From Protocol Specification to Statechart to Implementation

Kiri L. Wagstaff, Ken Peters, and Lucas Scharenbroich
Jet Propulsion Laboratory,
California Institute of Technology

October 11, 2008

Abstract

This paper describes two major steps in model-based system design and implementation: 1) the process involved in converting a text-based system specification into a UML-compliant, graphical statechart, and 2) the use of automatic code generation tools to convert the statechart into a C or C++ implementation. We also describe how to use the graphical, interactive "test harness" to test the behavior of the statechart's generated code, a very useful tool for system (protocol) design refinement. Finally, we describe how to automatically generate a Promela version of the statechart model that can be verified using the SPIN model checker. Throughout the paper, we focus on how these tools can be used to make the **communications protocol development process** more streamlined and reliable.

1 Purpose and Intended Audience

Model-based design benefits designers, customers, and future adopters and enhancers of the system being designed. These benefits are generally phrased in terms of the work associated with software system development. However, our recent experience has highlighted the specific benefits that can accrue to communications protocol designers who adopt a model-based approach to protocol design. First, the graphical statechart is a powerful tool for visually illustrating the behavior of a system that adheres to the protocol. Second, it is possible to interactively exercise the statechart and determine whether it behaves appropriately in a variety of situations. Third, since statecharts take action based on events, to which messages exchanged between entities naturally correspond, they are ideal for modeling communications protocols. The intended audience of this document includes protocol designers, testers, and implementers. Some specific instructions will only apply for those working on JPL systems, with the JPL Autocoder.

2 Statecharts for Protocol Design

Statecharts are currently used, both formally and informally, to illustrate system design and behavior. A statechart consists of a set of disjoint *states*, often represented as boxes, and *transitions* between the states, often represented as arrows. A transition occurs due to an *event*, specified as an annotation to the transition arrow. States may also be associated with entry or exit actions.

2.1 Statechart Example

A simple traffic example is shown in Figure 1. This statechart models the current state of the intersection in terms of what kinds of traffic flow are permitted: north-south or east-west, with the "slow" states modeled separately to permit a non-abrupt transition between the two. Transitions specify how the system moves between these states. The filled circle at the upper left specifies the start state (NorthSouth_Go).

This statechart contains transitions that use both events and timers. The "Car_E_W" event indicates that one of the car sensors in the east-west direction registered the presence of a car waiting. If the system is currently in NorthSouth_Go, this event will cause it to transition to NorthSouth_Slow and change the north-south lights from green to yellow. From this state, a timer event ("at(5)") specifies that 5 seconds later, the system exits this state, sets the north-south lights to red, and transitions to EastWest_Go, setting the east-west lights to green. From this state, the system waits until a car is detected by one of the north-south sensors. Note that in this state, if the

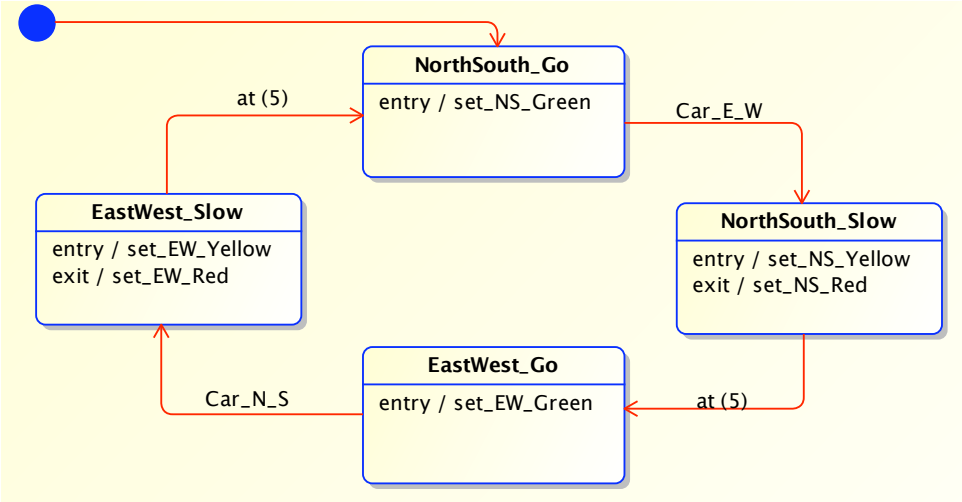


Figure 1: Simple statechart modeling traffic flow control at an intersection. There are four possible states, corresponding to the directions of permitted traffic flow. The presence of a car at one of the East-West or North-South sensors causes the system to transition from a "Go" state to a "Slow" state, and five seconds later, it transitions to the other "Go" state. Entry and exit actions update the traffic lights.

Car_E_W event happens, the system does nothing. The system responds only if an event happens that is associated with an action or transition from the current state.

Figure 2 shows a more complex model of the same system, in which the current state of the intersection's traffic lights is also explicitly modeled. Orthogonal regions, separated by horizontal dashed lines, permit the independent modeling of traffic flow (top), north-south lights (middle), and east-west lights (bottom). The full system configuration at any time includes one state from each orthogonal region. The value of explicitly modeling the lights is that the system designer or tester can see the critical system output (color of the lights) at a glance, and system testing can directly confirm whether the current light configuration is as expected. As discussed later in this document, the statechart can be converted into a Promela model for model-checking purposes, which enables answering critical queries such as "Can the system ever reach a state where it is in NorthSouth_Green and EastWest_Green at the same time?"

2.2 Statecharts for Communication Protocols

Statecharts can be put to good use in modeling communication protocols. At the most basic level, the statechart can track whether the system is in a "send" or "receive" state. Full-duplex, half-duplex, and simplex communication result in different kinds of transitions between these activities. Fundamental events may include notification from the application layer that user data is ready to be sent and notification from the link layer that data has been received. Timer events can be used to implement delays and timeouts to permit synchronization between communicating units and the detection of link dropout.

To create a protocol statechart, the first step is to identify

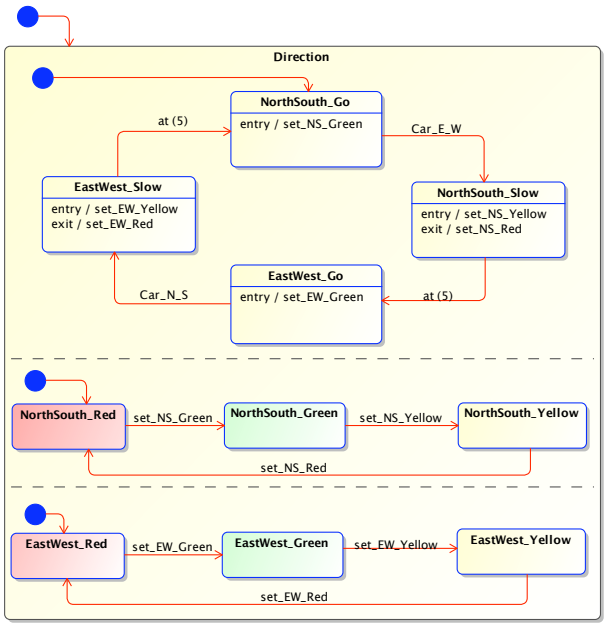


Figure 2: Statechart illustrating a traffic intersection, with orthogonal regions to represent the current state of the lights as well. The orthogonal regions are separated by horizontal dashed lines.

1. the discrete system states and
2. the transitions between those states

to be modeled in the statechart. This process requires the identification of the desired level of abstraction to be represented, anywhere from the simple send/receive two-state model just described down to the modeling of individual bit transmission on the link. For most purposes, capturing behavior somewhere between these extremes is appropriate.

Decisions about what to include in the statechart can be guided by considering how the statechart will be used. For system documentation, all behavior that implementers and customers need to understand should be included. For automatic code generation, all behavior that needs to be under statechart control (e.g., automatically updated and checked for consistency each time the statechart is changed) should be included. For model checking, enough detail must be included so that the relevant system attributes can be checked. In general, more detail is better, although large diagrams can become unwieldy or illegible. In such cases, *submachines* can be used to encapsulate part of the system in a separate, linked statechart. We will show an example of this shortly in our Proximity-1 statechart.

The specification of entry and exit actions also relates to the decision about abstraction. In Figures 1 and 2, the actions are simply events which trigger state transitions in other regions. If connected with hardware that controls the actual lights, this system would also need to specify actions that update the physical state of the lights. These actions can be specified as function calls, such as set_NS_Yellow(). This function call should not be confused with the event set_NS_Yellow. Events only affect the internal system state, as specified in the statechart. Interactions with hardware

(e.g., via device drivers) must be specified with this kind of function call, which must be implemented outside of the statechart. With respect to abstraction, low-level behaviors are also best encapsulated in these external calls.

The statechart itself can be created with any statechart drawing tool. We used MagicDraw[1], which saves out statecharts in a generic, open format (UML). If there is a desire to automatically generate C or C++ code, to use the interactive test harness, or to generate a Promela model of the statechart, then it is advisable to use this tool, since the JPL Autocoder (described later) is tailored to the UML structures it uses.

2.3 Case Study: Proximity-1 Reference Implementation Statecharts

We recently completed a project in which we provided a statechart-based implementation of the Proximity-1 protocol for use in the JPL Protocol Test Lab. The Proximity-1 protocol (CCSDS, 2006) is currently being used by the Mars Exploration Rovers to communicate with Mars Odyssey and the Mars Reconnaissance Orbiter. It has been adopted as the standard for future Mars missions such as the Mars Science Laboratory (MSL), the next rover mission to Mars. We have designed statecharts to describe the protocol and to provide a reference implementation against which other systems that adhere to the protocol can be tested. The implementation consists of automatically generated code that was produced from the statecharts directly by the JPL Autocoder (Benowitz et al., 2006).

2.3.1 Proximity-1 Statecharts

Working from the Proximity-1 specification, we developed a high-level statechart that consists of five orthogonal regions (see Figure 3). The first region tracks the current activity of the system as a whole: it can be one of the Idle, Hailing, DataServices, or CommChange states. The second and third regions track the current state of the transmitter (Off, Idle, or Transmitting) and receiver (Off, Idle, or Receiving). The fourth and fifth regions track the current state of the COP-P progressive retransmission part of the Proximity-1 protocol. FOP-P determines what frame should be sent next: a new frame, a previously sent but unacknowledged frame, a PLCW (Proximity Link Control Word), etc. The receipt of an invalid PLCW switches FOP-P into the ActiveTimer state, which transitions to the Resync state if a valid PLCW is not received before the timer expires. FARM-P determines what actions should be taken when a frame is received.

The details of the Hailing, DataServices, and CommChange states in the first orthogonal region (MAC) are contained in submachines, indicated by the chain icon in the lower right corner of these states. Encapsulating these details in submachines permits the high-level state to remain clear and uncluttered. The system begins in the Idle state, and when it enters, it performs the action of calling `mac_go_inactive()`. All actions that appear as function calls are hooks into lower-level methods that are implemented manually. The system transitions to Hailing if it receives a "SetTransmit" or "SetListen" directive. Each of these transitions is associated with a different *entry point* to the submachine. The submachine associated with Hailing is indicated at the top of the state as prox1hail. There are also two *exit points* from the submachine; a successful hail exits at "doneHailing" and transitions to DataServices. If the hail fails, the "failedHailing" exit point permits the system to transition back to Idle.

When the system is in DataServices, it can actively send and receive data frames as needed. When there is no more data to send, it exits via "doneData" back to the Idle state. At any time while in DataServices, a local or remote request to change the communications parameters (frequency, bit

[1] http://www.magicdraw.com/

rate, etc.) may occur and causes the system to switch to the CommChange state. If this succeeds, the system returns to DataServices, and if it fails, the system drops back to Hailing to regain the link.

2.3.2 Proximity-1 MAC Submachines

Figure 4 shows the details of the three submachines referenced in the MAC region of the main chart. The first is the prox1hail submachine, which includes both entry and exit points for the submachine. As specified by the protocol, the SetListen directive causes the radio to wait for hailing directives to be received, and the SetTransmit directive instructs the radio to issue hailing directives and wait for a valid frame to be sent back. If a valid frame is not received within a specified time, then the system loops from S35 back to S31 and tries again. After too many attempts, the system exits via the "failedHailing" exit point. A successful hail exits via "doneHailing" and transitions to DataServices.

The prox1data submachine, which is associated with the DataServices state, appears in Figure 4(b). From the perspective of the MAC, nothing happens while the system is in the SendRecv state. All of the activity in terms of deciding which frames to send and parsing the frames that are received takes place in the FOP-P and FARM-P regions, and the actual sending and receiving of frames happens in the Transmitter and Receiver regions. This statechart only responds when a local (LNMD) or remote (RNMD) "no more data" directive is received. An LNMD directive causes the system to wait for pending frames to go out and then send an RNMD to the other radio. It then waits for an RNMD to come back, which completes the handshake and permits the session to be terminated. Alternatively, if the system receives an RNMD while in SendRecv, it waits for a local LNMD to occur before sending the handshake RNMD back. Both paths terminate in the "doneData" exit point, which transitions the MAC back to Idle.

The prox1commchange submachine in Figure 4(c) provides the details for a local or remote communications change request. A local request is handled as follows. The radio waits for the queue of pending frames to empty, then sends a Remote Communications Change Directive (RCCD) to the other radio. Once its FIFO queue empties, the radio waits for the communications link to be lost (modeled as a "BitLockFalse" event), indicating that the other radio has changed its parameters. If this does not happen before a specified amount of time, the radio loops back to WaitForEmpty and re-sends the RCCD. As in the hailing submachine, after too many tries the system will give up and exit through the "failedCommChange" exit point. If "BitLockFalse" occurs, the radio updates its local receive parameters, waits for a valid frame to be received from the other radio, and then updates its own transmit parameters. At that point, the two radios can resume regular communications by exiting through "doneCommChange" back to DataServices.

A remote request, received on the link by the local radio, is handled as follows. The radio waits for its local FIFO queue to empty, pauses, and then updates its transmit and receive parameters. It transitions to RadiateCarrier and then RadiateAcqIdle to permit the remote requestor to receive a valid frame and update its own parameters, and the two radios can then resume their session.

3 Automatic Code Generation

Using established coding templates, statechart models can be automatically mapped into code by means of a statechart Autocoder tool. Rather than treating statecharts as just part of the design documentation, they can be maintained along with the source code, subject to the same level of scrutiny and review.

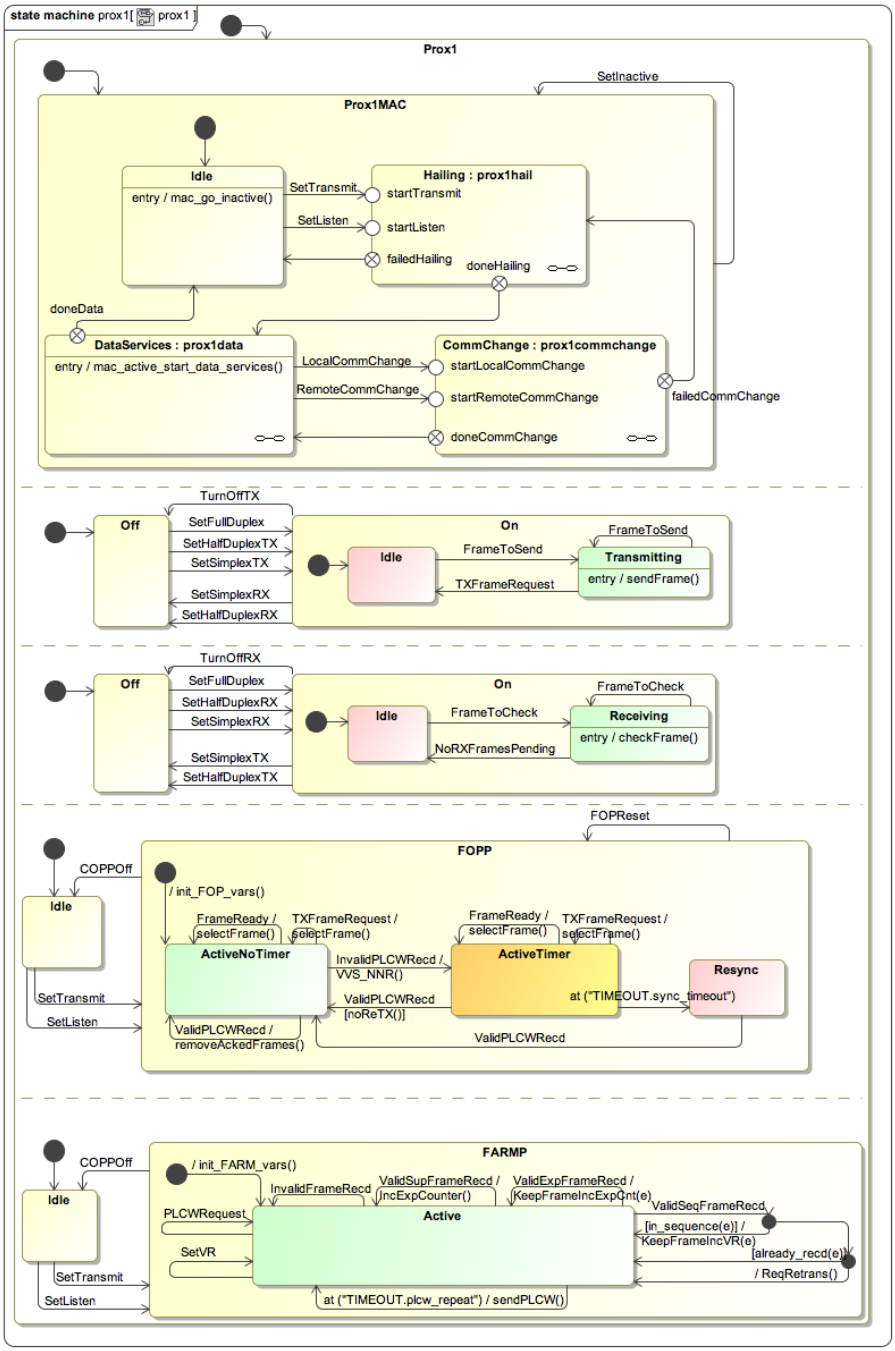
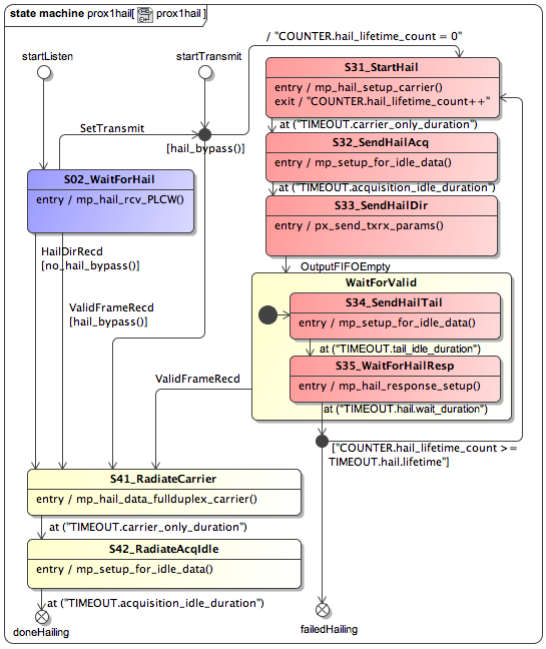
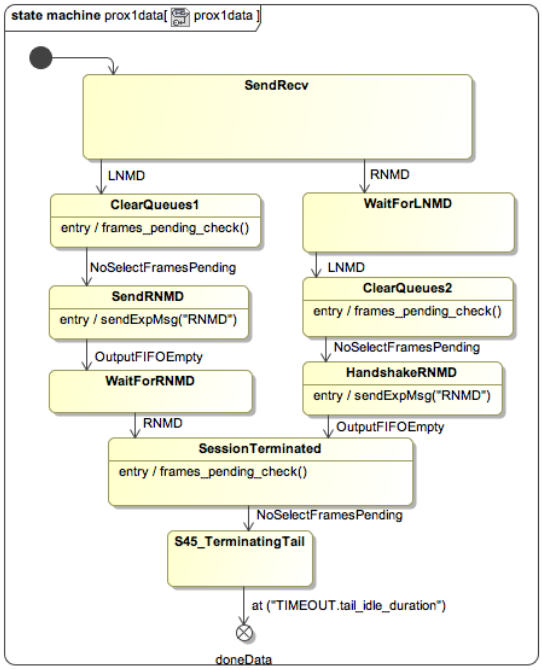


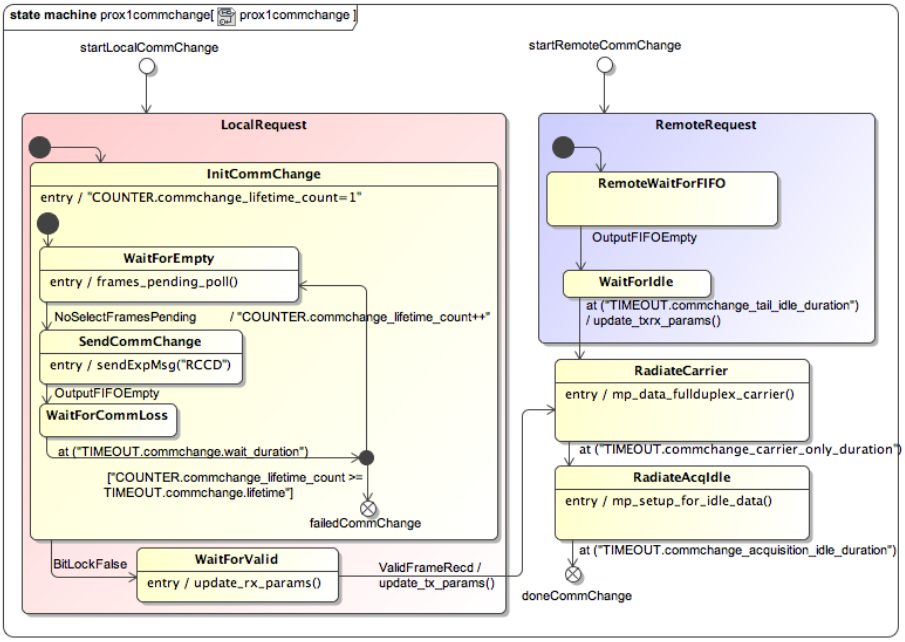
Figure 3: Proximity-1 statechart, consisting of five orthogonal regions. From the top, they are: the MAC region (state of the Medium Access control layer), transmitter, receiver, FOP-P (sender logic), and FARM-P (receiver logic).



(a) Hailing



(b) Data Exchange



(c) Communications Change

Figure 4: Proximity-1 statechart submachines.

3.1 Background and History

The first JPL mission to use auto-coding with statecharts was Deep Space 1 (1998–2001), a mission designed to demonstrate advanced technology such as ion propulsion, autonomous navigation, onboard reactive planning, and others (Rouquette et al., 1999). Matlab's Stateflow tool[2] was used to design and generate code for the fault protection subsystem. Stateflow was also used to generate fault protection code for Deep Impact, which studied the results of an artificial impactor striking comet Tempel 1 in 2005 (Barltrop et al., 2002).

After evaluating these experiences, JPL decided to implement its own lightweight model-based code generation tool to provide the following advances over Stateflow: 1) a non-proprietary file format in which the statecharts are stored, 2) precise control over which programming constructs are used, to ensure adherence with flight software requirements, and 3) the ability to plug in different front-end statechart drawing tools and different back-end output modules. For example, each mission can impose its own coding standards that should be applied in the code being generated. As noted below, the code generator can also output other forms of code, such as a model representation that can be used by a model-checking tool. The resulting JPL Autocoder was used to model and generate code for the fault protection subsystem of the Space Interferometry Mission (SIM) (Marr, 2003) and the Proximity-1 protocol, as discussed above.

3.2 The JPL Autocoder

The JPL Autocoder (Benowitz et al., 2006) takes in statecharts encoded in UML files and can produce C or C++ code. Development on the Autocoder began in 2004, and the Autocoder itself is currently classified as NASA Class B (mission critical) code. Statechart features supported by the Autocoder include: initial states, composite states, events, actions, signals, guards, junctions, orthogonal regions, submachines, and deep history states. The Autocoder expects as input UML consistent with that produced by the MagicDraw visual UML modeling tool.

The generated C/C++ code uses the standard statechart translations provided by the Quantum Framework (Samek, 2002b). Each state is represented as a method that takes the triggering event in as an input parameter. The body of the method is a switch statement that, based on the event, dictates the resulting behavior (execute an action, transition to another state, etc.). The Autocoder generates code for all components of the statechart as well as inserting calls to external methods that implement low-level actions.

3.3 How to Use it

After creating a statechart using MagicDraw and saving it out as a .xml file, you can generate C or C++ code using the Autocoder as follows. **Note: This applies only from inside JPL.**

1. Download the latest JPL Autocoder package from this website:
   `http://alab.jpl.nasa.gov/mbe/pages/download.html`
   It is currently available for Linux and Windows systems.

2. Compile the system using the instructions here:
   `http://alab/mbe/pages/RunningATest.html`
   There are also instructions for running the test state chart provided.

3. Make a copy of `src/test_harness-C`. Replace `Test.xml` with your .xml file. Modify the `Makefile` to refer to your statechart and do `make clean; make all`.

[2] http://www.mathworks.com/products/stateflow/

Table 1: Semantic lines of code manually and automatically generated for the Proximity-1 reference implementation, calculated with the SLiC version 2.0 tool.

Component	Lines of Code
UML Statechart	2049 (C) 674 (Python)
Proximity-1 implementation	674 (C)
QP API	198 (C)

The Autocoder will create several .o files in the `linux/` subdirectory as well as the executable, `linux/active` (the names can be modified in the `Makefile` to fit your application). Running this executable will start up a copy of your statechart. To send events to it, you can either write your own interface and link it into the object files, or use the interactive test harness, which is described in the next section.

3.4 Case Study: Proximity-1 Reference Implementation Code Generation

We used the JPL Autocoder to generate C code as the basis for a reference implementation of Proximity-1. We provided the statecharts described in Section 2.3 as input. The first row of Table 1 shows the lines of code automatically generated by the JPL autocoder. The generated C code provides the implementation for the statechart, and the generated Python code provides the interactive test harness framework.

The same table reports the additional code that had to be written manually to complete a full reference implementation. The automatically generated code defines the states, transitions, timers, and all other system details captured by the statechart. Additional code must be written to implement each of the low-level functions defined as actions (indicated with function calls) within the statecharts. Finally, additional code must be written manually to connect the statechart implementation with the Quantum Platform (QP) framework (Samek, 2002b), which provides the infrastructure needed to run the state machines, dispatch events, track timers, and so on. However, once these manual components are written, the statechart can be modified and the C code regenerated without re-writing the manual functions, unless different behavior is needed at the low level. Overall, we see that the autogenerated C code amounts to 2.3x the amount of manually written code. In other terms, the autogenerated code constituted 70% of the code needed for the reference implementation.

4 Interactive Statechart Testing

The JPL Autocoder can also produce a Python graphical interface to the generated code using the Tkinter package, which is built on Tcl/Tk. The entire model can then be executed in simulation, with an interactive interface in which the user clicks on buttons to send events to the state machine. The developer can conduct a series of behavioral tests to confirm that the model, and the generated code, respond as expected and that they match the specification. In this way, many design errors can be caught and corrected prior to integration with a larger code base or conducting hardware tests.

4.1 How to Use it

To run the test harness, first follow the compilation process described above. Then run `./setup.py`. This will create several new windows:

- Light brown: DMS window. If it is not running, type `./dms.py` in this window.
- Blue: GUI window. If it is not running, type `./gui.py` to start up the GUI, which will pop up two Tk windows: one with the state machine shown in simulation and one with the list of signals (events) that you can send to the state machine.
- Black: Application window. Type `linux/active` to start running the executable. This should also cause the GUI Tk window to highlight the start state in green, indicating that the state machine is now running.
- Brown: DMS terminal. Type `python -i ./dmsterminal.py` to start up the interactive terminal for controlling the state machine.

At this point, you can click on signal buttons to send that event to the state machine and observe how it responds. You can also run scripts through the DMS terminal to automate the process of interacting with the state machine (for example, to perform regression tests). At any time, click the "Shutdown" button to cleanly exit the test harness.

4.2 Case Study: Proximity-1 Reference Implementation Statechart Testing

Our experience in the process of developing the Proximity-1 reference implementation has been that the graphical test harness allowed us to spot several bugs much earlier than if we'd waited for later downstream testing to catch them. In one example, the log output of all of the state machine transitions showed that after the no-more-data handshake completed, the system correctly returned to the Idle state. However, when we viewed the process graphically, it became immediately obvious that the other four orthogonal regions had not returned to their Idle states. We had forgotten to specify the generation of the TurnOffTX, TurnOffRX, and COPPOff events as part of the return to Idle in the MAC region. This was an easy fix, and correct behavior was immediately confirmed, again with the interactive test harness. The test harness is invaluable for exactly this kind of testing, and the fact that you can instantiate multiple copies of the same machine permits testing a variety of network configurations to stress-test any protocol.

5 Statechart Model Verification with SPIN

The JPL Autocoder can also produce a Promela (Process or Protocol Meta Language) representation of the statechart for use with automated verification tools. Promela is the language used to represent models that can be checked by the SPIN tool (Holzmann, 2003). SPIN was designed to detect software defects in concurrent system designs, which is ideal for complex statecharts with concurrent regions. SPIN can also perform reachability and other useful analyses of the model. One of the barriers preventing more widespread use of model verification tools such as SPIN is the large amount of manual effort that must be invested to create an accurate model of the system and re-invested each time the system is modified or updated. The Autocoder greatly reduces this burden by automatically providing an up-to-date Promela model each time the statechart changes and code generation is performed.

5.1 How to Use it

The model verification process is described in detail by Garth Watney in a report titled "Building Verification Models from UML Statecharts: Design and Users Guide Version 2.1", which is available at `http://alab.jpl.nasa.gov/mbe/pages/Documentation.html`. It includes the simple traffic light statechart as an example and demonstrates how to check whether it can ever reach a state where the lights in both directions are green at the same time.

6 Summary

In this document, we have described the process of using model-based design to develop a statechart representation of a software system, particularly with an eye towards communications protocol development. The JPL tools available for this work include MagicDraw, for designing the statechart and writing it out in UML format, and the JPL Autocoder, for converting the statechart to C or C++ code, a Python test harness representation, and/or a Promela model for later model checking. Our description of this process includes a case study in which we created a reference implementation of the Proximity-1 protocol using statecharts and automatic code generation. We found that most (70%) of the code necessary for the reference implementation could be automatically generated. The effort involved in generating that code was instead devoted to designing and testing the statechart itself. We see this process as freeing the protocol designer to focus on high-level (protocol-level) behavior before needing to delve into low-level bit transfer and link-level details.

Acknowledgments

We thank Ken Clark, Garth Watney, and Eddie Benowitz for their initial work on the JPL Autocoder that made this kind of development possible. Garth and Eddie also helped greatly with our initial work with statecharts and Proximity-1, specifically on the Hailing and CommChange submachines. Garth has invested a lot of effort in packaging the Autocoder and providing tutorials and documentation to make it more usable by a wider audience.

We also thank Peter Shames for his support of this work, in terms of finances and enthusiasm, and Greg Kazz for providing his time and Proximity-1 expertise. We thank Leigh Torgerson, Jackson Pang, and John Veregge at the JPL Protocol Test Lab for their assistance and support of the reference implementation.

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

References

Barltrop, K., Kan, E., Levison, J., Schira, C., & Epstein, K. (2002). Deep Impact: ACS fault tolerance in a comet critical encounter. *Advances in the Astronautical Sciences*, *111*, 111–126.

Benowitz, E., Clark, K., & Watney, G. (2006). Auto-coding UML statecharts for flight software. *Proceedings of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*.

CCSDS (2006). *Proximity-1 spacelink protocol: Data link layer* (Technical Report 211.0-B-4). Consultative Committee for Space Data Systems (CCSDS).

Holzmann, G. (2003). *The SPIN Model Checker*. Addison-Wesley.

Marr, J. C. (2003). Space Interferometry Mission (SIM): Overview and current status. *Proceedings of the SPIE* (pp. 1–15).

Rouquette, N. F., Neilson, T., & Chen, G. (1999). The 13th technology of Deep Space One. *Proceedings of the 1999 IEEE Aerospace Conference* (pp. 477–487).

Samek, M. (2002a). *Practical Statecharts in C/C++*. CMP Books.

Samek, M. (2002b). *Practical statecharts in C/C++: Quantum programming for embedded systems*. CMP Books.