilt.isis.sm.IDataPrinter
    edu.vanderbilt.isis.sm.CSVDataProvider
    edu.vanderbilt.isis.sm.SymbolicDataProvider
edu.vanderbilt.isis.sm.IDataProvider
    edu.vanderbilt.isis.sm.CommandLineDataProvider
    edu.vanderbilt.isis.sm.CSVDataProvider
    edu.vanderbilt.isis.sm.SymbolicDataProvider
edu.vanderbilt.isis.sm.IDataReader
edu.vanderbilt.isis.sm.IEventHandler
    edu.vanderbilt.isis.sm.Interpreter
        edu.vanderbilt.isis.sm.RhapsodyInterpreter
        edu.vanderbilt.isis.sm.StateflowInterpreter
        edu.vanderbilt.isis.sm.UMLInterpreter
edu.vanderbilt.isis.sm.ILooper
    edu.vanderbilt.isis.sm.DefaultLooper
    edu.vanderbilt.isis.sm.NondeterministicLooper
edu.vanderbilt.isis.sm.IPseudoParent
    edu.vanderbilt.isis.sm.Region
    edu.vanderbilt.isis.sm.State
edu.vanderbilt.isis.sm.IRegionParent
    edu.vanderbilt.isis.sm.State
    edu.vanderbilt.isis.sm.StateMachine
edu.vanderbilt.isis.sm.IVertex
    edu.vanderbilt.isis.sm.Pseudostate
    edu.vanderbilt.isis.sm.State
edu.vanderbilt.isis.sm.properties.Pattern
    edu.vanderbilt.isis.sm.properties.Absence
```

---

<sup>2</sup> The class diagram model is in the XML format used by the Generic Modeling Environment (GME), available from <http://repo.isis.vanderbilt.edu>, using the UML Modeling paradigm.

```
edu.vanderbilt.isis.sm.properties.Existence
edu.vanderbilt.isis.sm.properties.Precedence
edu.vanderbilt.isis.sm.properties.Response
edu.vanderbilt.isis.sm.properties.Universality
edu.vanderbilt.isis.sm.properties.PropertyException
edu.vanderbilt.isis.sm.properties.Scope
edu.vanderbilt.isis.sm.properties.AfterScope
edu.vanderbilt.isis.sm.properties.BeforeScope
edu.vanderbilt.isis.sm.properties.BetweenScope
edu.vanderbilt.isis.sm.properties.GlobalScope
edu.vanderbilt.isis.sm.properties.UntilScope
edu.vanderbilt.isis.sm.Transition
```

### **StateMachine class**

The *StateMachine* class is the Java equivalent of a Statechart model. As the Doxygen generated documentation shows, a *StateMachine* can contain *Regions*. The *Region* class is the concept used to create sequential or parallel States. All States are contained inside exactly one *Region*, and a *Region* is contained inside exactly one State, except for top level *Regions*, which are contained directly by an instance of the StateMachine class. The semantics are that exactly one State in a *Region* is active when that *Region* is active, and when a State is active, all of its *Regions* are active. Thus, to implement parallel states (i.e., AND-states), a State should contain multiple *Regions*, and sequential states (i.e, OR-states) should contain exactly one *Region*. Simple states that contain no other states should contain no *Regions*. A state's containing *Region* should be passed as an argument to the constructor when it is instantiated.

### **Region class**

Instances of the Region class are used to represent parallel or sequential decomposition of a State. Every instance of a State is contained inside a Region. States, in turn, contain some number of Regions. If a State has no Regions (and hence no States), it is a simple State. If a State has exactly one Region, then it is considered a sequential (i.e., OR-) state. If a State has more than one Region, it is a parallel (i.e., AND-) state. When a Region is active, exactly one of its contained states is active, and when a State is active, all of its contained regions are active.

### **State class**

If a State needs to implement custom functionality, such as containing a variable or performing an entry action, then a class that is derived from the base class State should be created. In this way, States can contain data members, and entry, exit and during actions of a State can be implemented by overriding the virtual methods *entryAction()*, *exitAction()* and *doAction()* of the State class. These members are documented in the doxygen generated documentation. The Region containing a State is passed as the second parameter to the constructor.

### **Pseudostate class**

Pseudostates such as junctions and initial-pseudostates are implemented by the *Pseudostate* class. The specific type of pseudostate is given by an enumeration value, *Pseudostate.Kind*, that is passed to the constructor of *Pseudostate*. The *Pseudostate* class should not be derived, as there is no custom functionality that belongs to a *Pseudostate*. The parent element of a *Pseudostate* is passed as the first argument to the constructor.

### **Event class**

The *Event* class represents the concept of an Event using strings. Events can be thought of as a wrapper around the name of an Event.

### **Transition class**

Transitions are implemented by the *Transition* class. Custom functionality, such as guards, transition actions and condition actions can be implemented by overriding the virtual methods *guard()*, *action(IEventHolder e)* and *conditionAction(IEventHolder e)*, respectively. The *IEventHolder* object allows the action or condition action to emit an event that can be processed by the interpreter at a certain time depending on the semantics. Thus, the *IEventHolder* object can be thought of as one of the semantic interpreters. The source, target and containing *Region* of a transition should be passed to the *Transition* constructor upon creation, except for the case where the transition is instantiated before its source or target. In these cases, the *initialize(IVertex source, IVertex target)* should be called to set the source and target of the transition before it is used. A list of events that trigger a transition should also be passed to the constructor upon creation.

## **The Polyglot Engine**

The Polyglot engine is based around the idea of decoupling a Statechart's structural representation from its execution semantics. The structural features of a Statechart model are built by instantiating a set of base/derived classes, and then a *semantic interpreter* provides an interpretation to this structure using a particular variant of Statechart semantics. This allows a single Statechart model to be easily executed using different semantic variants.

Additionally, the Polyglot engine can be executed within the context of Symbolic Pathfinder (SPF), the symbolic execution engine of Java Pathfinder (JPF), which is a software model checker for Java. SPF provides a mechanism for automated test-case generation, in which sequences of inputs that drive a Statechart model through different execution paths can be automatically generated.

### **Driving the semantic modules**

The semantic modules (described in the Section below) implement the execution semantics of the different Statechart variants. To perform a step of execution with one of these semantic modules, the *step()* method, defined in the class *Interpreter*, is called. To simplify the execution, an interface named *ILooper* has been defined, along with a class named *DefaultLooper* that implements the *ILooper* interface. The *DefaultLooper* class implements the *doEventLoop* method and the *doDataAndEventLoop*



method, which read events and data from a data source, perform a step of execution and check any user defined properties.

## Semantic modules

There are currently three semantic modules, or interpreters: one implementing the UML semantics of Statecharts, one implementing the Rhapsody semantics of Statecharts and one implementing the Stateflow semantics for Statecharts. These semantic modules interpret the Java representation of a Statechart model (described above) using one of these semantic variants. That is, they provide a way to execute the Java representation of a Statechart model according to different semantics.

Each module is derived from the base class *Interpreter* and overrides the virtual method `step()` to implement a single execution step of a Statechart model. The various Statechart semantics all perform some variant of the following during a logical execution step:

1. Traverse the State hierarchy.
2. At each level of hierarchy, compute transition paths by testing all parallel states at that level for enabled transitions.
3. Perform actions associated with any computed transition paths.
4. Updated the state configuration.

The best way to understand how to implement a new semantic module (for instance, to provide an execution mechanism for a new dialect of Statecharts) is to study the code in the three included semantic modules. At the minimum, the new semantic module should override the `step()` method from the *Interpreter* class. This overridden implementation should contain the logic of the implementation of an execution step of the new Statechart dialect, for instance, how the hierarchy should be traversed looking for enabled transitions, when transition actions should be performed, when to process generated events, etc.

### StateflowInterpreter module

The *StateflowInterpreter* class implements the Stateflow semantics for Statecharts. The basic execution of the `step` method is a top-down traversal of the State hierarchy. At each level, the interpreter checks all parallel states at that level of hierarchy to see if they have an enabled outgoing transition. Upon finding a level of hierarchy where at least one parallel state has an enabled outgoing transition, the hierarchy traversal ends, although all parallel states at that level are tested for an enabled outgoing transition.

Missing/disabled features: currently, histories are not supported because they are not part of the official MAAB guidelines. They may be enabled in the future.

### UMLInterpreter module

The *UMLInterpreter* class implements the UML semantics for Statecharts. The UML specification for State Machine diagrams is fuzzy in places, and this interpreter makes assumptions that are vague or unclear in the official specification.

### **RhapsodyInterpreter module**

The RhapsodyInterpreter class implements the Rhapsody semantics for Statecharts.

Missing/disabled features: The Rhapsody toolsuite supports a large number of features which are not supported by the Rhapsody2MGA interpreter which translates Rhapsody models into Java code used by the semantic modules. These include the ability to define arbitrary Java code that can be called during Statechart execution. If these features are needed, they must be added manually to the code generated by StatemachineCG after the Rhapsody2MGA interpreter is executed.