# Auto-coding UML Statecharts for Flight Software

Ed Benowitz, Ken Clark, Garth Watney
*Jet Propulsion Laboratory, California Institute of Technology*
*{Edward.Benowitz, Ken.Clark, Garth.Watney}@jpl.nasa.gov*

## Abstract

*Statecharts have been used as a means to communicate behaviors in a precise manner between system engineers and software engineers. Hand-translating a statechart to code, as done on some previous space missions, introduces the possibility of errors in the transformation from chart to code. To improve auto-coding, we have developed a process that generates flight code from UML statecharts. Our process is being used for the flight software on the Space Interferometer Mission (SIM).*

## 1. Introduction

Designs are often specified, formally or informally, as hierarchical statecharts. As space missions become more complex, the software complexity must be communicated both within a software engineering team, and between software and system engineers. In a rapid flight software development environment, it would be desirable to move from statechart to code with as few translation errors as possible. The goal of this work is to automate the translation of statecharts to flight code. Our goal is not to change all flight code into a statechart. Rather, our goal is to take as input existing requirements and designs already expressible as statecharts. Those statecharts are drawn in a Unified Modeling Language (UML) [5] graphical design tool. Then our auto-coder tool transforms a UML statechart to working flight code. The generated code works with the Quantum Framework [3], a reusable statechart library.

## 2. Prior Work

### 2.1. Deep Space 1

The Deep Space 1 (DS1) [1] mission was the first JPL mission to use auto-coding with statecharts. For this mission, auto-coding was used for the fault protection subsystem only. Statecharts were drawn in Stateflow for MatLab [4]. Stateflow's internal auto-coder was used to generate code. The auto-generated code was then post-processed into flight code, compliant with the flight software design team's coding style and constraints. The team reported an overall positive experience with auto-coding, but highlighted the importance of open code generation algorithms.

### 2.1. Deep Impact

Like the Deep Space 1 mission, Deep Impact (DI) [2] used Stateflow as a drawing and simulation tool. Flight code was automatically generated for both fault protection monitors and responses. Deep Impact used an updated version of StateFlow, which was incompatible with the file format version used on DS1. Additionally, DI was written in C++, and had different requirements for auto-coder output. The DI team wrote a new postprocessor as a series of M4 scripts. This tool post-processed the code generated by StateFlow into flight C++ code. In addition, spreadsheet tables were used as a part of the auto-coding process.

## 3. STAARs Process Overview

The auto-coding of UML statecharts fits into a larger software development and verification process at JPL called STAARS (STate-based Architecture and Auto-coding for Real-time Systems). STAARS is a process that loosely combines 5 software tools for the purpose of building rapid executable and verifiable models that can be implemented directly into the application software. Each of these software tools is light-weight and dedicated to a specific task. This provides flexibility not permitted under the large all-purpose commercial modeling tools. Each of these software tools can be swapped in and out or modified to fit the needs of a specific project. The learning curve is low and the cost minimal. The tools of the STAARS process are:

2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)
0-7695-2644-6/06 $20.00 © 2006 IEEE

- *UML Modeling*. The process starts with a drawing tool that supports the UML statechart specifications. The only requirement on the drawing tool is that the output model be saved as an XMI file. Any statechart drawing tool that meets this requirement may be used in this process.
- *State-based framework*. A good state-based software framework is essential to this whole process, providing a library of code to support the execution and representation of statecharts within C or C++. The framework we have selected is the Quantum Framework developed by Miro Samek [3]. This framework allows for an easy transformation from the UML statechart diagrams into code. The framework and design patterns are clean, efficient, and are designed with real-time embedded applications in mind.
- *Autocoding*. Our auto-coder is a light weight tool that provides the automation of what would otherwise be a manual effort of converting statechart diagrams directly into code.
- *Test harness*. This tool provides a method to execute the state machine model. It consists of a few python scripts that allow the user to execute the model in a GUI. The user generates events with the click of a button, and graphically sees states and transition changes.
- *Model checking*. The Autocoder tool will soon be augmented to output verification models in the Promela language. Promela is the input language to a powerful model-checker called SPIN. This tool will give the user the capability to exhaustively check for various correctness properties in the model.

## 4. Drawing Tools

In support of the STAARS process, a trade study was conducted in the summer of 2004 to select a UML drawing tool to be used for the SIM RTC Flight Software design and development. The SIM RTC project desired a common tool that could be used, both by the Flight Software and Control Algorithm development teams, to document the complex behavior of the SIM RTC system via State and Sequence diagrams. In addition, the flight software development team wanted to use the tool to design and document the flight software's C++ class structure, and possibly to generate a standard XMI input format for a flight software auto-coding tool.

The following criteria were evaluated as part of the UML tool trade study:

- *Weight* - Light, Medium or Heavy weight. Is the tool quick and easy to install? Quick and easy to start up? Small learning curve. Does not require a great amount of resources in time or resources to learn and use?
- *Cost* – The tool should be moderately priced, or possibly free if it is open source.
- *User interface* - Is the tool easy to use? Does it have good drawing capabilities?
- *UML support* - Does the tool support most of the UML diagrams we are interested in? (Class, State, and Sequence diagrams…)
- *Export XML* - Does the tool export the UML diagrams in XML – particularly XMI [8] which is defined by the Object Modeling Group.
- *Presentable* - Does the tool produce good looking drawings that can be used for presentations? Cut and past into PowerPoint and Word documents? Large fonts, color coded etc?
- *Cross reference between diagrams* - Can an interaction diagram be setup from a class diagram?
- *Product Support* - How will questions get answered, problems fixed, and product maintained?

In parallel with the UML tool trade study, a prototype state chart auto-coder tool was created in August 2004, reading simple XML state charts created by one of the candidate UML tools. The relative ease with which an auto-coder tool could be created, using as input the XMI state chart representation, convinced the SIM RTC project to undertake the development of a NASA Class B (mission critical) version of the state chart auto-coder in November 2004.

## 5. Auto-coder

Drawing from the lessons learned of DI and DS1, it was important for SIM that we have complete control over the auto-coding process. We developed our own auto-coder, consisting of 3 parts: a parser stage, which reads an input file, an internal model representation, and a code generator. The parser to be swapped out, allowing us to accommodate different drawing tools. Additionally, we provide multiple back ends for code generation, outputting both C and C++.

### 5.1. Auto-coder Inputs

The inputs to the state chart auto-coder are XML files containing XMI [8], the standard meta-data representation of UML state charts. A primary goal of using a UML tool as part of our software development process was to use a non-proprietary XML state chart representation based upon an open standard.

XML provides several advantages for us an input format. It is human readable as a text file, and many free parsers exist for XML on a variety of platforms. As our auto-coder is implemented in Java, we use the standard parsers that come with the Java development kit. An XML document consists of elements and attributes. Ordering and restrictions on these XML elements is typically provided by a custom schema for each application domain. XMI provides constraints on XML files, and provides a meta-model for describing UML within XML.

During the development of the state chart auto-coder, a recurring issue was the "moving target" nature of the representation of state charts in XMI. Over the course of developing the auto-coder, a new version of XMI (from v1.2 to v2.1) was adopted by our chosen UML tool vendor, resulting in a new version of the UML tool and a significant change to the state chart representation in our XML input files. This forced a redesign of the front end of the auto-coder, both to support the new representation, and to try and better accommodate future XMI changes.

Our current assessment is that the XMI representations output by UML drawing tools, while maturing, are not yet in a final form. Future updates to both the UML and XMI specifications could force changes to the front end (input processing) component of the auto-coder. While the state machine model descriptions within XML appear stable between tools, diagram layout was not interchangeable between tools. MagicDraw, for example, uses its own custom XMI extensions to represent diagram geometry.

The UML Diagram Interchange [7] is intended to remedy this situation. Unfortunately, adoption of this standard is mixed among UML tools. The authors of [9] discuss several issues with the XMI representation. We agree with their claim that the many UML and XMI versions make adoption and interchange of UML diagrams between tools less than ideal. They surveyed 8 UML tools and found that only 2 supported diagram interchange. They also suggest that diagram interchange is not adequate, and will need to be extended to support the needs of drawing tools.

## 5.2. Auto-coder Design and Implementation

Learning from the lessons of DS1 and DI, JPL decided to create its own auto-coder, so that we had complete control over the code generation process. With our own auto-coder in hand, we are able to ensure that the generated code meets our mission's requirements for coding standards and implementation.

The auto-coder has been implemented incrementally; as users ask for specific UML functionality, these new features are added. Currently, our auto-coder supports the following features of UML: composite states, events, actions, signals, guards, junctions, orthogonal regions, initial states and deep history states.

Our auto-coder, which is written in Java, can be separated into three areas: the front end, the state machine model, and the back end. The front end is responsible for parsing an input file, and creating the state machine model. The state machine model is a set of Java objects representing every element of the state machine, from States to Transitions. The back end is passed the state machine model, walks it, and then produces code for a given programming language.

Our front end parses MagicDraw's XMI output files. Although we had to make changes to our front end to accommodate new versions of MagicDraw, we were able to do so without making changes to the model representation. The idea is that one could develop a new front end, and so long as it can output a state machine model, the new front end can integrate into our tool chain.

Our state machine model maps each state chart element to a class. For example, we have classes for states, transitions, events, and actions. A State, for example, has direct Java references to its incoming and outgoing Transition objects. Figure 1 shows the classes used within our statechart model. Notice the use of the composite pattern [13] used to describe composite states.

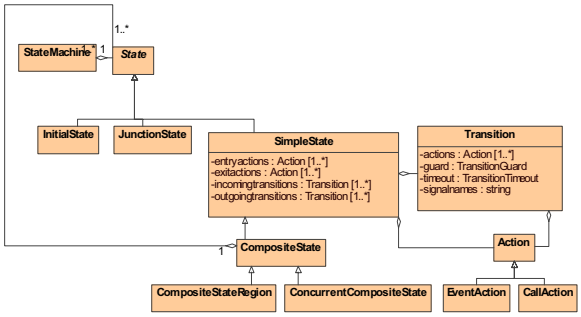


**Figure 1. State machine object model.**

Our back end's responsibility is to walk through the state machine object model, generating code. While we considered using the visitor pattern [13], we anticipated that various code generators would like to walk the state machines in different orders. We currently provide code generators for C, C++, and also output a Python GUI for testing the behaviors of state machines. This demonstrates that our backend is flexible enough to accommodate multiple outputs. An advantage here is that a given flight project has the complete source to the code generator. A project can

customize the coding style, or target a new statechart library, while still reusing the front end and the object model.

We have added several restrictions on state machines to simplify the auto-coding process. We do not allow nested orthogonal regions, that is, one orthogonal region nested within another. Additionally, junctions currently only are allowed to have two outgoing transitions. This restriction allows junctions to be implemented as simple if else statements, but the restriction could easily be removed with a small change to the auto-coder.

For safety, we force deep history states to have exactly one outgoing transition. This is enforced for the following reason: suppose a transition occurs to a deep history state, but the history state's parent state has never been visited yet. The semantics in this case are undefined by UML, so we avoid this situation by requiring a default transition out of the history state.

### 5.3. Auto-coder Output

Most developers hand-code statecharts using the Quantum Framework. Developers quickly noticed that the hand translation was quite repetitive; there was only one cookie-cutter way to translate a statechart to code. This hand generated code was quite readable. So the goal with the autocoder output was to generate code that looked exactly like the hand-generated version, in the style recommended by Samek [3].

Samek has each state represented as a method within a class. Each state method has an event passed in as a parameter. A switch statement, based on the event, determines the behavior of the state. State transitions essentially take place by changing function pointers.

Samek's Quantum Framework code is originally written in C++. He provides a C version, which essentially adds many macros to C to simulate C++ features such as vtables and subclassing. We found that both auto-coded and manually coded C++ statecharts looked considerably cleaner than the corresponding C versions, which were polluted by the use of many macros.

## 6. Use of the Quantum Framework

The motivation to use a state-based framework was driven by several observations. We did not have a consistent method for implementing hierarchical state machines. Previously, developers would typically use a combination of tables, switch statements or various other design patterns to implement state machines in code. We needed to implement these state machines in a systematic and uniform way.

There is a trend towards specifying behaviors using the UML standard for state machines. Previously, statecharts were often specified in Powerpoint, with much ambiguity in the notation. Some of our existing implementations of these Powerpoint state machines used unspecified semantics. This resulted in a miscommunication between the systems and software disciplines. For a discussion of differences between UML and other statechart semantics, see [12].

Correct implementations of complex hierarchical state machines are difficult and labor intensive. Existing design patterns to implement state machines such as tables and switch statements break down when using hierarchy or other statechart features like history states or orthogonal regions.

As previously mentioned, the state-based framework we use is the Quantum Framework developed by Samek [3]. The Quantum framework is a small light-weight framework intended for real-time embedded applications. The entire framework consists of approximately 800 lines of code and comes with both C and C++ implementations. The underlying architecture consists of Active objects. Active objects are hierarchical state machines that communicate with each other via an exchange of event instances. The framework is extremely flexible. The user may decide to only use the base class for hierarchical state machines, or may decide to use Active objects with the Quantum framework's Publish and Subscribe mechanism for event communication. The framework also comes in different flavors – a simple one thread or multithreading environment.

For our applications, we use the simple one thread environment and rely on the specific application code to provide the tasking and scheduling. This allows our use of the Quantum framework to be completely portable to desktop work-stations and our embedded VxWorks targets. Out the box, the Quantum Framework supports only event-driven applications. State machines "sleep" until an incoming event causes the state machine to execute. If more than one state machine receives an event, a priority scheme determines the order of execution. For one of our projects, we needed a more deterministic approach. So we selected to have a rate-driven application as opposed to an event-driven application. Each state machine would only execute at a specific time-slot based on a scheduler. The state machine would then drain its event queue consuming events at this time only. In order to support this rate-driven paradigm, we needed to make slight modifications to the Quantum framework. Due to the small size and good design

constructs used by Samek, our modifications to the Quantum framework were fairly easy to undertake.

## 6. Future Work

Our goal is not only to auto-generate code from statecharts but to verify that these statecharts will always satisfy their design requirements. We plan to augment the auto-coding tool to output verification models that can be used for this purpose. These verification models can be used to verify the correct behavior of a single state machine or the correct interaction of multiple state machines in a multi threaded environment. We plan on performing this model-checking in the SPIN environment.

SPIN [11] was developed by Gerard Holzmann as a tool for detecting software defects in concurrent system designs. The input language to SPIN is called Promela. The autocoder will then output Promela as well as C/C++ code. Promela however is not intended to be a low-level implementation language. In addition, there is no state-based framework like the Quantum framework in Promela. Even if a complex state machine were implemented in Promela, the state space would be far too large to perform adequate model checking. To solve this problem we will combine Promela and C code in our verification model. Promela supports a C interface [10]. The strength of Promela lies in the specification of concurrent processes. The strength of C is as an implementation language. Combining these 2 languages will allow us to use the power of SPIN to search through the state space of our model and verify that certain correctness properties or design requirements are satisfied.

## 6. Acknowledgements

This work was supported by the NASA's Space Interferometer Mission (SIM), and by the Jet Propulsion Laboratory (JPL), California Institute of Technology under a contract with the National Aeronautics Administration. Additional funding was provided by JPL's Laboratory for Reliable Software (LaRS), and by a JPL R&TD research proposal award for "Software Assurance for the Emerging Discipline of Model-Based Design." supported by JPL's section 512.

We thank Alex Murray for his work on the initial auto-coder, and Hanry Hartounian for his work in obtaining performance numbers.

## 7. References

[1] N.F. Rouquette, T. Neilson, and G. Chen, "The 13th Technology of Deep Space One", *Proceedings of the 1999 IEEE Aerospace Conference*, Vol 1, March 1999, pp. 477-487.

[2] K. Barltrop, E. Kan, J. Levison, C. Schira, and K. Epstein, "Deep Impact: ACS Fault Tolerance in a Comet Critical Encounter", *Advances in the Astronautical Sciences*, Vol. 111, 2002, pp. 111-126.

[3] Samek, M., *Practical Statecharts in C/C++*, CMP Books, San Francisco, 2002.

[4] Stateflow. www.mathworks.com/products/stateflow/

[5] Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Boston, 1999.

[6] MagicDraw. www.magicdraw.com

[7] OMG. Unified Modeling Language: Diagram Interchange version 2.0, July 2003. OMG document ptc/03-07-03. .www.omg.org , 2003.

[8] OMG. OMG XML Metadata Interchange (XMI) Specifi cation. OMG Document 03-05-02. www.omg.org, 2003.

[9] M. Alanen, I. Porres, "Model Interchange Using OMG Standards," *euromicro*, pp. 450-459, 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005.

[10] G. Holzmann, R. Joshi, "Model-Driven Software Verification", Proceedings of the 11th SPIN Workshop, *Lecture Notes in Computer Science*, Volume 2989, Springer Verlag, April 2004.

[11] Holzmann, G. *The SPIN Model Checker*, Addison-Wesley, Boston, 2003.

[12] M.L. Crane, J. Dingel. "UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal". *Proc. of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005)*, Montego Bay, Jamaica, October, 2005.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Softwar*e, Addison Wesley, Boston, 1994.