SMAP Python Statechart Code Generator Tool User's Guide (UG) / Software Operator's Manual

Release Version 1, March 30, 2010

Prepared by: **Leonard J. Reder**

Document Custodian: **Leonard J. Reder**

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Jet Propulsion Laboratory
Pasadena, California

Revision	Date	Description	Author
Draft for Review	18 Nov. 2009	Initial version	L. Reder/S. Cheng
Draft for Release	30 Dec. 2009	Alex Murray review integrated.	L. Reder
Reviewed for typos	5 Jan. 2010	Tom Lockhart review	L. Reder
Windows support	27 Jan 2010	Added paragraph about support of Windows	L. Reder
Relocation of unit test and support python for tests	30 March 2010	Moved the unit test execution scripts into the QF_Py/bin	L. Reder

Table of Contents

1.0 INTRODUCTION

The new second generation JPL Statechart Autocoder has been considerably reengineered and now has the capability to generate Python implementations of UML Statecharts. The UML specifications does not use the term "Statechart" but rather uses the term "State Machine". In this document we use the two terms interchangeably. Often when we say Statechart we are indeed referring here to a state machine implementation. The Python code generated is based on the open source Quantum Framework originally written in C and C++ and used previously for projects such as SIM, AMD, MSL and Electra Radio. This document explains the setup and basic operation of the JPL Statechart Autocoder and presents detailed information on using the Python Quantum Framework implementation. The operation of the Autocoder to generate Python state-machine codes from UML Statechart models is explained. The software described in this document runs on Mac OS X Leopard, Red Hat Linux and Windows (using Cygwin).

1.1 Identification

This document describes the binary distribution of the JPL Statechart Autocoder bundled with a Python implementation of the C/C++ Quantum Framework. Quantum Framework is a public domain software framework for implementation of multi-threaded hierarchical state machines in software. This software is packaged for the user as a single tar file that will be named Autocoders_<day><month><year>.tar file. This file contains that following directory contents:

1. autocoder -> Contains the autocoder.jar file. This is the JPL Statechart Autocoder executable.
2. autocoder/lib -> Contains the log4j library jar used for logging and the velocity template engine library jar used to generate code products.
3. QF_Py/bin -> Contains a Python start up script called sim_state_start.py to run automatically generated sets of state-machines.
4. QF_Py/examples -> Contains directories of Statechart model examples.
5. QF_Py/test -> Contains the unit tests for autocoder.jar when producing Python.
6. QF_Py/lib -> Contains support Python packages. Currently Pmw (a megawidget toolkit built on Tkinter) for GUIs is included. This package is not currently used, however, a GUI for sim_state_start.py is in development that will use this.
7. QF_Py/src -> Contains the Python package called qf, an implementation of the C Quantum Framework in Python.

Note the JPL Statechart Autocoder is capable of generating many languages and various design patterns. At the time of this writing a C++ and Promela generator are in development. The included JPL Statechart Autocoder can produce C implementations of Statecharts but the C Quantum Framework is needed to compile, link and execute them.

1.2 Overview

The JPL Satechart Autocoder is a lightweight tool utilized by 1) flight software engineers to convert UML Statecharts (stored as XML files) to executable state-machine code, and 2) systems engineers for simulation of their state machines. This effort was funded by SQI, HRDF (ECH funding) and SMAP and Ares1. The immediate goal is to implement Python code generation from Statecharts for simulation of

SMAP fault protection system behaviors using Statechart models. These same models will, in some cases, later be used to generate C++ flight software code.

The JPL Statechart Autocoder supports the following features:

- Simple States (or Leaf States)
- Composite States
- Orthogonal Regions in states
- Initial Pseudo-State
- Deep History
- Junction and Choice pseudo-states
- Transition
- Transition to self
- Guards (using implementation functions that return true/false rather than expressions)
- Actions (Entry, Exit, Transition and signalEvent using implementation functions rather than code snippets)
- Sub-machines
- Threaded do activity actions
- Final Pseudo-state

The JPL Statechart Autocoder does not yet support:

- Terminate
- Entry point
- Exit point
- Connection Point Reference
- Shallow History
- Fork/Join
- Guards using OCL logic expressions
- Actions using code snippets

Each UML Statechart is an abstraction of an implementation that is responsive to events. In UML there are various types of events but the tool currently only supports two types of events, these are: signalEvent and timerEvent. Every transition within the Statechart can contain an optional Guard. Guards are Boolean expressions and in UML there are again several types. For the JPL Statechart Autocoder use only Constraint type and the Boolean expression must be a function that returns a True or False (more on this later). In the UML Statechart notation there are also actions associated with state entry/exit, internal events and transition trigger events. UML provides various types of activities but the tool only supports the Activity type.

Section 2.0 of this document will explain how to install the software and transform a Statechart UML model into Python. A simple Statechart example will be presented and it will be explained how to execute it. Section 3.0 will cover the Python code implementation usage and tell you how to add manually coded functionality to the auto-generated Python and the framework.

1.3 Document Scope

This document describes how to setup and use the JPL Statechart Autocoder and Python implementation of the Quantum Framework publish/subscribe framework to generate state-machine Python simulation codes based on UML Statechart models. The document is intended for software and system engineers who are interested to use the capability for modeling various sorts of systems to build behaviors.

This document assumes that the reader has at least introductory knowledge of (1.) UML (or SysML) Statechart notation, (2.) use of the MagicDraw tool, (3.) Python language. Knowledge of the Python language is not needed until reading section 3.0.

1.4 References

1. Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems, Miro Samek, Newnes, 2009. http://www.state-machine.com/.

2. Learning UML 2.0, Russ Miles & Kim Hamilton, O'Reilly Inc., 2006.

3. Learning Python 2nd., Mark Lutz & David Ascher, O'Reilly Inc. 2004.

4. Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns, Bruce Powel Douglas, Addison Wesley, 1999.

5. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2, http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF.

6. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2 , http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF.

1.5 Document Maintenance

As new versions of this software are released this document will be updated to reflect feature additions and changes.

2.0 OPERATOR'S MANUAL

2.1 Setup Process

The Autocoder and the Python infrastructure can be executed on Mac OS X Leopard, Red Hat Linux and Windows XP or Windows XP via Cygwin. To execute the Python programming language you will need to have either Python 2.5 or Python 2.6 installed on your machine. Note we currently do not use Python 3.x and have not tested with it. Most systems will have a preinstalled version of Python already but if needed a Python distribution for install can be obtained from http://www.python.org.

New in this release is support for running the Autocoder and generated Python state machines on Windows XP native. On Windows XP the unit tests however will not run since the Expect language used within the unit tests does not work on Windows XP. WE RECOMMEND NOT USING CYGWIN ON WINDOWS SINCE THREADING PROBLEMS WITH THE PYTHON HAVE BEEN REPORTED. Having said this if you follow the procedure in section 2.2 all unit tests and GUIs should function.

2.2 Windows XP Cygwin Setup Process

On Windows XP, Cygwin is needed and provides a Unix-like command shell environment. If you are on a Linux or Mac OS machine you may wish to skip to the next section. Also an alternative on Windows XP to using Cygwin is to use the Microsoft Virtual PC 2007 program available from http://www.microsoft.com/windows/virtual-pc/ and then install your favorite flavor of Linux in the Virtual file area.

The Cygwin package can be obtained from the http://cygwin.com/ site. The following procedure will guide you through installation and a few addition steps to make our applications execute properly:

1. Installed cygwin by first clicking the "setup" link on http://cygwin.com/. This should download the setup.exe program and installed the generic cygwin. Than use setup.exe to install the cygwin python 2.5 binary.

2. Next in order for the cygwin shell to find our applications we have discovered that you need to first click on Start:Run... and execute

C:\cygwin\bin\ash.exe

the ash shell. Use the directory where your cygwin is installed. An ash shell window will appear. Make sure no other shells are running and do not try and start ash.exe from a dos shell prompt. From within the ash shell execute the following commands

cd /bin
./rebaseall

Next kill the ash shell.

3. Start a cygwin shell and set PYTHONPATH to

PYTHONPATH=<root stuff>/QF-Py2.0/src:<root stuff>/QF-Py2.0/bin:<root stuff>/QF-Py2.0/lib

This line will need to be added to the .profile configuration file that the cygwin's bash shell uses for environmental configuration. With this step complete your cygwin environment should be ready to use with the Python Statechart Autocoder.

2.3 Python Statechart Autocoder Setup

The autocoder.jar can read MagicDraw 12.5, 16.0 and 16.5 XMI format model files. It can also read MagicDraw mdzip format. To create model files you will need a copy of MagicDraw from the JPL SSCAE site (https://sscae-help.jpl.nasa.gov). Follow the directions on the SSCAE site to install MagicDraw if you do not already have it on your machine. Currently we recommend using MagicDraw 16.5 Standard Edition.

Now that you have Python and MagicDraw installed on your machine you should have a tar file named as QF_Py_20091230 or similar tar file with a name of the form QF_Py_<year><month><day> containing the software. Unpack this file into a convent location on your computer. This is done by executing

```
% tar xvf QF_Py_20091230.tar
QF_Py/bin/sim_state_start.py
….. more lines of output
QF_Py/test/autocoder/validate.pyc
QF_Py/test/build.xml
autocoder/autocoder.jar
```

Once unpacked, the autocoder.jar capability can be tested to validate various Statechart features are being generated as Python and executing correctly. A unit test set is stored in the QF_Py/test directory. Before you can run unit tests set your PYTHONPATH environmental variable. This variable tells Python where to find packages such as the 'qf' package. The 'qf' package implements core functionality that provides state-machine to state-machine event communications, hierarchical execution, etc. If you are on Windows and installed cygwin you have done this in the previous section. On a MAC or Linux start a terminal shell. If you are running the standard bash shell you can set the PYTHONPATH by adding this

```
PYTHONPATH=${PYTHONPATH}:<root directory>/QF_Py/src
PYTHONPATH=$(PYTHONPATH>:<root directory>/QF_py/bin
PYTHONPATH=$(PYTHONPATH>:<root directory>/QF_py/lib
```

To your .profile file. If this fails since you do not have a PYTHONPATH already defined then add this

```
PYTHONPATH=<root directory>/QF_Py/src:<root directory>/QF_py/bin:<root
directory>/QF_py/lib
```

to your .profile file. If you are running tcsh or csh set your PYTHONPATH by executing

```
setenv PYTHONPATH ${PYTHONPATH}:<root directory>/QF_Py/src:<root
directory>/QF_Py/bin:<root directory>/QF_Py/lib
```

```
if this fails since you do not have a PYTHONPATH already defined then
execute this
```

```
setenv PYTHONPATH <root directory>/QF_Py/src
setenv PYTHONPATH ${PYTHONPATH}:<root directory>/QF_Py/bin:<root
directory>/QF_Py/lib
```

These lines can be added to your .cshrc file for tcsh or csh users.

Next to run the unit test cases execute:

```
% cd QF_Py/bin/
% ./pythonsuite.py
```

This will generate a series of test case runs where the autocode.jar was run on various models. The generated Python output code from the autocode.jar is than executed and validated for each test. After the tests have completed a display similar to the one shown below will be displayed

```
Final Test Results:

Simple State-machine:   PASSED
Simple2, Internal Transition:   PASSED
Simple3, Self-transition:   PASSED
Simple4, Transition Effect:   PASSED
Simple5, Transition Guards:   PASSED
Composite state with Simple Transitions:   PASSED
Composite2, Deep History:   PASSED
Composite3, 2 Orthogonal Regions:   PASSED
Composite4, Cross-hierarchy Transitions:   PASSED
Composite5, Timer Events:   PASSED
Composite6-1, Top-level Orthogonal, Cross-Dispatch, Hidden Region :
PASSED
Composite6-2, Inner Orthogonal Regions:   PASSED
Composite6-3, Inner Orthogonal Regions & Unnamed+Hidden State:   PASSED
Composite7-1, Init Action, Internal Transition in Super/Orthogonal/Leaf
States:   PASSED
Composite8-1, 3 Orthogonal Rs, Inner Composite, Multi-level Trans, Action
List:   PASSED
Composite8-2, wrapped in a super-state, 3 Orthogonal Rs, Inner Composite,
Multi-level Trans, Action List:   PASSED
```

If any of these tests fail it should be reported to the contacts given at the end of this document. However, one, two or more tests failing does not necessarily mean that the software is unusable. Completing the execution of these python tests means that the basic code generation of Python is working. Next we will walk you through a simple Statechart example to illustrate how to execute the Autocoder and the Python generated state machine codes.

2.4 Instructions for Operations

To use the QF-Py software one needs to first create either an XMI or mdzip model file using MagicDraw. The state machine elements created in MagicDraw can be located at any point within the model data tree. Execute the autocoder.jar program by executing the command

%java -jar <path>/autocoder.jar –python <file.xml>

where the <path> is the directory path to the autocoder.jar file and <file.xml> is a valid MagicDraw XMI model file. For a complete list of commands execute

%java -jar <path>/autocoder.jar -h

Executing the autocoder.jar will generate three Python files for each Statechart that was found in the model file. The files produced are:

1. <state-machine-name>.py is an implementation of the MagicDraw StateChart drawing that is displayed as a Python window and used to follow the active state as the generated state-machine is executing.
2. <state-machine-name>Active.py is the implementation of the actual StateChart behavior represented as an executable Python module.
3. <state-machine-name>Impl.py is the implementation a class, generated only if one does not exist. This file will be discussed more in section 3 but it is used to implement code for action and guard functions manually.

where <state-machine-name> is the name of the Statechart diagram found in the model. All the generated files must be stored in the same common directory to use the QF-Py/bin/sim_state_start.py program to launch them. The state machines utilize and depend on the Python Quantum Framework package qf for communications and timer event implementation. The qf package also starts the trace widget GUI panels that display active state.

To start running a set of generated state machine Python codes execute the sim_state_start.py program as follows

% cd <root>/QF_Py/bin
% ./sim_state_start.py -p <target path to generated state machine codes>

This will search the target path for generated state machines. It will also execute a small program called gui.py to start the Python trace GUIs showing Statecharts. For example to start up the a simple TrafficLight controller example set of two auto-generated state machines execute the following from the bin directory

% ./sim_state_start.py –p ../examples/TrafficLight/autocode/

You will see three windows displayed. Two Statechart GUI trace windows called NSTrafficLight and EWTrafficLight and a row of buttons called Statechart Signals. Clicking the buttons sends signal events to the Statecharts and will cause state transitions. Also in the shell you will now see the prompt

>>>

This is an interactive Python prompt. Any of the signal events can also be generated by using the SendEvent("signal event") Python function call at the prompt. You can generate a timer tick by using the tick() function at the Python prompt and to exit use quit() function at the prompt.

For a listing of sim_state_start.py command options execute

% ./sim_state_start.py -h

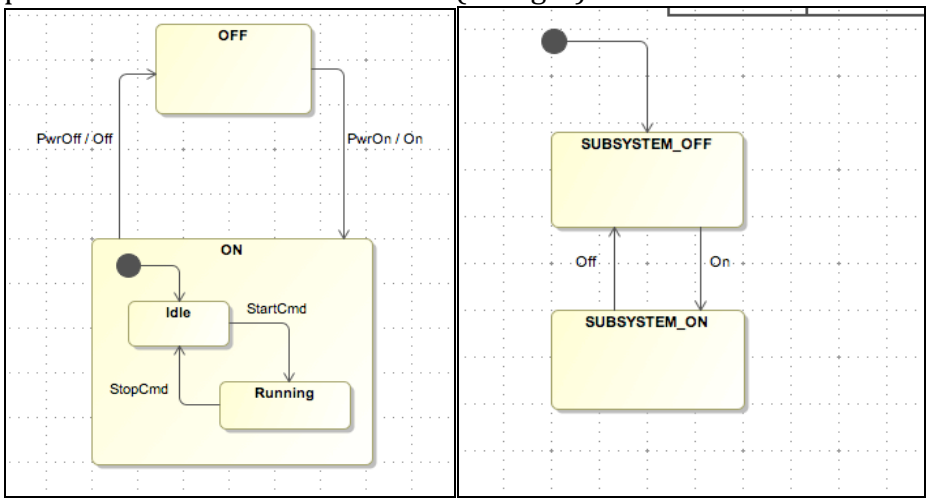
Note that you can use the –L option to create a log file of all state activities that are printed on the console as events are executed.

This section has described the use of the two main end user applications (autocoder.jar and sim_state_start.py). To make the process clear we will next walk through a simple Statechart model: creation, code generation and execution example. This will be followed in section 3.0 by detailed reference instructions for programming using the generated implementation (*Impl.py) files to hand code Statechart actions and guard functions.

2.5 Simple Example

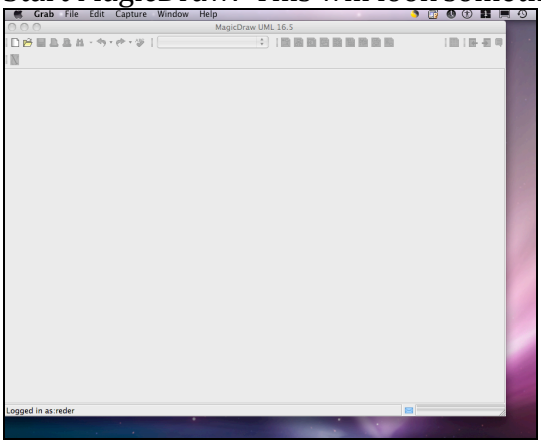
This section presents a simple example model and will take you step by step through the creation of a Statechart using MagicDraw 16.5. We will then use autocoder.jar and sim_state_start.py to generate and execute the model. This section is written to give the user a quick start if you have never used MagicDraw or code generation. If you are experienced you may want to skip this section and read section 2.4 that covers the details of configuring actions and guards within MagicDraw Statecharts. Also presented is an introduction to using the generated Python state machine implementation interface class.

The Simple model will guide you through creating the Statecharts shown in the following figures. The PwrOff and PwrOn triggers (e.g. signal Events in left Statechart) cause Off and On events to be published to a second Statechart (on right). The Off and On events cause the state to transition.

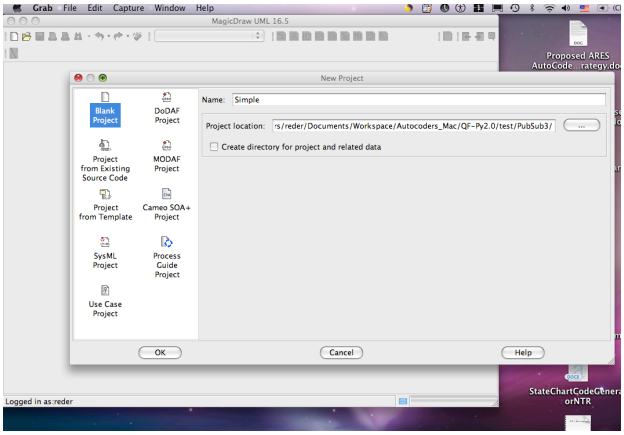


The steps to produce and run the model are:

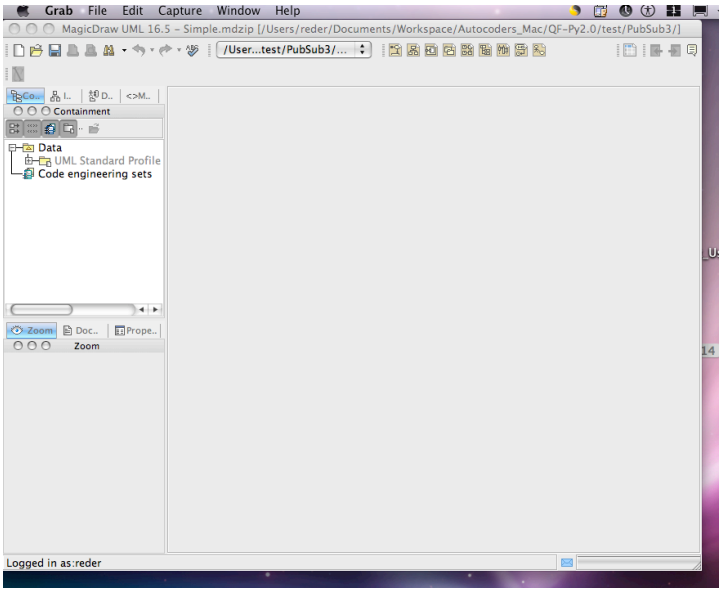
1. Start MagicDraw. This will look something like the figure below when it first starts.



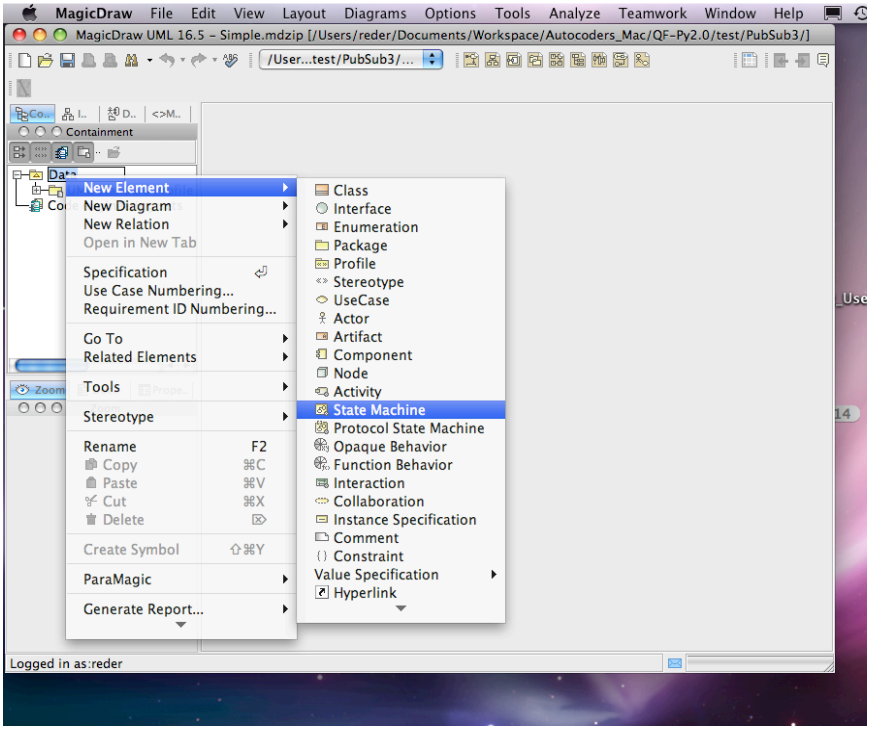
2. Next click on File:New... menu and the following will appear.



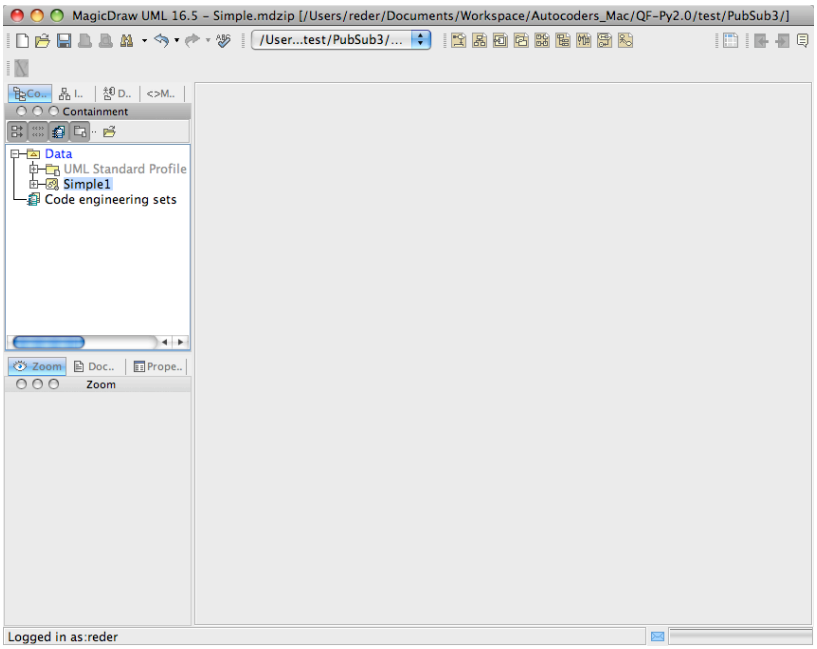
Select either the "Blank Project" or "SysML Project" choices; since UML and SysML Statecharts are identical it will not make any difference. Click the OK button and the basic model tree will be created as shown in the next figure. A tree of state machine, state and transition elements will be created under the Data root node of this tree. The diagram of the Statechart you will create in the next step will also be represented.



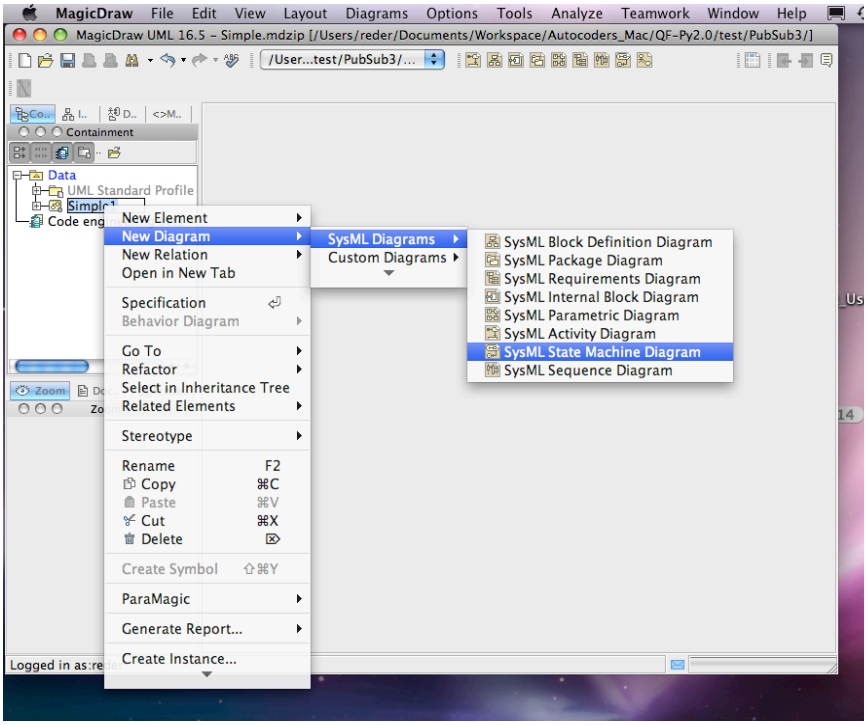
3. To create a state machine model select the Data root node and right-click to show the action menu. Select "New Element:State Machine" to create a new state machine model. See the next figure.



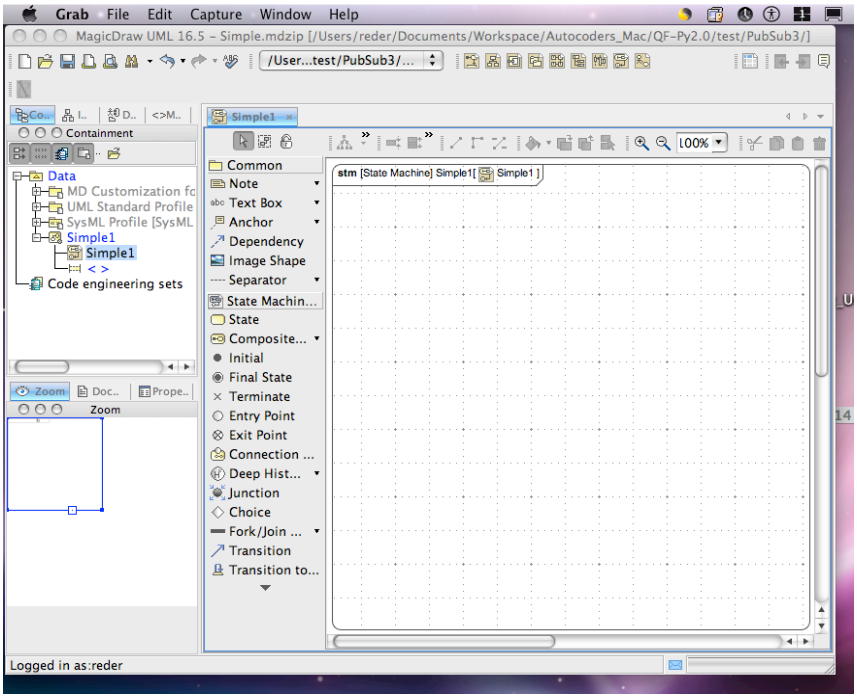
Next name your state machine model "Simple1" by typing into the highlight untitled space in the Data tree. See the next figure.



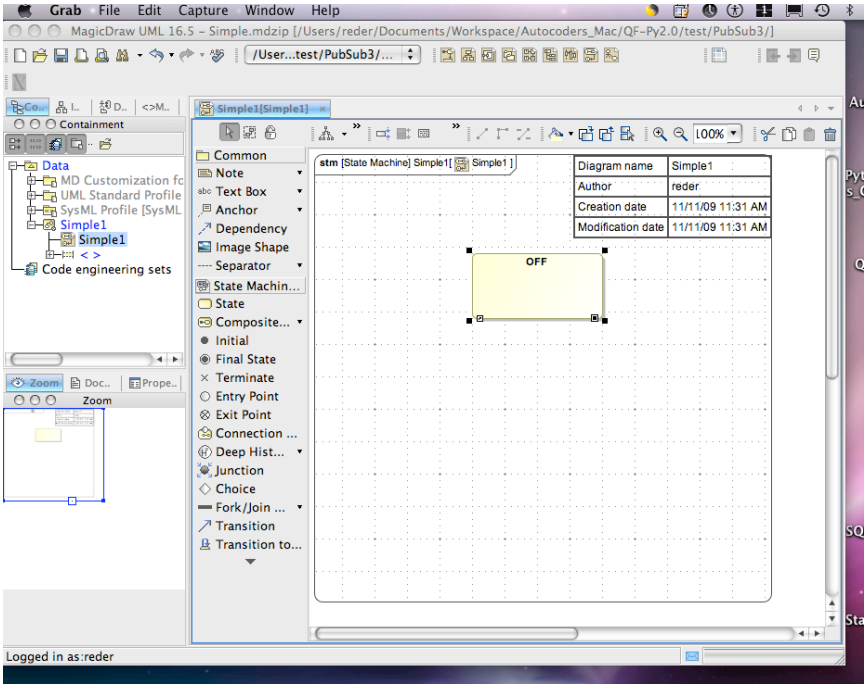
4. Now that the Simple1 state machine model is created we want to create a Simple1 diagram so that all the model elements can be graphically entered into MagicDraw. To create a diagram right-click on the Simple1 model and select "New Diagram:SysML Diagrams:SysML State Machine Diagram". If you do not have the SysML plug-in installed your menu might look a little different but there will still be a State Machine Diagram option.



After generating the diagram the MagicDraw user interface should look like the following figure. Notice the area to the left of the diagram contains all the elements of a Statechart. You will drag these into the diagram to create the states and transitions of your model.



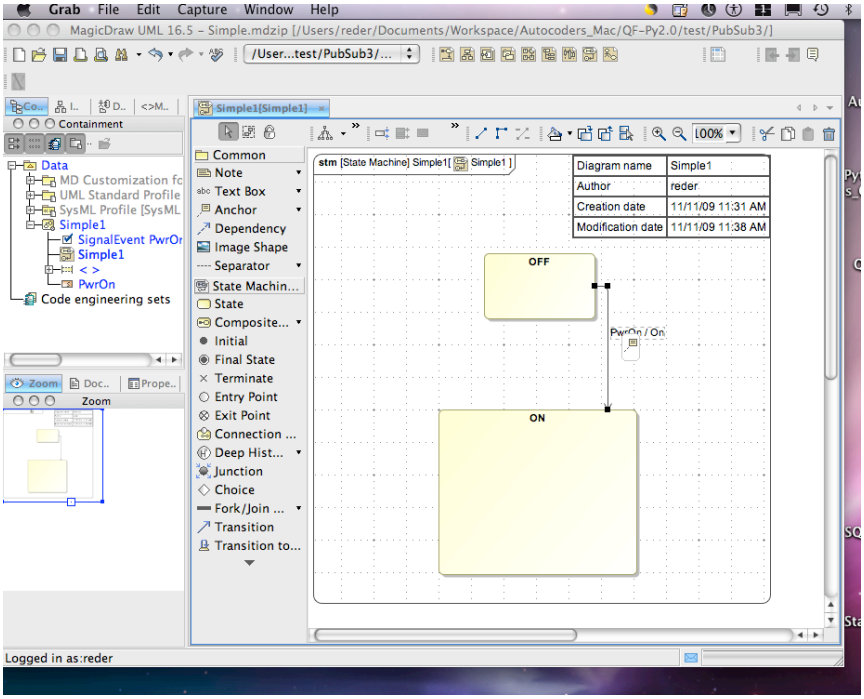
5. Create the first state in the Simple1 model by dragging the State icon from the left of the figure onto the white canvas area. Once you release the mouse the State is selected and you can type "OFF" into it and that will be the state name.



Your diagram should look like the above.

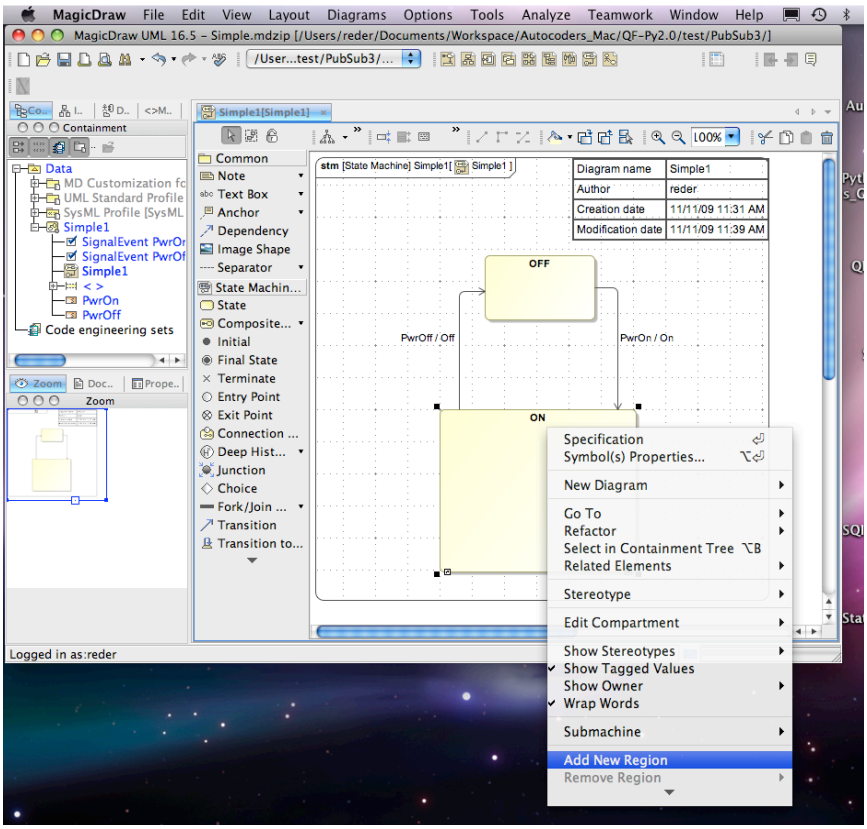
6. Drag another state onto the diagram and type "ON" into it. You can use the four tabs on the selected state to resize it to be similar to the following diagram. Next create a transition by dragging the transition from the left (line with arrow symbol) on to a state. The state will turn

blue and be the source state of the transition. Release the mouse and drag the transition to the target state. In the figure below the source state is the OFF state and the target state is the ON state. While the transition is selected use the tabs on it to position it similar to the figure. Finally type "PwrOn/On". This will assign a signal event of "PwrOn" to it with an action to generate (or publish) an "On" event.

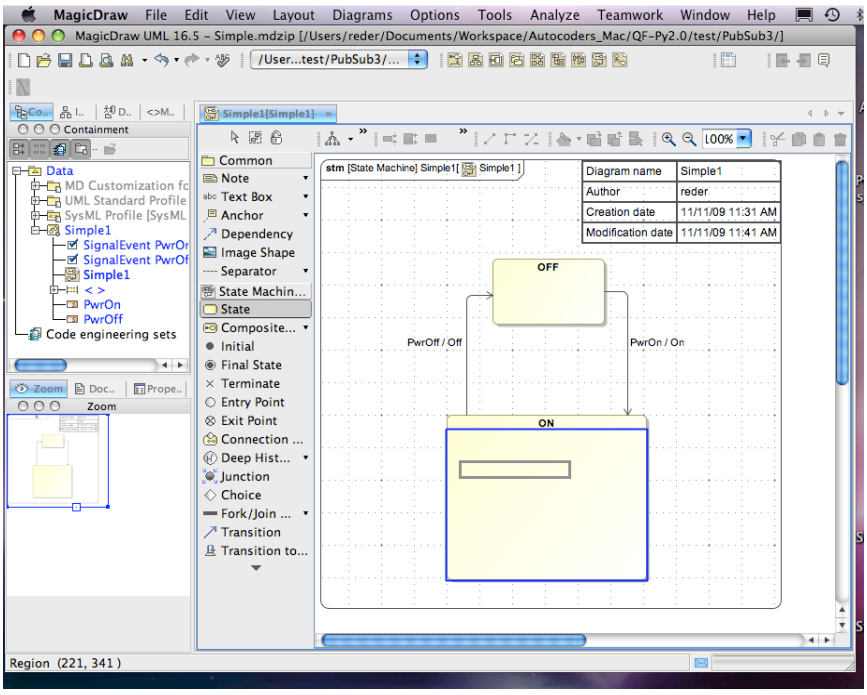


Create another transition but this time from the "OFF" state to the "ON" state. Position it and type "PwrOff/Off". The diagram should look like the one shown in the next figure.

7. Recall that the "ON" state needs to be a composite state that will contain states "Idle" and "Running". To do this, right-click on the "ON" state to show the action menu. Select "Add New Region" which adds a region to the state that will contain states and transitions dragged on to it. See the next figure.



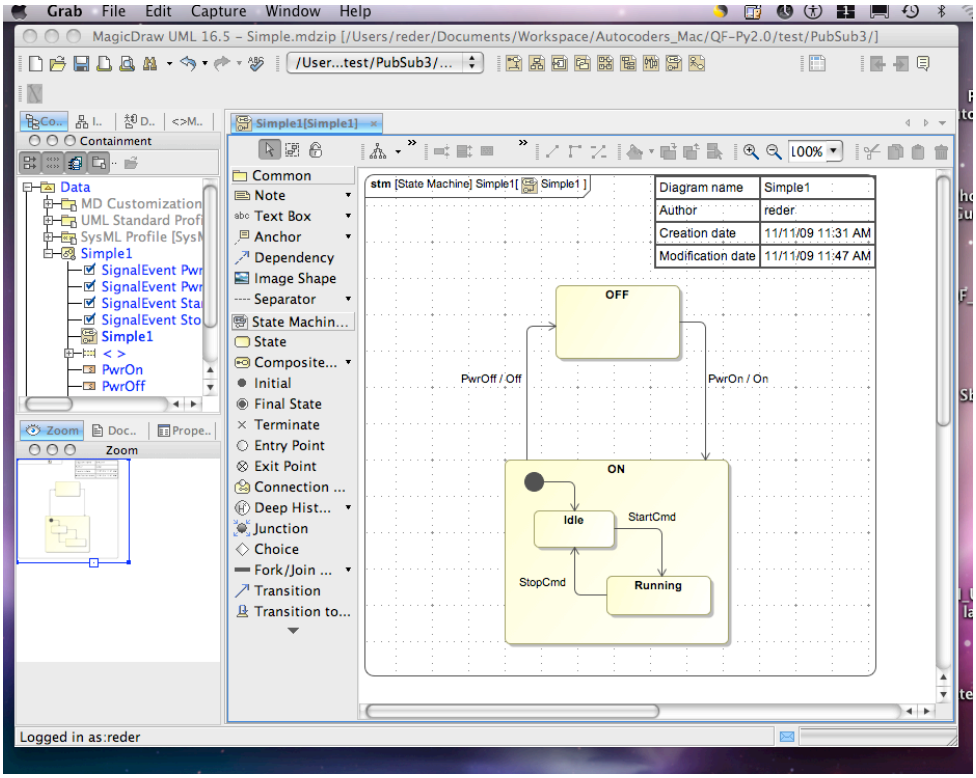
8. The “ON” state with the new region will not appear to have changed but when a state is dragged to it, it will be highlighted in blue. This highlighting indicates the “ON” state is a type of composite state. Drag a state from the left panel onto the diagram and into the ON state as shown in the following figure.



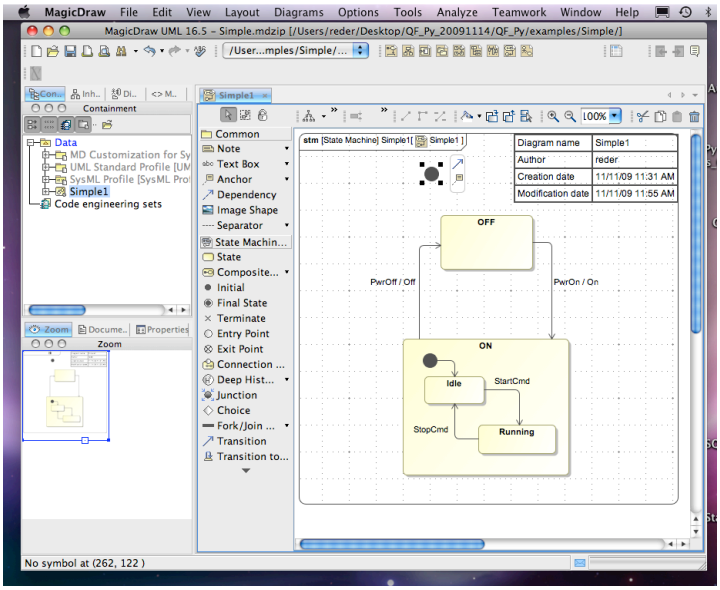
Label the new state within “ON” with the name “Idle”. Again by typing into it while the state is

still selected.

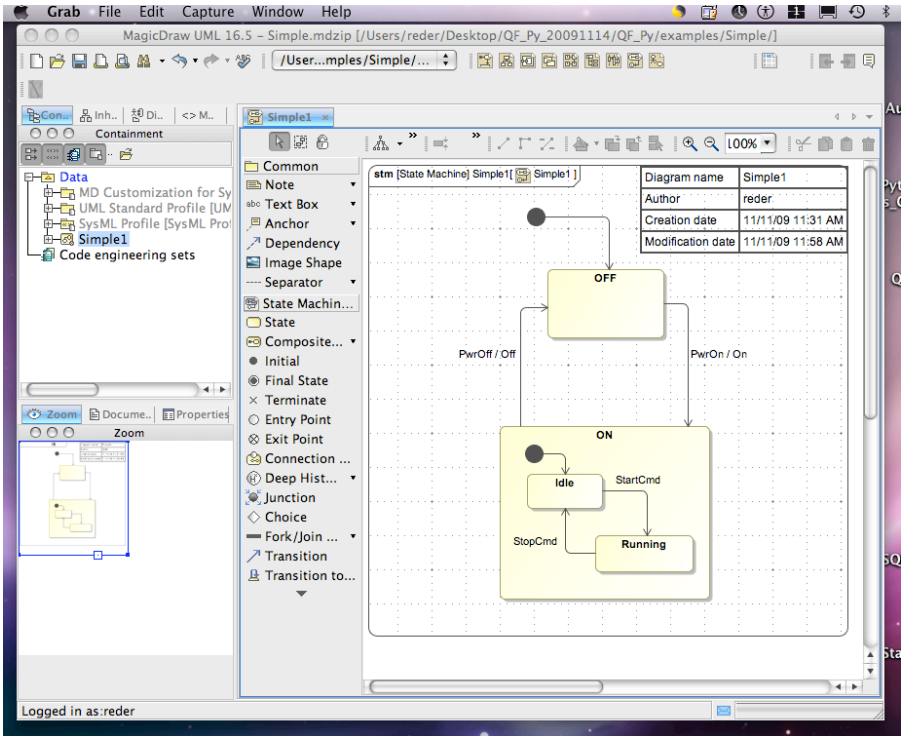
9. Repeat the above procedure to add the “Running” state and transitions inside the “ON” state as shown in the next figure.



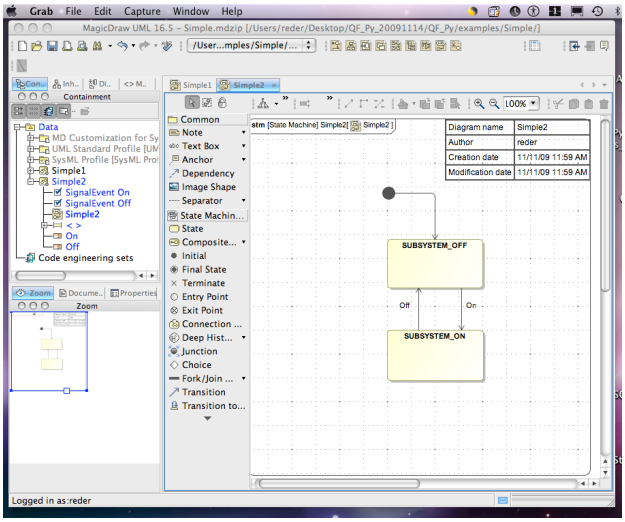
10. The above figure looks like it is almost the completed Simple1 diagram shown at the top of page 7 but it is missing the initial pseudo-state. All Statecharts must



contain at least one initial pseudo-state at the root level of the diagrams hierarchy. The initial pseudo-state determines how to initialize the Statechart. All composite states and each orthogonal region of a state must also contain one and only one initial pseudo-state. At this point to complete the Simple1 Statechart drag the initial pseudo-state into the diagram and create a transition from it to the "OFF" state. You should now have something that looks like the following figure.



11. Next create the Simple2 state machine that contains a Simple2 diagram and the two states and transitions shown in the next figure.



Simple2 will receive events from Simple1 when you execute the model so make sure the case of

the “On” and “Off” events match the events published in the Simple1 Statechart.

12. Finally save your model as an XML file using the MagicDraw File:Save As… menu item. In the dialog that appears select “Extensible Markup Language (*.xml)”. The autocoder.jar can also read the MagicDraw mdzip format files if desired. We have already saved this example model in <root>/QFPy2.0/examples/SimpleRef/Simple.xml
and will guide you through generation of Python and execution of the model. In this same location an example Makefile exists for executing the autocoder.jar but we will do this manually for learning purposes.

    To generate the Python code from the model execute the following:

```
[reder-2:QF-Py2.0/examples/SimpleRef] reder% mkdir autocode
[reder-2:QF-Py2.0/examples/SimpleRef] reder% cd autocode
[reder-2:examples/SimpleRef/autocode] reder% java –jar \
../../../../autocoder/autocoder.jar -python ../Simple1.xml
```

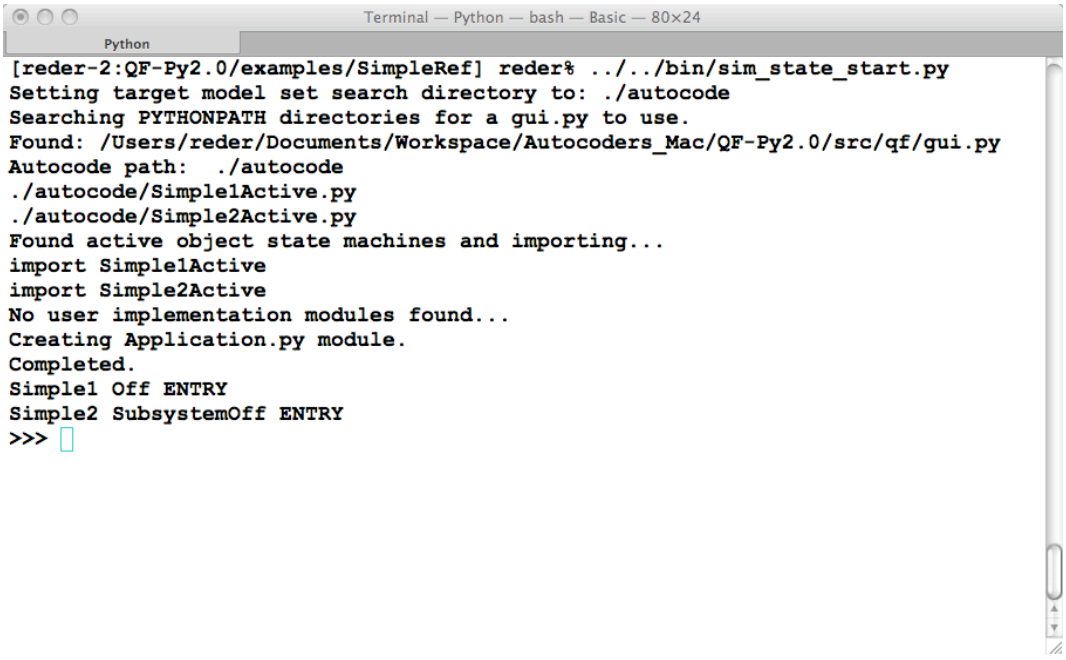
    where the output from the code generation is

```
Autocoder 2.1 beta (build 20091028-0900)
/ CWD /Users/reder/Documents/Workspace/Autocoders_Mac/QF-
Py2.0/examples/SimpleRef/autocode
\ Found generator kind(s) [C, Signals, Python, Promela]
Autocoding to target Python
Loading XMI doc from ../Simple1.xml...
Model contains 2 State Machines, 7 SignalEvents, 0 TimeEvents, and 7
Signals.
Processing State Machine: Simple1
Processing State Machine: Simple2
Collecting Diagram data...
Writing Python target Simple1Active.py...
Writing Python target Simple2Active.py...
Writing Execution Trace GUI target Simple1.py...
Writing Execution Trace GUI target Simple2.py...
Writing Signals target StatechartSignals.h...
Writing GUI Signals target StatechartSignals.py...
Autocoding Finished (1186ms).
```

    Notice that two *Active.py files are generated. These are the implementations of the state machines in Python and the other two *.py files are trace gui widget that will display the run-time current (or active) state that the state machine is in as the result of transitions triggered by events.

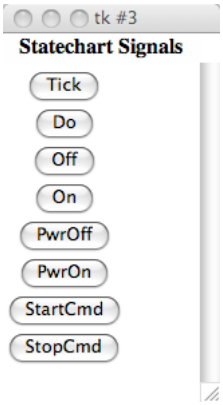
    To execute the generated code the utility `sim_state_start.py` is used. We first change directories out of the `autocode` directory. The `sim_state_start.py` by default looks in `./autocode` for executable model code - this can be overridden using the –p command line option to the tool. The execution of `sim_state_start.py` is shown in the next figure. This will start all the state machines found in the `autocode` directory. Note that

`sim_state_start.py` can also start only certain state machines by specifying there Statechart names as arguments.



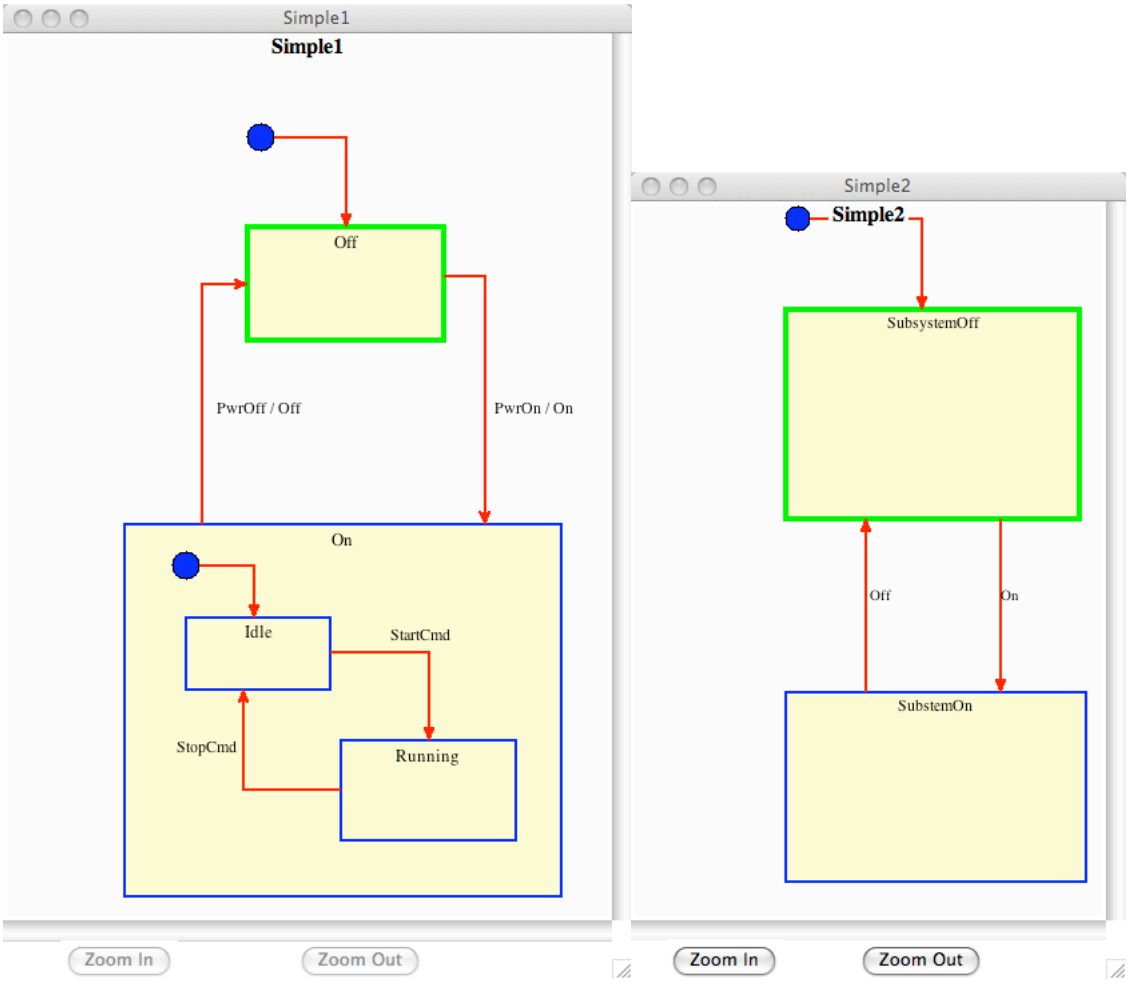
In the above screen snapshot the start up console is displayed. Note that the indication that Simple1 is in the Off ENTRY condition meaning it is in the Off state and Simple2 is in the SubsystemOff state. The >>> after the messages is an interactive Python prompt. You can import your own test routines or use it to execute function calls to generate events or simulate clock ticks as will be explained more in section 3.0.

Three windows will appear, two are trace widgets representing your Statecharts and a common signal (e.g. signal event) button widget is generated. The Statechart Signals widget is shown in the next figure. If one clicks PwrOn a PwrOn signal event is published causing Simple1 to transition.



It is also possible to generate this exact same event at the Python prompt by typing SendEvent('PwrOn').

The two Statechart trace widgets are shown in the next figure. They have been resized to fit side by side and are show in the initial state. Of course after the SendEvent('PwrOn') they would have both transitioned.



Each trace widget indicates the active state by highlighting the state color as green. There are Zoom In and Zoom Out buttons located at the bottom of the window but they have no effect until you press Zoom In. Also clicking the mouse and dragging over an area and releasing zooms to that area. If you zoom into a single state and than transition to another state the trace gui view area will move to the active state region.

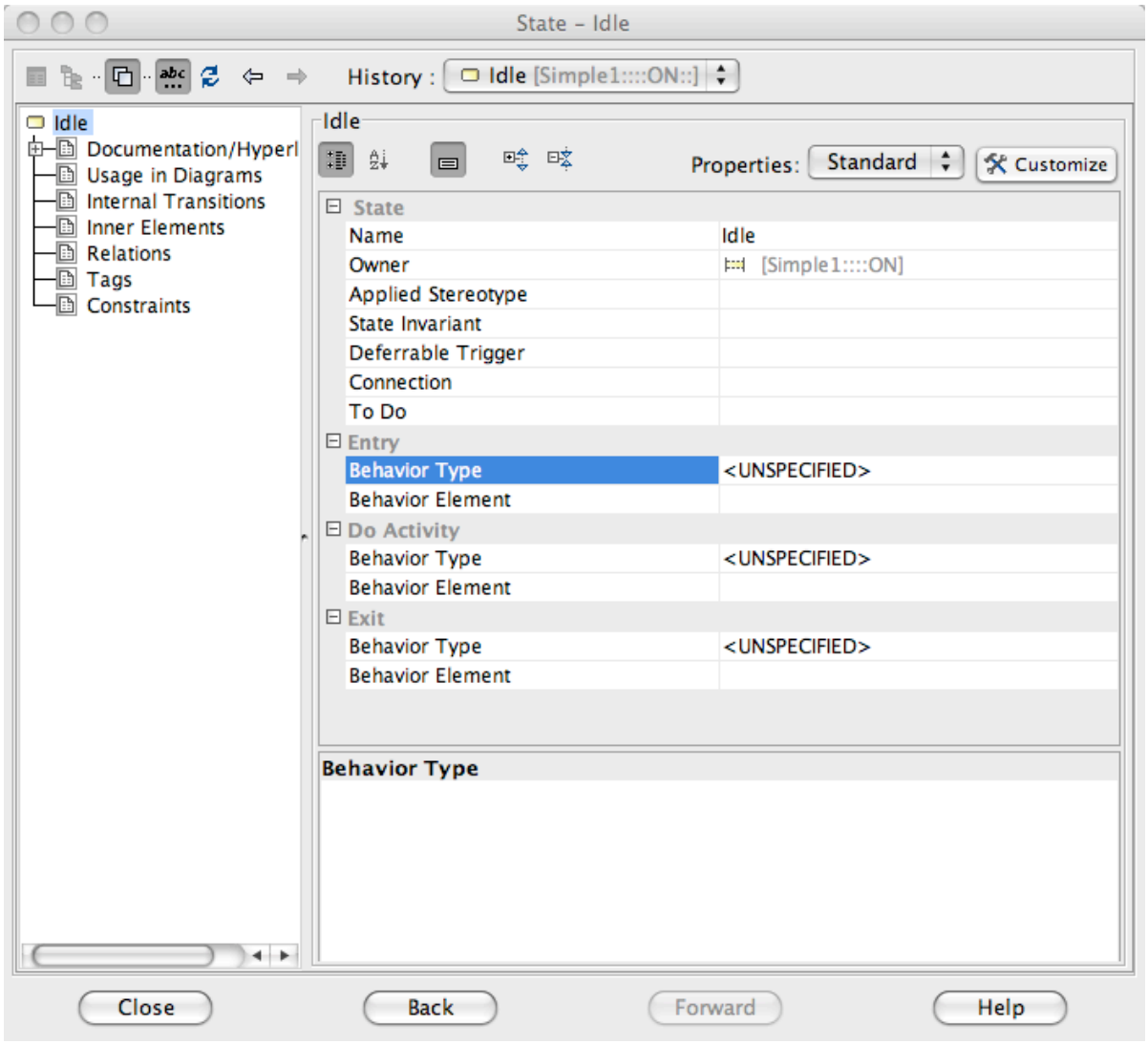
These are the twelve basic steps to getting a simple model entered into MagicDraw and executing it. However we have not discussed implementation of actions or guards within models. The Simple model example presented above includes two actions. These are in Simple1 and are the On and Off events. We refer to these sorts of actions as event actions that publish events to other Statecharts. If these events were expresses as On() and Off() with "()" than the code generator would interpret this to mean generate a function call and associated implementation method stubs within an implementation class. More about the implementation class with be presented in Section 3.0 on the Python detailed API provided but it is important to understand how to make sure your transitions and state specifications are set correctly to generate desired execution code. We will discuss the state and transition specification dialogs found within MagicDraw as they are used to generate Statecharts in the next section.

2.6 State and Transition Specification in MagicDraw

When creating models in the MagicDraw CASE tool all elements have a Specification dialog associated with them that allows setting of various attributes and properties. These Specification dialogs have many entry fields and are often daunting and confusing to the new user. For Statechart creation we only need to be aware of the State and Transition Specification dialogs that are opened when you either double click or press <return> on a selected State or Transition.

Within the Simple1 Statechart diagram above in MagicDraw one can select the Idle state and then hit the enter key. The State Specification dialog as shown in the figure should appear. As you look at the specification dialog, in the upper right corner, make sure to set the Properties to "Standard". This will minimize the number of fields displayed to the ones needed for Statechart State configuration. Below the Name property is set to "Idle"; this is the state name. The Name property can be used to change the name of a state. The Statechart Autocoder supports Entry and Exit activities. Entry and Exit activities are actions that are automatically executed upon entry or exit, respectively, to or from the state. To assign an Entry or Exit first select the type as "Activity" (currently the only supported type). Then enter a name for the Behavior Element – this is the actions name. These action names will appear in the *Impl.py implementation class file as functions that you write by hand.

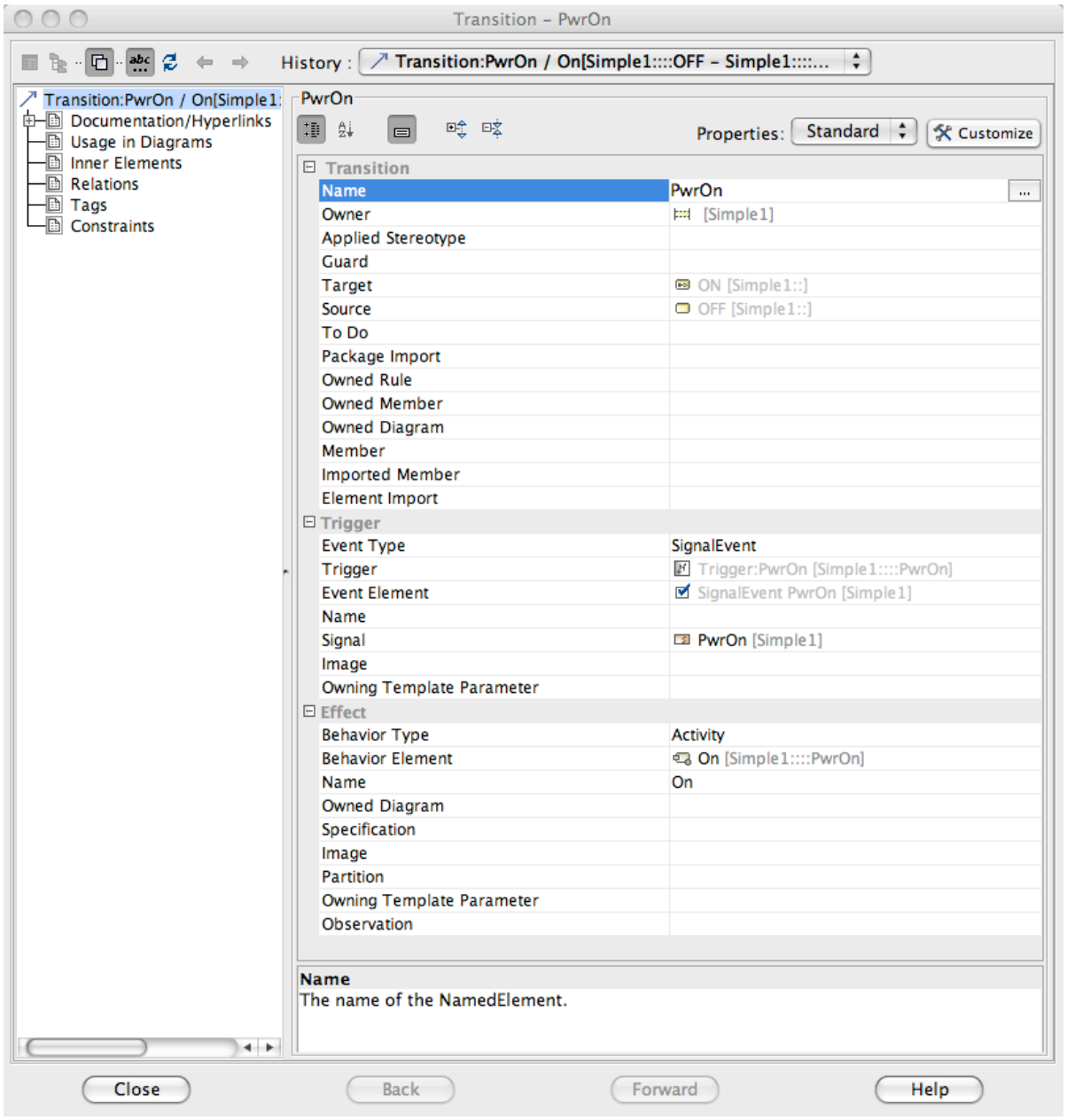
The Do Activity is also supported. Again a Do Activity is generated as a implementation class function that you will implement. However, on entry into the state the Do Activity is executed by spawning a separate thread. The activity runs until it exits itself or until the state is exited. Care should be taken when using Do Activities to have them exit before leaving the state but this is not required since the exit of the state will force a kill of the do activity thread. The problem is that the kill does not care what the user's do activity implementation code is doing and the kill will be abrupt on state exit.



Note that a name given without any "( )" is read by the Autocoder to be an event to be published. A name with suffixed "( )" is interpreted to be an implementation function call that executes an action. If you give a name without the "()" it is assumed that it will be published (or broadcast) to other state machines. At least one state-machine should use the event in some way. If the name has the "()" suffix it is interpreted as an Activity action that is an implementation function that will be called. These activities get manually coded in the generated *Impl.py files. Do not worry about the implementation functions initially since you can add them later in Python. Your generated code will run without them but messages that your function action is not implemented will be printed. This is unlike C++ that requires any actions to be implemented or the code cannot be compiled.

In the above Specification dialog you can use the "Internal Transitions" item in the left box to create internal events with Activity actions. These will be shown inside the state and will cause the Activity in the state to be executed if you currently are in the state when the event is dispatched.

For the Transition elements within a Statechart set the Properties to "Standard" to minimize parameters as we did for State elements. The next figure shows the Transition Specification dialog.



Transitions for the Statecharts are expressed as

Signal Event [GuardFunction()]/TransitionAction()

Where in the Specification dialog the signal event is entered in the Trigger area. Set the type to SignalEvent and on the right side of the signal property entry click the pull down button to display and select a signal element. If the signal you are looking for does not exist you create in the model Data tree like you did above for the state machine. Recall, as above, we use right-click and select New Element and next select Signal and then enter the name of the signal element we wish to add. Now the signal will appear is the Trigger area signal property list and you can select it.

Also in addition to SignalEvent type the Autocoder supports TimeEvent type where you enter a number that represents some number of clock tick delay before the transition is made. The number is

entered into the When property. If you have a selected transition you can enter "at(5)" and this will create a TimeEvent of 5 ticks in the Transition Specification dialog.

NOTE that all transitions, except initial transitions for the JPL Statechart Autocoder are required to have either a SignalEvent or a TimerEvent associated with them. On transitions Guards and Actions are optional.

If guards or actions are used on transitions they must be assigned as function calls or in the case of a transition action a valid signal event name can be used. In the Transition Specification dialog under the Transition area use the Guard property to enter the guard function. Always specify guards as Constraint type and use the GuardFunction() with the () to indicate a function. Note that guard action functions always return True or False and generic actions never return any value.

If a transition action is desired than in the Transition Specification under the Effect area select the Behavior Type as Activity and for the Name property enter a name of a valid event signal to be published or a function call to be executed. Note that published events are automatically subscribed to by the Statecharts that are using them so you do not need to worry about configuring this.

3.0 USER'S GUIDE

We have covered in section 2.0 a simple example to get you working with the Python Statechart capability. In this section some of the details of how the generated state machines work and how the publish/subscribe Python Quantum Framework work will be covered. The Python package called "qf" located in <your root>/QF-Py/src area implements a Python version of the Quantume Framework described in Reference 1 section 1.4. The JPL Statechart Autocoder is generating the same basic state machine design pattern presented in Reference 1.

When you create a set of UML Statecharts each Statechart will be executed as a separate thread. When they are launched using the sim_state_start.py program a main qf thread is running within the Python interpreter. From this main thread the user can execute programs to drive your set of Statecharts. In the section the API for such codes will be explained. For state-machines that contain general actions functions or guard functions an implementation class is generated for you to insert implementation code into. The implementation class is only generated once so your code will not be over written as you change and add new things to your Statecharts. We will explain the implementation class more in the following subsection. Finally we will review top level how the Python is organized and how to use the sim_state_start.py program as a module rather than end user executable for your building your own stand alone codes with this framework package and capability.

3.1 Public Function API

Within the `sim_state_start.py` program there is a small set of helper functions and module global variables that all you to create test programs and interact with the state machines. The module globals are listed in the following table.

	sim_state_start.py module global variables
AUTOCODE	Set to the path that `sim_state_start.py` searches for state machine objects to start up.
ACTIVE	Set to a dictionary of state machine name to active class references mapping.

	For each keyword of state machine name there is a value that is a reference to the generated state machine active object instance defined in *Active.py file.
IMPL	Set to a dictionary of state machine name to implementation class references mapping. For each keyword of state machine name there is a value that is a reference to the implementation class defined in an *Impl.py file.
GUI_PY	This is a pointer to the gui.py file that is used to execute gui.py used to start the state machine gui trace widgets.
qf	Reference pointer to the Python Quantum Framework singlton object. This is explained in the next section.

The module contains a set of useful module level functions for the end user to either call at the Python prompt or from within test and simulation scripts.

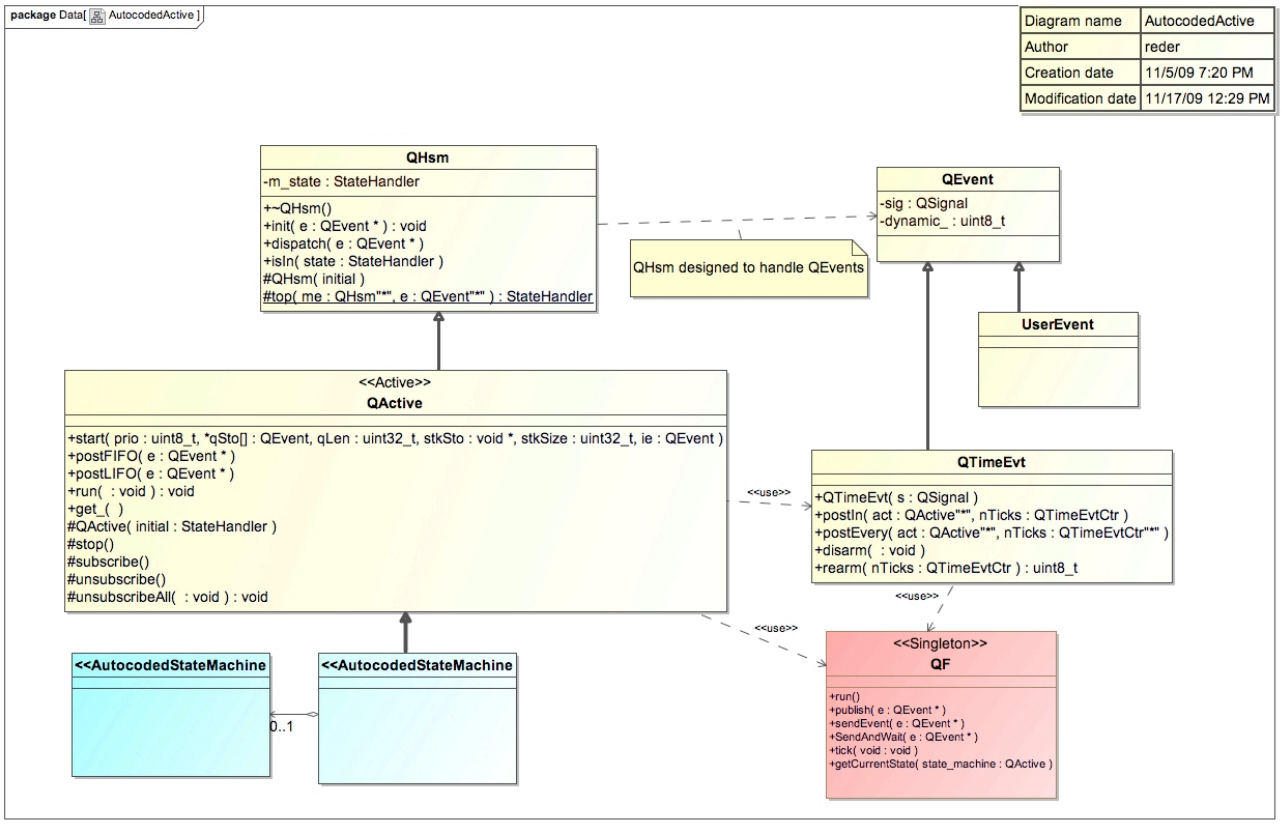
	sim_state_start.py module functions
main(args=[])	Main start up routine. This routine searchs for generated state machine codes to import. It than imports them, initializes and starts threads. It also detects implementation classes being present and loads them. All the arguments are optional and defined as: @param args: a list of names of state machins to be executed or empty for all state machines. @param nogui_flag: enable/disable trace gui (default: disabled gui for testing) @param noimpl_flag: enable/disable implementation class instancing (default: disabled) @param log: record state transition information to log file, give a string logfile name to this argument to enable logging to a file of that name. @param list_flag: print list of generated state-machines found and exit. @param path: specify a target search path (default: ./autocode).
sendEvent(e)	User function to publish an event. This is a wrapper function that makes publishing events friendlier to the the user. The argument e can be several things. The argument e can be specified by a string name of an event in which case an event.Event() object is generated and published containing the signal name. Also e can have the value "tick" that essentially calls the tick() function explained below. And e can be "quit" which is a special event.Quit() event to exit the Python. Finally and this is beyond the scope of this document e can be an actual Python event.Event() object that contains a signal and perhaps additional data.
sendAndWait(e,delay)	This function executes sendEvent(e) and than does a time.sleep(delay) to wait delay seconds before allowing the program to resume.
tick()	Processes all armed time events at every clock tick. This simulates a clock tick that causes each armed timer event object to decrement it's count. When a TimeEvent object's count reaches zero the timer event is fired and the machine transitions.
currentState(name)	Return the current state that a state machine is in. @param name: string name of state-machine @return: returns string name of state
quit()	Terminate all running Python threads and exit from Python. If gui.py is running kill that process.

Using these calls the casual user will be able to automate tests and simulation scenarios in a straight forward manner. The user should be aware that all these calls are executing in the main thread of the Python interpreter and any implemented actions or guards will be executed in the state machine object thread. Because of this separate execution care must be taken when implementing any interactions. To help understand this better we next present a discussion of the Python qf package

that implements the Quantum Framework on reference 1 but in Python.

3.2 Statechart Active Object Model and the qf Package

Each UML Statechart generated state machine code is contained within a class. The autocoded state-machine class inherits a class called Active. The following diagram shows the nominal relationship of the autocoded (blue) state machine classes with the other classes that make up the framework. The figure uses C++ class names but if the Q is removed from the start of the name the result is the associated Python classes.



Starting from the bottom of the above figure there are two blue classes. The one to the far left represents an optional generated <StateChartName>Impl class that is used by the right blue class that is called <StateChartName>Active. <StateChartName>Active is made up of state transition functions that are called from the inherited HSM class dispatch method. The HSM class handles all the hierarchical behavior of the UML Statecharts. <StateChartName>Active class also inherits the Active class that implements the threading and interfaces to the QF singleton object for providing publish/subscribe event communication between state machines and controlling timer events. The Active class contains the state-machines main run() method that gets event objects off a dedicated queue and executes the HSM.dispatch(e) call.

<StateChartName>Active classes each are instanced in their own thread and are typically called active objects. These active objects use the Event object as a message that is published from one active object to many. Optionally a user can inherit the Event class and build custom signal events that carry data in addition to a single signal attribute. This feature has not been exercised at the time of this writing. Finally note that timer events are implemented with the TimeEvt class that inherits Event. Autocoded

Statecharts implement an instance of TimeEvt for each time delayed transition. Basically the TimeEvt class implements a counter that is decremented. The tick() method of the QF singleton maintains a list of armed timers.

All the supporting classes shown in the above diagram are implemented in the package called qf. The following table lists the Python modules in qf and describes each of them.

	Python qf Package Module
active.py	Emulates the Quantum Framework QActive class that provides infrastructure threading functionality for each state machine. This class also provides interface to publish/subscribe and timer operations implemented in the framework.py module.
event.py	Emulates the Quantum Framework Event class that provides infrastructure of abstract event class for the user to implement custom Event classes. We encapsulate Statechart signal in the event.Event class for publish/subscribe.
framework.py	Emulates the Quantum Framework QF class that provides publish/subscribe and timer event management and generation. This class also launches trace gui widget handlers.
framework_gui.py	Implements a separate thread for communicating with the legacy gui.py application from any of the auto-coded state machine active objects. The ENTRY/EXIT messages are generated in the active object and sent to gui.py via this interface class.
miros.py	A Python module that implements a Hierarchical State Machine (HSM) class (i.e. one that implements behavioral inheritance). This was obtained from http://www.embedded.com/2000/0008. It emulates the QHSM class of the Quantum Framework C/C++ implementations.
time_event.py	Emulates the Quantum Framework TimeEvt class for timer events.
gui.py	Application for starting up Statechart trace gui widgets and via stdio managing messages to and from trace widgets. It also handles the signal button panel.

3.3 Programmable Interface (*Impl.py class)

If general actions functions or guards are added to a UML Statechart, a <StateChartName>Impl.py file will be initially generated. Once generated it is not written over to allow the user to implement manually coded logic. The best way to explain this is with an example. If in our Simple1 Statechart we add a guard called ModeBlock() to the PwrOn signal event transition and we add an entry event called EntryAction() to the Running state than a Simple1Impl.py file is generated that defines the class Simple1Impl. The following is a descriptive part of the file.

```
"""
File: Simple1Impl.py

Date Created:  17-Nov-2009 21:40:45
Created By:    reder

Python custom-implementation class for functions referenced in
the Simple1 Statechart model.
"""
*********Module imports and global code would be here*******

class Simple1Impl(object):
    """
    Simple1 state machine implementation object.
    """
    def __init__(self):
        """
        Constructor
        """
        print "*Accessible: IMPL['Simple1']"
        self.set('ModeBlock', False)

************Standard set, get and clear attribute methods are defined here************

    def ModeBlock(self):
        """
        Implementation Guard method for ModeBlock()
        """
        print "ModeBlock()", "==", self.get('ModeBlock')
        return self.get('ModeBlock')

    def EntryAction(self):
        """
        Implementation Action method for EntryAction()
        """
        pass
```

In the above Simple1Impl class the user would insert action code in the EntryAction method to replace the pass statement and in the ModeBlock() method logic would be implemented to return True or False. Again a certain amount of caution is needed when programming the Simple1Impl class since it is executing within the Simple1 state-machine thread context.

3.4 Using the sim_state_start.py as a Python Module (import of sim_state_start)

The sim_state_start.py script that we have been using to launch our auto generated state machine codes can also be used within your own scripts as a Python module. You can import it. The example script in QF-Py/examples/SimpleRef/start.py shows this and is reproduced next.

Start.py demonstration script of using the sim_state_start module

```
#!/usr/bin/env python -i
"""
Basic example of using sim_state_start module.
"""
import sim_state_start
import sys
import time

def checkState(machine, state):
    """
    Check for a state and timeout if not found.
    """
    timeout = 0
    current_state = ""
    while (state != current_state):
        current_state = sim_state_start.currentState('Simple1')
        timeout = timeout + 1
        if timeout > 100000:
            print "Timeout: looking for expected current state: %s" % current_state
            sys.exit(-1)
    return current_state


if __name__ == "__main__":
    #
    # Start all Statecharts in ./autocode without trace gui widget
    #
    sim_state_start.main()
    time.sleep(5)
    #
    # send event PwrOn to Simple1
    #
    print "Send PwrOn event."
    sim_state_start.sendEvent('PwrOn')
    #
    # wait for current state to enter Idle
    #
    current_state = checkState("Simple1","Idle")
    print "Current state is: %s\n" % current_state
    #
    # send event PwrOff to Simple1
    #
    print "Send PwrOff event."
    #
    # wait for current state to enter Off
    #
    sim_state_start.sendEvent('PwrOff')
```

```
current_state = checkState("Simple1","Off")
print "Current state is: %s\n" % current_state
#
print "Interactive Python prompt next"
```

The start.py example code shows first import of sim_state_start. Next the sim_state_start.main() function is used to start the active object state machine threads but this time with the trace gui widgets disabled. Next a PwrOn event is generated and a while loop is entered to wait for the current state to transition to Idle. Sending a PwrOff event is then done and the code waits for the current state to be Off. Than the script has completed execution and an interactive Python prompted is presented to the user. So watch all the messages you can execute QF-Py/examples/SimpeRef/start.py in your environment.

3.5 Assistance and Problem Reporting

If you have problems or questions using this software send an email message to both Len Reder and Owen Cheng. For all problems please include the following:

1. The xml or mdzip model file you are attempting to generate and execute.
2. Any hand coded Python files.
3. A description of how to run and reproduce the problem.
4. Tell us what operating system you are running on.