7/17/2014

Jet propulsion laboratory, California institue of technology

Cody Lanier

Corrina Gibson

Michael Bonnici

Robert Karban

COMODO User Guide

Model Transformation for Dummies

Table of Contents

Table of Contents 1

1. Introduction 2

1.1 Team Members 3

1.2 General Overview 3

2. Getting Started Guide 4

2.1 Requirements 4

2.2 Setting Java Environment Variables 4

2.3 Installing Xtend/Xpand Eclipse Plug-in 4

2.4 Installing Java Pathfinder Eclipse Plug-in 4

2.5 Importing the COMODO Project into Eclipse 5

2.6 Building COMODO 6

2.7 Importing the Java Pathfinder Projects into Eclipse 6

2.8 Building & Testing Java Pathfinder 6

3. Model Checking 7

3.1 Exporting the SysML Model 7

3.2 Editing the Build File 7

3.3 Executing COMODO 7

3.4 Importing the Java Project 8

3.4.1 Copy to New Location 8

3.4.2 Import Project 8

3.5 Verifying the Model 8

3.5.1 Preface to Running JPF 8

3.5.2 Running JPF 9

4. General Information 9

4.1 Input & Output 9

4.2 Common Practices 10

4.3 Future Capabilities 11

5. Detailed Overview 11

5.1 States 11

5.2 Variables 13

5.3 JavaScript 13

5.4 Action Language Helper 13

5.4.1 ALH.createObject 14

5.4.2 ALH.addValue 14

5.4.3 ALH.getValue 14

5.5 Guards 15

5.5.1 ALH.inState 15

5.5.2 Boolean Logic 15

5.6 Signal Events 15

5.7 Constraints 18

6. DEVELOPMENT 19

6.1 Programming Languages 19

6.2 Common Folders & Files 20

6.2.1 Understanding COMODO 20

6.2.2 Develop COMODO 20

6.3 Application Programming Interfaces 21

7. Appendices 21

7.1 APPENDIX A: Acknowledgements 21

7.2 APPENDIX B: Acronyms 22

7.3 APPENDIX C: Definitions 22

Introduction

In this section you can find the persons involved in the development lifecycle of the COMODO software as well as a general overview of what COMODO’s purpose is.

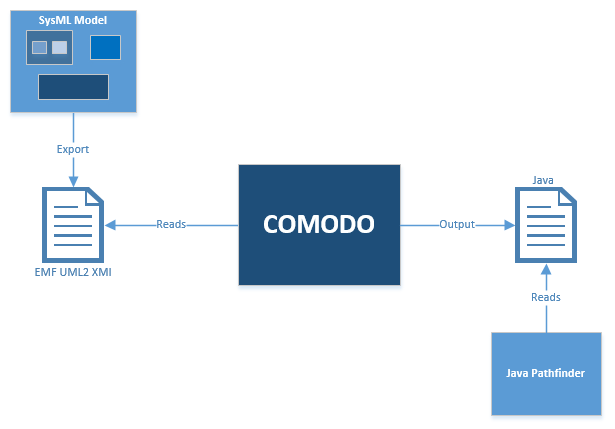
Team Members

|  |  |  |
| --- | --- | --- |
| Corrina Gibson | [corrina.l.gibson@jpl.nasa.gov](mailto:corrina.l.gibson@jpl.nasa.gov) | Systems Engineer |
| Michael Bonnici | [michael.f.bonnici@jpl.nasa.gov](mailto:michael.f.bonnici@jpl.nasa.gov) | Systems Engineer |
| Robert Karban | [robert.karban@jpl.nasa.gov](mailto:robert.karban@jpl.nasa.gov) | Software Developer |
| Luigi Andolfato | [landolfa@eso.org](mailto:landolfa@eso.org) | Software Developer |
| Cody Lanier | [cody.lanier@jpl.nasa.gov](mailto:cody.lanier@jpl.nasa.gov) | Software Developer Intern |
| Jessica Samuels | [jessica.a.collisson@jpl.nasa.gov](mailto:jessica.a.collisson@jpl.nasa.gov) | Group Supervisor |

General Overview

COMODO software was provided to JPL as an open-source software by the European Southern Observatory as an open source project.

COMODO is a model to text translation tool which translates an Eclipse Modeling Framework (EMF) Unified Modeling Language 2 (UML2) Extensible Markup Language Metadata Interchange (XMI) to text. It is important to note that because COMODO outputs to text, it can be adopted to translate a model to several different platforms and languages. For our purposes, a Systems Modeling Language (SysML) model is exported to an EMF UML2 XMI file. COMODO then takes that file and translates it into the equivalent Java representation. The Java file that is output by COMODO can then be used by Java Pathfinder (JPF) to perform checking and verification of the model. For more information related specifically to model checking, see the [Model Checking User Guide](https://gateway.jpl.nasa.gov/personal/cgibson/_layouts/15/WopiFrame.aspx?sourcedoc=/personal/cgibson/Documents/Shared%20with%20Everyone/SMAP%20FP%20MODEL%20AND%20MODEL%20CHECKING/SMAP%20FP%20Model%20And%20Model%20Checking%20Development/SMAP%20Model%20Checking/Model%20Checking%20Users%20Guide.docx&action=default).



Getting Started Guide

This section will guide you through setting up both COMODO and JPF on your computer. It is explicitly for a Windows Operating System machine, but a MAC Operating System should be very similar.

Requirements

* MagicDraw or a similar tool supporting the EMF UML2 XMI 2.x file format.
* Java 7 or higher
  + <https://www.java.com/en/>
* Eclipse IDE or equivalent
  + <http://www.eclipse.org/downloads/packages/eclipse-standard-432/keplersr2>
* Java Developer Kit 7 or higher
  + <http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>
* Git/Stash repository access
  + [susan.k.jones@jpl.nasa.gov](mailto:susan.k.jones@jpl.nasa.gov)

Setting Java Environment Variables

# ‘Start’ -> right-click ‘My Computer’ -> ‘Advanced System Settings’ -> ‘Environment Variables’.

# Create JAVA\_HOME variable and set to C:\Program Files\Java\<*Java JDK folder name here*>.

# NOTE: Program Files will be Program Files(x86) for 32-bit.

# Also, add C:\Program Files\Java\jre7\bin; to the beginning of the existing PATH variable.

Installing Xtend/Xpand Eclipse Plug-in

1. In Eclipse, Click ‘Help’ -> ‘Install New Software...’.
2. Paste <http://download.eclipse.org/modeling/m2t/xpand/updates/releases/> into the 'Work with:' field, then click ‘Add’ -> Give it a name, like “Xtend/Xpand” and click ‘OK’.
3. Check ‘M2T Xpand/Xtend-2.0.0’, or a later version, then click ‘Next’.
4. Click ‘Next’ again, then accept the terms and conditions, and finally click ‘Finish’.

NOTE: A warning about installing unsigned content may be displayed. If so, click ’OK’.

1. Restart Eclipse... a pop up should prompt you to do so, but if not, close and reopen Eclipse manually.

Installing Java Pathfinder Eclipse Plug-in

1. In Eclipse, Click ‘Help’ -> ‘Install New Software...’.
2. Paste <http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/projects/eclipse-jpf/update> into the 'Work with:' field, then click ‘Add’ -> Give it a name, like “Java Pathfinder” and click ‘OK’.
3. Click ‘Select All’, then click ‘Next’.
4. Click ‘Next’ again, then accept the terms and conditions, and finally click ‘Finish’.

NOTE: A warning about installing unsigned content may be displayed. If so, click ’OK’.

1. Restart Eclipse... a pop up should prompt you to do so, but if not, close and reopen Eclipse manually.
2. Create a .jpf directory in your user/home directory, i.e. C:\Users\<*Your User Profile Name here*>\

NOTE: You will likely need to do this from command prompt

* 1. ‘Windows key + R key’ -> type “cmd” -> ‘Enter key’.
  2. Navigate to home/user directory, if not already there, and then type “mkdir .jpf” without the qoutes.

1. Back in Windows, inside the directory create a file 'site.properties' with the following lines:

*jpf.home = C:/Users/<user>/git/smap\_fp\_jpf/ESO/source/GSME/Platforms/JPFSC/Modules*

\*\*\*NOTE 1: The first line is just an example, your first line will be similar.

\*\*\*NOTE 2: Also, notice the forward slashes instead of backslashes.

**\*\*\*NOTE 3: DO NOT INCLUDE THESE 3 RED NOTES IN YOUR FILE**

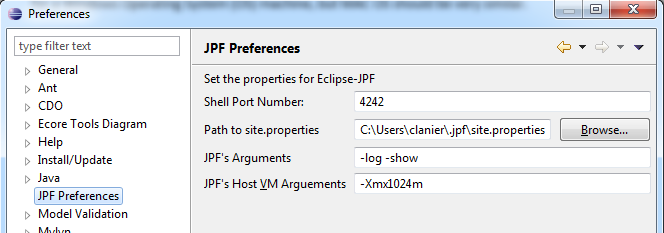
*jpf-core = ${jpf.home}/jpf-core*

*jpf-aprop = ${jpf.home}/jpf-aprop*

*jpf-statechart = ${jpf.home}/jpf-statechart*

*extensions = ${jpf-core},${jpf-aprop},${jpf-statechart}*

1. In Eclipse, goto ‘Window’ -> ‘Preferences’ -> ‘JPFPreferences’ -> make sure that your site.properties path is correct and input the rest of the information to match the below image:



Importing the COMODO Project into Eclipse

1. Navigate to <https://mbse.jpl.nasa.gov/stash/projects/MH/repos/smap_fp_jpf/browse>
2. Click on "Clone" in the upper left of the webpage, then right-click the highlighted text and click ‘Copy’.
3. In Eclipse, ‘File’ -> ‘Import’ -> ‘Git’ -> ‘Projects from Git’ -> ‘Next’ -> ‘Clone URI’ -> ‘Next’.
4. Since you just copied the URI to the clipboard in the previous step, the first 3 lines should be auto filled. If not, paste the URI into the 'URI' field and the rest should populate automatically.
5. Type in your password in the 'Password' field, click the checkmark next to 'Store in Secure Store', then click "Next".
6. Make sure all items are check-marked. If not, click "Select All".
7. Change the 'Initial branch' drop-down from 'master' to 'feature/SMAP-FP-JPF-Dev' and click ‘Next’. When it's done loading click ‘Next’ again.
8. Make sure ‘Find nested projects’ is checked -> click 'Deselect All' -> find and check ‘comodo’ (there are 2, one in all caps and one not… You want the one in lowercase) -> click ‘Finish’.

Building COMODO

1. In Eclipse, expand the comodo/src/ directory.
2. Right-click (ctrl-click on Mac) on build.xml and select ‘Run As’ -> ‘2 Ant build...’.
3. Make sure ‘build’ is the only thing checked, and that jre7 is selected from the 'JRE' tab, then click on ‘Run’.

NOTE: If jre7 is not an option, you'll need to add the path “C:\Program Files\Java\jre7” to the list of installed JREs

1. Should result in ‘Build Success’ in the Console output window of Eclipse.

Importing the Java Pathfinder Projects into Eclipse

1. Right-click some empty white space in Eclipse’s Package Explorer window (should be the left navigation pane), then click ‘Import’.
2. Select ‘Projects from Git’ -> ‘Next’ - > ‘Existing Local Repository’ -> ‘Next’ - > make sure ‘smap\_fp\_jpf’ is selected -> ‘Next’
3. Navigate to ‘ESO/source/GSME/Platforms/JPFSC/Modules/’ then select the ‘jpf-core’ folder and click ‘Next’ -> ‘Finish’.
4. Repeat steps 1 through 3 but you will select ‘jpf-aprop’ instead of ‘jpf-core’.
5. Repeat steps 1 through 3 but you will select ‘jpf-statechart’ instead of ‘jpf-aprop’.

Building & Testing Java Pathfinder

1. Expand the ‘jpf-core’ project and right-click on the ‘build.xml’ file, then select "2 Ant build..." -> check 'test' -> click ‘Apply’ -> click ‘Run’.

NOTE: may receive one error during the URLClassloader tests.

1. Repeat step 1, but for the ‘jpf-aprop’ project.

NOTE: may receive one error during the Sandbox tests.

1. Repeat step 1, but for the ‘jpf-statechart’ project.

NOTE: checking ‘test’ may be unnecessary because I do not think ‘jpf-statechart’ currently has any tests.

Model Checking

COMODO is an intermediary tool… a way to get to the end goal, which is model checking. Model checking allows verification and validation of a model by using Java Pathfinder. This section guides you through start to finish of model checking one or multiple models.

Exporting the SysML Model

These next instructions are provided based on the modeling software currently in use at the NASA JPL, known as MagicDraw.

1. Commit the project to the Teamwork server
2. Save the project locally with no teamwork information to ‘comodo\test\models\<*model\_name*> (you want this instance of the model saved so you know exactly what JPF will be model checking)
3. Run the **Convert Element Value Plugin** in MagicDraw (Modifies the Constraints so they will export in the next step for coversion to Java Assertions). You can get this plugin from Stash>Model Helpers>SMAP FP Plugins.
4. In MagicDraw, click ‘File’ -> mouse over ‘Export To’ -> click ‘Eclipse UML2 (v2.x) XMI File’
5. You must export to ‘comodo\test\models\<*model\_name*>
6. Close the MagicDraw project without saving changes (you do not want to save the modifications that were made using Convert Element Value Plugin, this is just a type change in the constraints to make them exportable).

Editing the Build File

Do this section only if your model has never been run with COMODO under its current name

1. In Eclipse, in ‘comodo\test\’, double-click ‘build.xml’ to open it in the editor
2. Scroll about half way down in the file and stop when you see ‘<!-- Examples for Java PathFinder platform !-->’
3. Copy one 8 line block of code, located below the line from step 2, from ‘<target’ all the way to ‘</target>’ and paste it in approximately the same location. Below the previously copied text is a good place to put it.
4. Change ‘<target name=”<*copied\_model\_name*>” ’ to be ‘<target name=”<*your\_model\_name>*” ‘
5. There should be four more places to overwrite the copied model name with your model name in the target block
6. Save the file and close it

Executing COMODO

When a model is selected for execution in COMODO’s build file, it will run that model as long as it can find the model in the ‘comodo\test\models\’ folder location.

1. In Eclipse, in ‘comodo\test\’, right-click ‘build.xml’ -> mouse over ‘Run as’ -> click ‘2 Ant Build…’
2. Find your model name and check mark it, if not already checked. If you do not see your model name, repeat 4.2 Adding the Model to COMODO

NOTE: You can check mark more than one model to run multiple models back to back, but they must be located in the ‘comodo\test\models\’ folder

1. Your model name should now be listed in the ‘Target execution order:’ of the current window
2. Click ‘Apply’, if not grayed out -> click ‘Run’

NOTE: If you know your model to be already listed in the ‘Target execution order’ of COMODO, you can skip all of this and simply click the ‘build.xml’ file to select it, and then click the ‘Run’ button from Eclipse’s main screen, or right-click the ‘build.xml’ file and select ‘Run as Ant Build 1…’

Importing the Java Project

COMODO outputs the translated model to the ‘comodo\test\output\JPFSC\’ location. You must now import it as a project in order to use JPF on it. For several reasons, we first copy it to a new location and import it from there.

Copy to New Location

1. Using Windows Explorer (or Finder), locate ‘comodo\test\output\JPFSC\<*your\_model\_name*>’ and copy the ‘<*your\_model\_name*>’ folder

NOTE: The full path will likely be: ‘C:\Users\<*user\_name*>\git\smap\_fp\_jpf\ESO\source\GSME\tools\comodo\test\output\JPFSC\’

1. Paste the previously copied ‘<*your\_model\_name*>’ folder to: ‘<*repository\_root*>\smap\_fp\_jpf\ESO\source\GSME\Platforms\JPFSC\Examples\’

NOTE: The respository root is likely in ‘C:\<*user\_name*>\git\’

Import Project

1. In Eclipse, click ’File’ -> click ‘Import’
2. In the new window, under the ‘General’ folder, select ‘Existing projects into Workspace’ -> click ‘Next’
3. Click ‘Browse’ and navigate to: ‘<*repository\_root*>\smap\_fp\_jpf\ESO\source\GSME\Platforms\JPFSC\Examples\’

NOTE: The repository root is likely in ‘C:\<*user\_name*>\git\’

1. Select the ‘<*your\_model\_name*>’ folder -> click ‘OK’ -> Click ‘Finish’

Verifying the Model

Now that the model has been successfully transformed and imported as a Java project, JPF can perform model checking on it.

Preface to Running JPF

* You need to comment out all parts of the model that you don’t want to run in the model checker
* Comment out or fix the Java syntax for any code that is causing errors in the transformed code.

NOTE: If COMODO worked correctly, there will be no errors that need commenting

* Add the assertions that you want to check

Running JPF

1. In Eclipse, double-click the newly imported project’s ‘<*model\_name*>.jpf’ file to open it in the editor, and make sure to change ‘target\_args’ to ‘target.args’
2. Right-click the project’s ‘build.xml’ file in ‘<*model\_name*>\src\’ -> mouse-over ‘Run as’ -> click ‘Ant Build 2…’

NOTE: ‘build [default]’ should be the only thing checked on the ‘Targets’ tab

1. Click the ‘JRE’ tab -> change the ‘Runetime JRE:’ to ‘jre7’ is currently set to ‘jre6’
2. Click ‘Apply’, if not grayed out -> click ‘Run’

NOTE: You should see ‘BUILD SUCCESSFUL’ in the console output text

1. Within the project, you can now right-click the <*model\_name*>.jpf file and then click ‘Verify’ to start JPF

General Information

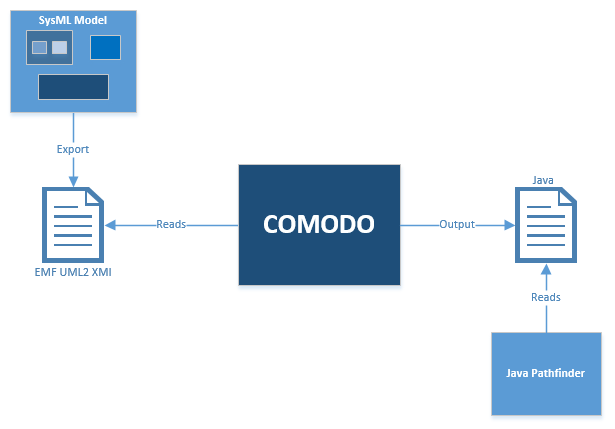
This section is intended for those seeking a more in-depth understanding of how COMODO works as well as a detailed description of the translations it performs.

Input & Output

COMODO is a model to text transformation tool, and therefore, can be used to output many different file types. For NASA JPL purposes, we are concerned with Java output files since this is what JPF requires as input for model checking.

COMODO uses an EMF UML2 XMI file as input. This file can be created from a SysML model by exporting the model to that format. The actual file used is ‘<*model\_name*>.uml’ and it is structured very similarly to an XML file. By default, COMODO looks in the ‘comodo/test/models/’ directory for the models and XMI files that it is tasked to translate. In order to set a model as COMODO’s input, you must add the model to the COMODO build.xml file and run it as a custom Ant build (See [Model Checking : Editing the Build File](#EditBuildFile)). COMODO can translate multiple models in one execution, but does so in the order specified by you.

Once COMODO has finished executing, a Java project containing a single Java file representing each translated model can be found in the ‘comodo/test/output/’ folder. If you execute COMODO in accordance with this guide, the newly created project(s) and Java file(s) will have the same name as the input model(s). JPF can now use COMODO’s output Java file to perform model checking.



Common Practices

The practices and conventions in place at the time of this writing are mentioned below. This does not mean that they cannot ever change and there is no guarantee that this list is 100% accurate with current known practices. If practices and/or conventions change, this list should be updated. COMODO can be quite flexible, but it may need editing or may not work correctly if the most current practices and conventions are not followed.

* All JavaScript var references must be equivalent to Java int variables
  + i.e. no Booleans
* All multi-dimensional Arrays have only 2-dimensions
* All multi-dimensional Arrays are Integer Arrays at the 2nd level
  + Ex. A1 is an Array that holds an Array of Integers
* The 2nd level Array is a temporary Array and, in the model, shall be named “<*main\_array\_name*>\_TEMP”
* Signal port names can’t use abbreviated names and must be prefixed by FROM\_ or TO\_
  + i.e. the full state machine name where the signal can be found must be used
  + Ex. TO\_ERROR\_MONITOR instead of TO\_MON
* Signals referenced with a ALH.sendSignal command must exist in the model of the sendSignal command should be commented out
* All ALH.sendSignal commands must end with a semicolon
* There can be no typos in the path of the ALH.inState guard paths
  + i.e. the path must exist in the model
* Signal names are constructed from the owning region and state names in the form r\_<region>\_s\_<state>\_signal\_<signal\_name> and the constructed name must be unique within each state machine
* Named elements cannot have names with spaces. Use underscores instead

Future Capabilities

This section summarizes a set of future capabilities that may be integrated into the COMODO model transformation tool.

* Extend COMODO to transform additional types of models (i.e. activity diagrams)
* Extend COMODO to transform models to additional languages (COMODO already translates to Java, C++, XML, IDL, python, text, SCXML, TCL)
* Extend COMODO to implement internal vs external Events via stereotypes from the model so Java Pathfinder is constrained to injecting only the external events and understand sendEvents to explore the internal events. Work in Java Pathfinder needs to be completed before COMODO should implement this capability.
* Extend COMODO to generate assertions and place them in the correct locations in the model given specifications/requirements as input. (We are currently working on making COMODO interpret constraints from the MagicDraw model and transform the constraints to Java assertions)
* Extend COMODO to implement Linear Temporal Logic (LTL) assertions. Work in JPF needs to be completed before COMODO should implement this capability.

Detailed Overview

The COMODO tool uses a combination of UML, Xpand, and Xtend to navigate and convert an exported model to its associated Java code. The process involves starting at a very high level of the model and then digging down with loops within loops until all aspects of the model have been visited and correctly converted. It begins by starting an outer loop through all state machines, then loops the states within each. While inside the states, all actions are looped and expanded, and finally all transitions within the states get explored with another loop. This section provides details of how each part of the translation is accomplished.

States

The highest level of the model is the state machines and COMODO begins, as previously stated, by looping through all of them. Within this initial outer loop, each state machine is converted to a public Java Class, which defines the state machine using the following format:

**public** **class** SM\_<*state\_machine\_name*> **extends** State {

}

After resolving the name of the state machine, each state within the current state machine is found and looped through. This process of loops within loops continues to dig down into each subsequent level of sub-states until all states have been visited. For each state, a Java Class is created using the following format:

**public** **class** S\_<*state\_name*> **extends** State {

}

This class will be found INSIDE of the previously created state machine. Java requires that all class names be unique, so in the model, all states that are within the same level of a state machine must have unique names. If not, errors will exist in the Java file. Originally COMODO was developed to create unique state names by combining the owing region name with the state name. It was later found that the class names became too large for the operating system to create when very large models that have many levels of states within states are transformed.

At the end of the loops, you’ll have many classes, and many classes within classes. They may even go down into several sub-levels of classes. The top-most level classes are the state machines and all the other classes and sub-classes end up being the states that make up those state machines.

The classes are just definitions of the objects. In order to be used, these object must be created, also known as instantiated. An example of a translated state machine, defined and instantiated, containing three states, including one within another, is shown below:

**public** **class** SM\_ABC **extends** State {

**final** S\_A s\_A = **new** S\_A();

**final** S\_B s\_B = **new** S\_B();

**public** **class** S\_A **extends** State {

}

**public** **class** S\_B **extends** State {

**final** S\_C s\_C = **new** S\_C();

**public** **class** S\_C **extends** State {

}

}

}

COMODO instantiates each state machine and state object with a “final” designation, which means the object is in its final form and cannot be fundamentally changed. After “final”, is the type of the object, a.k.a. Class, followed by the name that the object will be referenced by. After the equals symbol is the Class again, preceded by “new”, which means that the referenced object will now be a new instance of that Class.

Variables

Each state visited may have value properties of Type Boolean or Integer. Some of these properties, a.k.a. variables, can have a multiplicity value of ‘\*’, meaning that they are an array of that Type. For the single instance variables, COMODO simply creates a Boolean or an int with the same name as the property. When COMODO finds that a property has multiplicity, an ArrayList of that Type is created. ArrayList was chosen because it does not need a specific size when created, whereas a standard Java Array does requires the size to be specifically declared during creation.

JavaScript

Inside the body of each transition, resides a text block. COMODO looks for the text “var ” in the body’s text and replaces it with the text “int “. This relies upon the assumption that all explicitly declared vars, in the model, are meant to be of Type int. COMODO also replaces all println statements with “System.out.println”. Lastly, if the Math.floor function (used for rounding numbers down) is found in the model, COMODO will prefix this with the text “(int)” because Math.floor returns a Double, but an Integer is needed. This prefix will cast the result of Math.floor to an Integer.

All other JavaScript syntax is equivalent to Java, so no other manipulation of the text is required.

Action Language Helper

Currently, an action language for UML, which allows actions to be handled in an executable UML model, is still being developed. Action Language Helper (ALH) is a temporary solution to accessing UML actions and will eventually be replaced by UML’s action language, known as Action Language for Foundational UML (ALF).

After performing the translation of the text for the two simple JavaScript cases, COMODO looks for three possible uses of the Action Language Helper (ALH) syntax. Those uses are as follows:

* ALH.createObject
* ALH.addValue
* ALH.getValue

ALH.createObject

“ALH.createObject” has only one argument which represents the Type of an Array to create, and when COMODO finds this text, a new ArrayList of the given Type is instantiated. The developed pattern is that the argument will contain the Type plus the text “List”, like “BooleanList” or “IntegerList”. COMODO just removes “List” from the String to get the Type. Currently, all cases of ALH.createObject have “IntegerList” as the argument, which equates to Integer for Java purposes. Here is an example:

Original text: L = ALH.createObject(“IntegerList”);

New text: L = **new** ArrayList<Integer>();

ALH.addValue

After completing all ALH.createObject instances found in the body text, COMODO looks for and deals with “ALH.addValue” text. ALH.addValue is used for Arrays and has three arguments, but COMODO only cares about the first one and the third one. The first argument is the Array to add the value to, and the third argument is the value to add to that Array. COMODO builds a new String from those two arguments and replaces the original String with the new one. Here is an example of the translation:

Original text: ALH.addValue(L, “i”, RSP\_ID);

New text: L.add(RSP\_ID);

ALH.getValue

When “ALH.getValue” is found within a body’s text, the entire portion that must be removed is separated from the rest of the body text. Once removed and stored, the String is stripped down to only the two arguments, a.k.a. inputs, of ALH.getValue, which are the state machine that the property can be found in and the name it is referenced by. COMODO pieces these two items together to make the appropriate Java reference call to the property. The newly pieced together String replaces the original ALH.getValue String that was cut out from the body text previously. Here is an example:

Original text: ALH.getValue(ERROR\_MONITOR, “EM\_EM”) = true;

New text: sm\_ERROR\_MONITOR\_BEHAVIOR.EM\_EM = **true**;

The only addendum to this pattern is when ALH.getValue is found used with a multi-dimensional array. In this case, the String has a portion in its middle removed, before being put back into the text body.

Original text: if ( ALH.getValue(MON\_RSP\_MAP.get(j), “i”).get(1) == Q1 )

New text: **if** ( MON\_RSP\_MAP.get(j).get(1) == Q1 )

Guards

Some transitions have conditions, called guards, attached to them. This makes it so that the transition cannot be completed unless the guards are satisfied. This is desirable because without guards, JPF will take all transitions from each state every time that state is visited during the model checking. Often, we want to limit the possible transitions so that we can see how a model reacts to certain inputs, especially when modeling behavioral models.

The guards are stored in the model as a text represented path in the transition’s Opaque Expression body field. Guards will be of two possible forms; ALH.inState guards or simple Boolean logic guards.

ALH.inState

For ALH.inState guards, COMODO will build up the Java path correctly based off of the text path it originally sees. The ALH.inState text path is a pattern following one of three possible patterns:

1. *“<region>***::**<*state*>”
2. “<*state*>**::**<*region*>**::**<*state*>”
3. “<*state\_machine*>**,** <*state*>**::**<*region*>**::**<*state*>”

You can see in the following example that the regions are not needed since they were originally thrown out when the states were created. The state names are all that is required in order to find the unique paths through the state machines and states.

Original text: if( ALH.inState (RWA1\_BEHAVIOR,"RWA1::POWER:ON”) )

New text: **if** ( sm\_RWA1\_BEHAVIOR.s\_RWA1.s\_ON.isActive() )

Boolean Logic

Regular Boolean logic condition guards are left as is because they use JavaScript which does not need translating.

Signal Events

Signal events are quite complicated because multiple transitions may need to execute when one single signal is sent to a state machine. Again, JPF will take every path it is allowed to when it reaches a state, so we also need for those transitions to prevent JPF from trying to use the transition multiple times. Here is an example to help you visualize how COMODO solves this issue:

First, each Java file that COMODO outputs will have a Class defining a Signal object, which contains a boolean called signaled and a String called name. When Signal objects are instantiated, they must take in the name of the signal as an argument. The Signal Class looks like the following:

**public** **class** Signal {

String name;

**boolean** signaled;

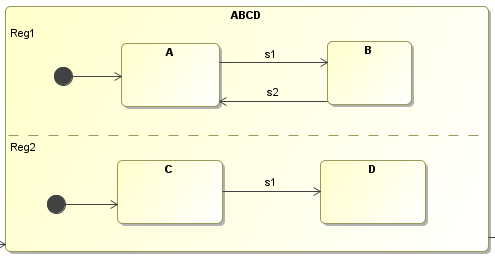
**public** Signal(String n) {

name = n;

signaled = **false**;

}

}



Within each state, every time a signal is found, a Java method is created using the signal’s name, and a Signal object is created using the signal’s owning region and state names. When the signal gets instantiated, it is also added to the owning state machine’s signal\_list for future reference. A guard is added to the signal method in order to keep JPF from transitioning until specifically told to. COMODO also adds a setNextState() method call to actually make the transition and virtually move to the next state once the guards are satisfied. The Java code produced from translating the above example would look like the following:

**public** **class** S\_A **extends** State {

**public** **void** s1() {

**if** (r\_Reg1\_s\_A\_signal\_s1.signaled == **true**) {

r\_Reg1\_s\_A\_signal\_s1.signaled = **false**;

setNextState(s\_B);

}

}

}

**public** **class** S\_B **extends** State {

**public** **void** s2() {

**if** (r\_Reg1\_s\_B\_signal\_s2.signaled == **true**) {

r\_Reg1\_s\_B\_signal\_s2.signaled = **false**;

setNextState(s\_A);

}

}

}

**public** **class** S\_C **extends** State {

**public** **void** s1() {

**if** (r\_Reg2\_s\_C\_signal\_s1.signaled == **true**) {

r\_Reg2\_s\_C\_signal\_s1.signaled = **false**;

setNextState(s\_D);

}

}

}

**public** **class** S\_D **extends** State {

}

Signal r\_Reg1\_s\_A\_signal\_s1 = **new** Signal("s1");

Signal r\_Reg1\_s\_B\_signal\_s2 = **new** Signal("s2");

Signal r\_Reg2\_s\_C\_signal\_s1 = **new** Signal("s1");

Signal[] signal\_list = { r\_Reg1\_s\_A\_signal\_s1,

r\_Reg1\_s\_B\_signal\_s2, r\_Reg2\_s\_C\_signal\_s1 };

ALH.sendSignal commands are used to execute signals during model simulation using the Cameo Simulation Toolkit. When COMODO comes across one of these commands, it translates ALH.sendSignal String into a Java for loop which loops through all Signals in the state machine. This for loop will set all signals that have the matching signal name to true. Let’s continue with our above example:

Original text: ALH.sendSignal(“s1”, TO\_ABCD);

New text: **for** (Signal s : sm\_ABCD.signal\_list) {

**if** (s.name.equals("s1")) {

s.signaled = **true**;

}

}

After the example sendSignal command is received, both s1 signals will be able to pass through their respective guards and set their own individual signaled variables back to false, thereby preventing JPF from taking the transitions more than once while also ensuring that both signals fire.

Constraints

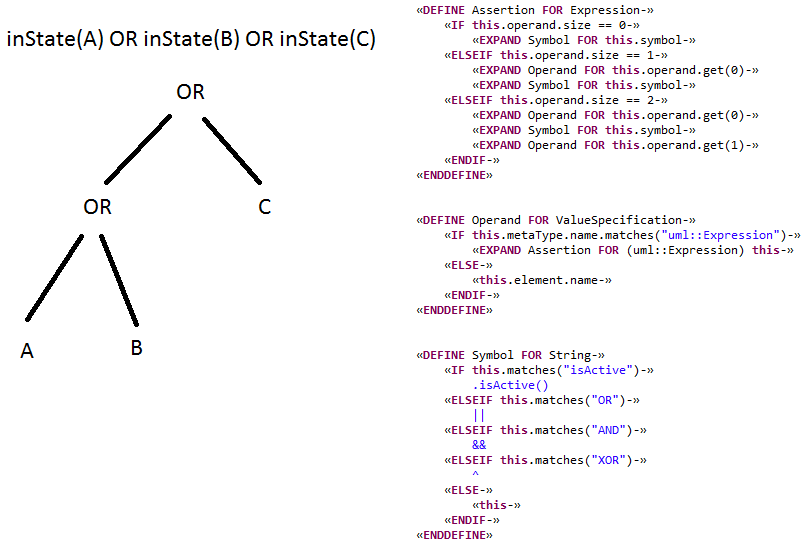
Because this is a behavioral model, we want to inject different scenarios in order to make sure that the model behaves the way it is supposed to or lets the user know that a requirement failed.

The constraints get modeled as Expressions within MagicDraw and the Expressions can be made up of more Expressions or Element Values. The Element Values actually reference model elements for use as operands in the constraints. However, Element Values are not exported to the XMI file that is used by COMODO so they must first be converted into Expressions by using a MagicDraw plugin called ‘Convert Element Values’ which can be found in the following repository location:

<https://mbse.jpl.nasa.gov/stash/projects/MH/repos/comodoplugins/browse>

The plugin takes an Element Value and creates an Expression in the same place as the Element Value. The Expression has a textual representation of the model element that is referenced by the Element Value, usually a qualified path to a State.

Once the plugin has run its course, the model can be exported to EMF UML2 XMI. During transformation, Expressions are found by finding the Dependency associated with them. The Dependency links to the Constraint Block from the model, which contains an Expression Tree. The Expression Tree is a binary tree that contains operands and operators. It is dealt with in a recursive manner and is probably easier to show than to explain. Here is an example of an Expression tree and the recursive function it goes through:



The result of the inState(A) OR inState(B) OR inState(C) constraint would be transformed into:

**assert** ( sm\_mySM.s\_mySTATE.s\_A.isActive()

|| sm\_mySM.s\_mySTATE.s\_B.isActive()

|| sm\_mySM.s\_mySTATE.s\_C.isActive() );

DEVELOPMENT

This section is for developers who want to add functionality to COMODO or for anyone who wants to see behind the curtain of the tool.

Programming Languages

Most of the development of COMODO was done using Xpand and Xtend. Those two are the bridge between UML and Java, but knowledge of several other languages is also necessary. The following is a list of all languages encountered during development:

* SysML
* UML
* Xpand
* Xtend
* JavaScript
* Java

Common Folders & Files

While there are many folders and files that make up the COMODO tool, only a handful are required in order to understand how the code works and change the way it functions.

Understanding COMODO

Start with the Xpand template files (XPT), ‘Code’ and ‘Fsms’. These may be some of the hardest to understand, but are responsible for looping through the model. ‘Fsms.xpt’ has a lot of the higher level looping, while ‘Code.xpt’ tends to deal with producing code at a more specific lower level. They are both very intuitive once you get used to the language. You can find the ones responsible for creating code for JPF in ‘comodo\src\comodo\templates\jpfsc\java\’.

The XPT files, located in two folders within ‘comodo\src\comodo\xtend\’ make calls to several Xtend functions found in the Xtend files (EXT). These functions use UML heavily and are mostly getters, condition functions, or String manipulators. They will probably look a bit more familiar to you since they resemble common languages like Java. In fact, for more complex functions, straight Java code can be developed and tied in to these files. Take a look at ‘helper.ext’ specifically, to see how the Java methods are integrated. You’ll also want to get familiar with ‘statemachine.ext’ to get a good understanding of many common functions that are used throughout the previously mentioned Xpand files.

The Java functions are grouped together in a file called ‘JavaExtension.java’ located in ‘comodo\src\comodo\util\’. Much of the actual translating is going on in this file.

Develop COMODO

Any of the files mentioned in the previous section are good candidates to edit, if changes in COMODO are to be developed. Try to stick to the formats and groupings that you see within them and make comments where necessary to show your changes. Again, ‘JavaExtension.java’ will be necessary for your more complex methods, as Xpand/Xtend have limited capability. Keep in mind that Xpand does not have while loops, so some creativity is sometimes necessary.

If new modeling operators are introduced for use in modeling Constraints (see 5.7 Constraints), the new operators can be added to COMODO via the Symbol function in the Code.xpt file.

Application Programming Interfaces

This section will refer you to all known documentation, Application Programming Interfaces (APIs), references, and other links for items related to development of COMODO.

* Java 7 API

<http://docs.oracle.com/javase/7/docs/api/>

* Xpand documentation

<http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.xpand.doc%2Fhelp%2Findex.html>

* Xtend2 API

<http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.xpand.doc%2Fhelp%2Fxpand_reference_introduction.html>

* UML2 API

<http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.uml2.doc%2Freferences%2Fjavadoc%2Forg%2Feclipse%2Fuml2%2Fuml%2Fpackage-summary.html>

* MagicDraw API

*<MagicDraw\_installation\_root>/*openapi/docs/javadoc/index.html

* Cameo Simulation Toolkit API

*<MagicDraw\_installation\_root>/*manual/Cameo%20Simulation%20Toolkit%20API%20User%20Guide.pdf

Appendices

APPENDIX A: Acknowledgements

The following acknowledgements are provided as further reference and information points of interest.

* Andolfato, Luigi. “The Generic State Machine Engine Software Design Specification”. European Southern Observatory.
* Andolfato, Luigi et al. “A Platform Independent Framework for Statecharts Code Generation”. European Southern Observatory.
* Chiozzi, G. et al. “A UML Profile for Code Generation of Component Based Distributed Systems”. European Southern Observatory.
* [Development References](#DevAPI).
* “GSMECG: User Manual for Model to Text Transformation”. European Southern Observatory.
* Mehlitz, Peter. “Trust Your Model – Verifying Aerospace System Models with Java Pathfinder”. Page 5. NASA Jet Propulsion Lab.

APPENDIX B: Acronyms

|  |  |
| --- | --- |
| ALF | Action Language for Foundational UMF |
| ALH | Action Language Helper |
| API | Application Programming Interface |
| EMF | Eclipse Modeling Framework |
| EXT | Xtend file |
| JPF | Java Pathfinder |
| SysML | System Modeling Language |
| UML | Unified Modeling Language |
| XMI | XML Metadata Interchange file |
| XML | Extensible Markup Language file |
| XPT | Xpand Template file |
|  |  |

APPENDIX C: Definitions

|  |  |
| --- | --- |
| Ant | Java-based build tool included with Eclipse that uses XML files to compile and run programs |
| Application Programming Interface (API) | Listing of classes and methods that can be used for a programming language |
| Argument | An input of a programming function or method |
| Boolean | True or False |
| Cast | Converting an object from one Type to another |
| Cameo | Model simulation software packaged with MagicDraw |
| Class | Java definition of an object |
| Double | A number with decimals |
| Eclipse | Programming software, a.k.a. Integrated Development Environment (IDE) |
| Global | Available to all subclasses of the variable’s parent |
| Guard | Constraint on a signal |
| Instantiate | Create an object from its Class |
| Integer | A number without decimals |
| JavaScript | Lightweight script version of Java |
| Java Pathfinder (JPF) | Java model checking software |
| MagicDraw | SysML modeling software |
| Method | Java term for a function |
| Signal Event | Transition between states |
| System Modeling Language (SysML) | Modeling language based on UML |
| UML | Programming language |
| Variable | A reference to some value(s) |
| Xpand | Programming language |
| Xtend | Programming language |