AUTOMATIC CODE GENERATION FOR INSTRUMENT FLIGHT SOFTWARE

Kiri L. Wagstaff, Edward Benowitz, DJ Byrne, Ken Peters, and Garth Watney

Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109, USA,
Email: <firstname.lastname>@jpl.nasa.gov

ABSTRACT

Automatic code generation can be used to convert software state diagrams into executable code, enabling a model-based approach to software design and development. The primary benefits of this process are reduced development time and continuous consistency between the system design (statechart) and its implementation. We used model-based design and code generation to produce software for the Electra UHF radios that is functionally equivalent to software that will be used by the Mars Reconnaissance Orbiter (MRO) and the Mars Science Laboratory to communicate with each other. The resulting software passed all of the relevant MRO flight software tests, and the project provides a useful case study for future work in model-based software development for flight software systems.

1. INTRODUCTION

A key challenge in flight software development is the need for high-reliability software that often must be produced in limited development time. Automatic code generation techniques can address these exact concerns, as they enforce standard programming practices and produce consistent code (both self-consistent and consistent with required coding style). Past JPL space missions such as Deep Space 1 [1] and Deep Impact [2] have used some automatic code generation to specify the desired behavior of their fault protection subsystems. However, the majority of flight software is still developed and written manually.

We have applied model-based design and code generation to produce software for a communications protocol, in which the system must handle both local and remote directives (events). The Proximity-1 protocol [3] is currently being used by the Mars Exploration Rovers to communicate with the Mars Odyssey orbiter, and it has been adopted as the standard for future Mars missions such as the Mars Science Laboratory (MSL), the next rover mission to Mars. In particular, the Mars Reconnaissance Orbiter (MRO), which is currently in Mars orbit, will use this protocol to communicate with MSL after it lands, in 2010. At the beginning of this project, MRO already had the basic Proximity-1 functionality running on its Electra UHF radio. However, it required a software update to add the ability to initiate and respond to communications change requests (e.g., new data rate or transmit frequency). Since this functionality had already been implemented for MSL, mission developers began work to adapt the MSL implementation for use on MRO. In parallel, we designed statecharts for the hailing and communications change parts of the Proximity-1 protocol. We then used the JPL Autocoder to convert the diagrams into C code, packaged the code into a library, integrated it into the flight software, and ran the resulting code through a series of MRO flight software tests. Although our implementation was not ultimately used for the MRO flight software update, it successfully replicated the system functionality, and the statecharts have provided the basis for an implementation of the entire Proximity-1 protocol, currently in progress.

This paper contributes a description of how model-based design and code generation can be used to good effect in flight software systems. Using the Proximity-1 experience as a case study, we also highlight the benefits obtained from this development approach. As a result, we advocate the use of model-based design and code generation to focus human time on the design of the system and leave implementation details to a deterministic, well understood process.

2. AUTOMATIC CODE GENERATION

2.1. Model-Based Software Development

In model-based engineering, the model is an integral part of the software development process. Component interaction can be captured in UML Sequence diagrams, while dynamic behavior can be captured in UML Statechart diagrams. A statechart contains rectangles, representing states, and arrows that represent transitions between

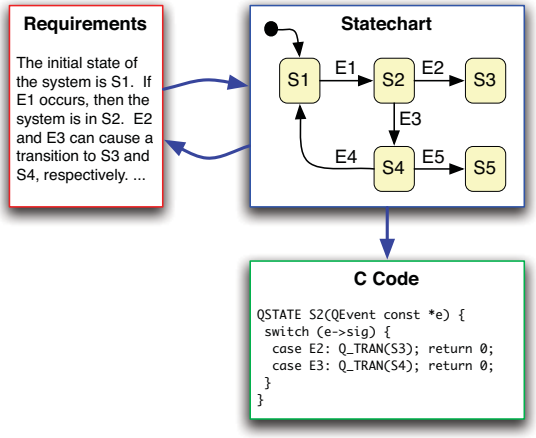

Figure 1. The process of translating system requirements into a statechart that can be converted into C code. The resulting statechart can reveal places in the requirements that require further specification.

states, annotated with the event that causes the transition to occur. Each state can specify actions that are taken on entry or exit. Transitions can also specify guards that restrict the contexts in which the transition will occur.

Using established coding templates, these statechart models can be automatically mapped into the application 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. In addition to providing increased visibility into the actual design of the system, joint reviews of the statechart by system engineers and software engineers can quickly identify loosely worded or misunderstood requirements, and provide an opportunity to correct them. Figure 1 represents this process graphically.

2.2. The JPL Autocoder

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 [1]. Matlab's Stateflow tool[1] 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 [2].

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) [4].

C/C++ Output. The JPL Autocoder [5] 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: composite states, events, actions, signals, guards, junctions, orthogonal regions, initial states, and deep history states. The Autocoder expects as input UML consistent with that produced by the MagicDraw[2] visual UML modeling tool.

The generated C/C++ code uses the standard statechart translations provided by the Quantum Framework [6]. 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.

Python Simulator Output. The 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 codebase or conducting hardware tests. An example of this kind of benefit is given in Section 4.4.

Promela Model Output. Finally, the Autocoder can produce a Promela 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 [7]. 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 providing an up-to-date Promela model each time the statechart changes and code generation is re-done.

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

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

Each of these output products (C/C++ code, Python model, and Promela model) are generated from the same original source, which is the UML statechart. Therefore, each time the statechart changes and the code is regenerated, it naturally propagates to all three outputs. As a result, the documentation (statechart) is always up to date with respect to the implementation, the simulation, and the model verification tools.

3. ELECTRA RADIO FLIGHT SOFTWARE

The Electra UHF radio is an advanced telecommunications and navigation subsystem for Mars missions, providing (among other features) communication among spacecraft at Mars using the Proximity-1 protocol [3]. The Electra hardware and its class B (mission critical) software were designed at JPL using relatively traditional processes. In particular, the Proximity-1 implementation was hand-coded by software engineers who had studied the protocol specification document to be able to translate the document's text and diagrams into operational source code. The resultant software was used to run test procedures which had also been hand-written by engineers familiar with the protocol specification, to verify that the specification was met. In this project, we sought to determine the benefits of implementing some of the same functionality using a model-based approach instead.

The Proximity-1 Space Data Link Protocol [3] is an international CCSDS standard for relatively short-range communication, such as between rovers and orbiters at the same planet, or between multiple orbiters at the same planet. Since the range is short, the signal strength is not expected to be extremely weak and time delays are fairly short. Since communication may often be blocked by the planet, the protocol expects relatively short, independent communication sessions. Hence, automatic communication establishment (hailing) is an important part of the protocol, as is the ability to adjust parameters (such as data rate) as the distance changes between the spacecraft during a session. We focused on both of these primary capabilities in our study.

The concept of a state machine is commonly used in system specifications because it naturally captures the idea of system state and specifies how state should change as a result of different events. However, there is no standard convention for how these diagrams are produced. A state transition diagram that captures the full-duplex operational behavior is included in the Proximity-1 specification (Figure 2). We distinguish between this kind of informal diagrams and statecharts, which have a well defined syntax and consistent notation and can therefore provide the basis for an automated translation from design (diagram) to code (implementation).

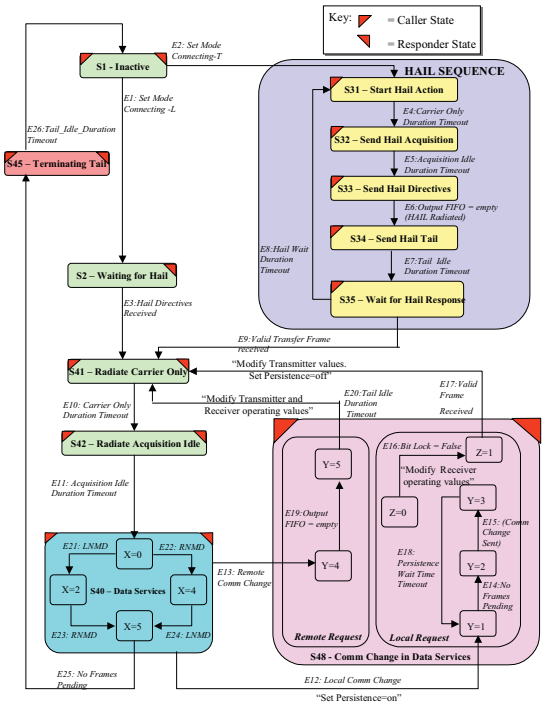

Figure 2. Figure 6-1 from the Proximity-1 specification, showing full-duplex operations [3]. The Hailing statechart in Figure 3 covers the top half of this figure, and the block in the lower right is captured by the Communications Change statechart in Figure 4.

4. RESULTS

4.1. Proximity-1 Hailing and CommChange Statecharts

We converted two key elements of the Proximity-1 Data Link Sublayer specification [3] into statecharts. First, we modeled the hailing process, obtaining the statechart shown in Figure 3. At any time, the system can be in one of the distinct substates within the larger "Hailing" state. When the radio is not attempting to hail, it is in `S01_Inactive`[3]. From that state, local directives can trigger `SetTransmit` or `SetListen` events, which cause the system to transition to `S31_StartHail` (caller) or `S02_WaitForHail` (responder), respectively. If the radio is in `S31_StartHail`, it takes actions to initiate a link with another radio, and it tracks the number of hailing attempts it has made (using `COUNTER.hail_lifetime_count`). Actions such as `mp_hail_setup_carrier()` are low-level behaviors not modeled explicitly by the statechart; instead, they are "device-driver" methods already implemented in the Electra codebase that the statechart leverages.

[3]The numbers prepended to the state names refer to state numbers found in the specification [3] for easy reference.

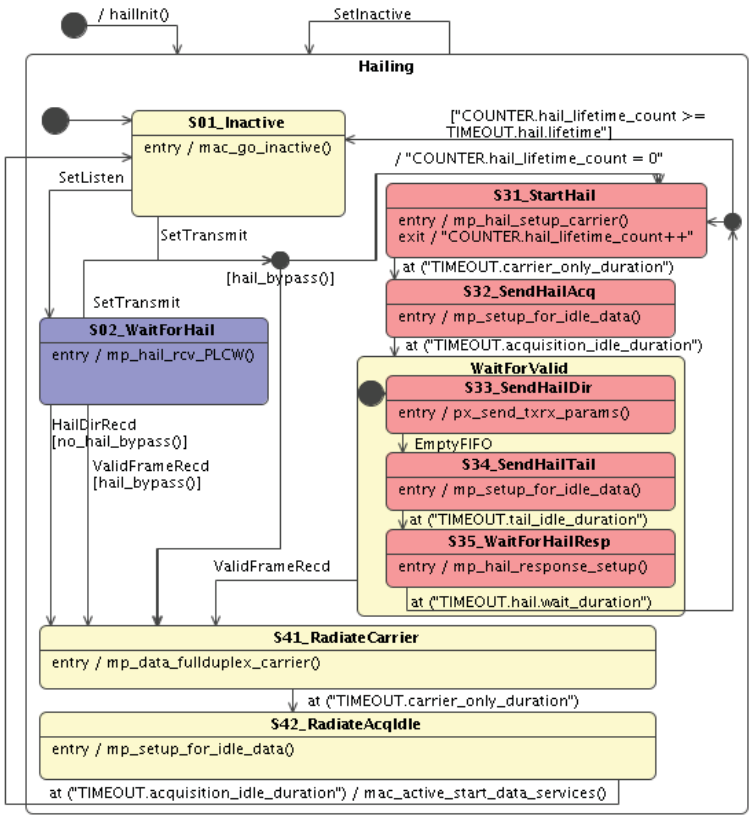

Figure 3. Proximity-1 Hailing statechart, composed of 11 states, 20 transitions, and 10 actions.

If the radio is in `S02_WaitFor Hail`, it does nothing until it receives a set of hail directives from another radio (`HailDirRecd` event) or, if the `hail_bypass` parameter is true, it receives a valid frame of any sort (`ValidFrameRecd`). The latter allows for Proximity-1 Simplex mode, where a radio either transmits or receives (but not both), and no hail directives are used. It also allows for a mode of operation where the Proximity-1 protocol is not actually used at all (raw data communication), which is not part of the protocol but which was a requirement of the Electra radio software.

Note the strong resemblance between this statechart and the informal diagram shown in Figure 2. Although the existing paper-based diagram provided a good starting point, converting it into an operational statechart exposed some limitations and incompletely specified behavior. For example, the Proximity-1 specification indicates that the caller should only respond to valid frames received from the responder when the caller is in state `S35_WaitFor HailResponse`. During testing, we developed a more robust version of the protocol that permitted the caller to respond to any valid frame received from the responder even if it was in `S34_SendHailTail` or `S33_Send HailDir`. Enclosing all three states in a single larger state, and specifying the `ValidFrameRecd` event as a trigger, was a simple update using this design process.

The second statechart that we created models the system behavior when a change in the communications parameters (frequency, bit rate) is requested by either radio in the link (see Figure 4). A communications change can be requested by the local radio (generating a `LocalComm Change` event) or received as a request from the remote radio (a `RemoteCommChange` event).

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. It will do this a maximum of `TIMEOUT.commchange.lifetime` times before giving up. 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.

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`

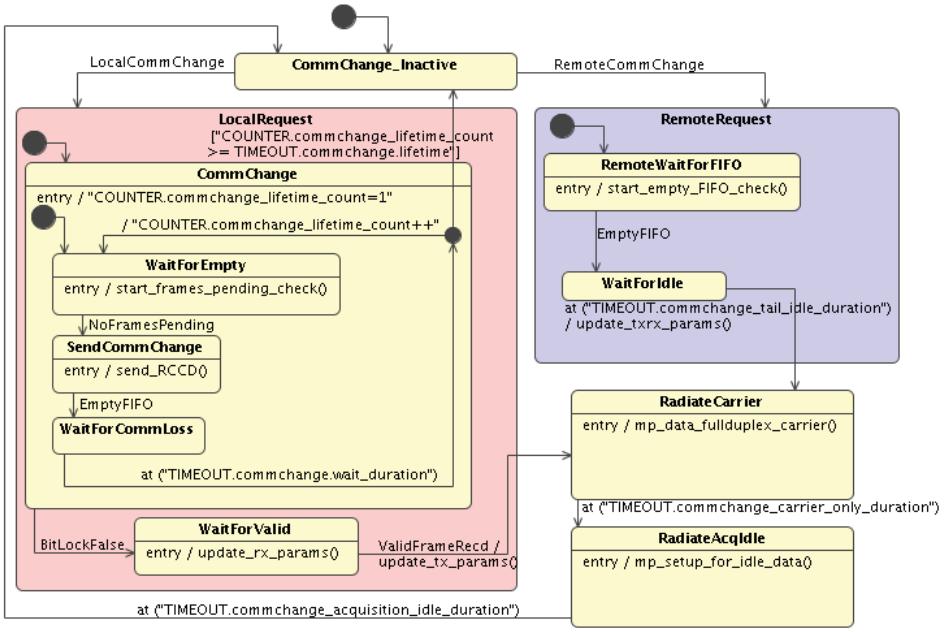

Figure 4. Proximity-1 Communications Change statechart, composed of 12 states, 17 transitions, and 8 actions.

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.

In this case, the statechart deviates to a larger extent from the diagram shown in Figure 2. This is due to the fact that the Figure 2 diagram less closely matches regular statechart semantics. For example, there is no defined way to reach state "Z=0", and no way to exit from the "Y=1,2,3" loop. The overall semantics and functionality of the protocol have been preserved and formalized in the statechart of Figure 4.

These diagrams went through several iterations before converging to the final versions shown in Figures 3 and 4. The ability to work in an efficient model–simulate–update loop permitted us to quickly reach the final diagrams.

4.2. Integration with MRO Flight Software

Once the diagrams were complete, we used the JPL Autocoder to produce C code implementing their functionality. Each state becomes a distinct method that takes in a pointer to a structure representing the current state of the system and a pointer to the next event that occurs. Figure 5 shows an example of the code generated for the `S01_Inactive` state from the Hailing statechart (Figure 3). The main body of the method is composed of a switch statement that determines what will happen next, based on the contents of the next event (`e->sig`). Two of the possible events are pre-defined events in the Quantum Framework: `Q_ENTRY_SIG` and `Q_EXIT_SIG` which correspond to entry and exit events for that state. The other two events defined are `SetListen` and `SetTransmit`, as defined in the discussion of Figure 3. For these events, the `Q_TRAN` macro from the Quantum Framework permits the method to signal that the system should transition from the `Inactive` state to a new state.

The JPL Autocoder converted each statechart into a .c file that contains one method to model each state, as well as a .h header file containing the necessary method declarations. Table 1 shows the source lines of code generated for each statechart; the counts were generated by David A. Wheeler's open-source 'SLOCCount' program[4]. In general, the auto-generated code tends to be longer than the original manual implementation, although exact quantitative comparisons are difficult since the original implementation of hailing also covered some functionality not modeled in our statechart (such as half-duplex communication), and there was no existing implementation of the communications change functionality to which we could compare.

The auto-generated code is longer because it includes references to the Quantum Framework and automatically logs each transition (e.g., `LogEvent_log(stateName)` statements in Figure 5). However, since it is all generated automatically, the increase in length does not require additional effort. In fact, it further enhances the reliability of the code, since there is no possibility for a logging statement to be accidentally omitted during implementation; everything is included. There is also an option provided by the Autocoder that suppresses the logging instructions, to improve the runtime efficiency of the code.

[4] http://www.dwheeler.com/sloccount

```
QSTATE Prox1_Inactive(
      Prox1 *me, QEvent const *e) {
  char stateName[256];
  strcpy(stateName, me->objName);
  strcat(stateName, " Inactive");
  switch (e->sig) {
    case Q_ENTRY_SIG:
      me->mystate = PROX1_INACTIVE;
      strcat(stateName, " ENTRY");
      LogEvent_log(stateName);
      Prox1Impl_mac_go_inactive(me->impl);
      return 0;
    case Q_EXIT_SIG:
      strcat(stateName, " EXIT");
      LogEvent_log(stateName);
      return 0;
    case SetListen:
      strcat(stateName, " SetListen");
      LogEvent_log(stateName);
      Q_TRAN(Prox1_WaitForHail);
      return 0;
    case SetTransmit:
      strcat(stateName, " SetTransmit");
      LogEvent_log(stateName);
      if (Prox1Impl_hail_bypass(me->impl)) {
        Q_TRAN(Prox1_RadiateCarrier);
      }
      else {
        Q_TRAN(Prox1_StartHail);
      }
      return 0;
  }
  return (QSTATE)QHsm_top;
}
```

Figure 5. C code automatically generated for the state `S01_Inactive` *in the Hailing statechart (Figure 3).*

Once the Autocoder had produced the C implementation of the statecharts, we packaged it up as a library and integrated it with the regular Electra flight software. Each place in the flight software that previously modeled hailing or communications change functionality was replaced with a call into our library. In general, original hand-coded switch statements (often one or two dozen lines of code) were replaced by one-line function calls to generate events for the autocoded state machine. This promises to make the code much more maintainable, since adjusting the state machine is easier and less error-prone in the statechart than in hand-coded switch statements scattered throughout the software. In fact, a great deal of the original development effort involved making sure that the proper states appeared in all necessary switch statements and no improper ones. It is usually comparatively simple to determine when to generate an event for the autocoded state machine, since the events (if well designed) map into natural occurrences in the system.

We also had to integrate the Quantum Framework into our system, so it could process the events and state transitions. The activities we modeled here had large time constants (tenths of seconds), so we used a simple but slow method of creating a new task that would periodically wake up and process the events. Our future implementation of the complete Proximity-1 protocol will require a tighter and faster integration with the real-time operating system, but the Quantum Framework is designed for use in real-time environments and supports more efficient integration methods [6].

Table 1. Source Lines of Code produced by the JPL Autocoder for each statechart.

Statechart	SLOC .c	SLOC .h
Hailing	324	43
Communications Change	310	42

4.3. MRO Flight Software Tests

We tested the integrated flight software, which contained our autogenerated Proximity-1 hailing and communications change library, on the MRO hardware testbed. There was an existing set of flight software tests developed for testing the Proximity-1 behavior of the Electra radios, and we applied each of the relevant tests to our version of the flight software. Table 2 shows the result of each of the five tests we performed; all of the tests passed. This result demonstrated that the automatically generated code correctly provided the desired communications functionality.

As noted in Section 4.1, during the course of these tests we discovered that, due to the timing behavior of the actual hardware, we needed to modify the Hailing statechart to permit the receipt of a valid frame to complete the hailing handshake even if the system was not yet in state `S35_WaitForHailResp`. This was the only update we found as a result of the tests. The majority of defects or design problems had already been identified during reviews of the statechart itself and by running the statechart in simulation. That is, 93% of the 15 tracked defects were detected, and corrected, prior to running on the hardware.

Table 2. MRO flight software test results using autogenerated C code based on the Proximity-1 hailing and communications change statecharts.

Test Name	Pass/Fail	Date
Prox-1 Basic Comm.	P	5/18/07
Prox-1 Hailing Process	P	6/4/07
Prox-1 Simplex	P	6/7/07
Relay Raw Data Comm.	P	6/7/07
Communications Change	P	4/22/07

4.4. Impact on Missions

One goal of this project was to identify ways in which model-based software development could be used by current and future mission flight software projects. Three such avenues bore fruit: the use of autocoding in future MSL flight software development, the use of the statechart simulator to communicate and identify design differences, and a follow-on project in which we will design statecharts for a full Proximity-1 reference implementation.

The MSL mission has supported model-based design for some time. In the course of this project, MSL flight software developers began using automatic code generation in several software modules. To date, they are using a simplified version of the JPL Autocoder that does not depend on the Quantum Framework. This version is called SMAC (State Machine Auto-Coder).

As mentioned in Section 2.2, one of the three types of output that the JPL Autocoder provides is a Python GUI that permits interactive testing of the model and generated code. During a Preliminary Design Review of the MSL rover landing radar behavior, this Python simulator was used to catch, and later resolve, divergent design interpretations by designers and implementers. By identifying the discrepancy early on, the mission saved money and time that would have been needed to detect, and correct, an error later once the software was implemented.

Finally, given the successful demonstration with the hailing and communications change parts of the Proximity-1 protocol, we are now proceeding with a full reference implementation of Proximity-1 that will also model the data exchange part of the protocol, half-duplex communications, etc. This reference implementation will be used in the JPL Protocol Testlab as a standard against which current and future radios that implement the protocol can be tested.

5. CONCLUSIONS AND LESSONS LEARNED

The use of model-based design and automatic code generation has proven to be a useful tool for the generation of instrument flight software. We used this approach to software development to replace and extend parts of an existing implementation of the Proximity-1 protocol for the Electra UHF radios used by MRO and MSL. We found that this development process provided several benefits over more traditional software development that are particularly important for critical flight software.

First, the design and the implementation are tightly coupled, so the documentation (design) is always up to date with respect to the code. The implementation has no opportunity to drift away from the original design. Second, since the JPL Autocoder produces C/C++, Promela, and Python output, additional tools can immediately be brought to bear on the design: the SPIN model checker can operate on the Promela representation of the system, and a Python simulator can "run" the statechart and permit immediate testing of the behavior produced as a result of a specific sequence of events. A large number of defects can be identified and corrected using these tools, before the C/C++ code is generated and run on the target hardware. Third, updates and corrections to the design can be easily and quickly accomplished; the developer need only modify the diagram and regenerate the code. There is no need to hunt for affected regions in the code and manually update each one. Ultimately, these benefits can result in reduced development and testing time and a reduction in software development and maintenance costs.

An important consideration when using model-based design is to determine what aspects of the system should be explicitly modeled (as states and transitions), as opposed to having their functionality embedded in a low-level action (e.g., `mp_hail_setup_carrier()` in Figure 3). Too much detail in a single statechart may obscure the overall functionality, while too little detail (e.g., modeling only an "active" and "inactive" state and pushing all detail down into actions) may render the statechart contentless. In this case, there is a good distinction between abstract protocol and concrete hardware, so we chose to model only the high-level protocol behavior, using actions to call out to actions that are specific to the underlying hardware on which the protocol is running. Not all systems provide such a natural separation. However, this is an important separation to make even in more conventional approaches to software development, so a model-based approach can be helpful in requiring that designers make that decision.

There are some additional challenges involved in adopting a model-based development approach. Developing code in this fashion raises questions about how Quality Assurance (QA) procedures should be updated as a result. Code reviews can be made more efficient when an accompanying statechart is available, especially one that can be examined dynamically (in simulation). In addition, after thorough review and validation of the automatic code generation tool, review efforts for an individual project can be focused on the designs (statecharts) rather than the implementations.

ACKNOWLEDGMENTS

We thank Harald Schone, David Brinza, Abdullah Aljabri, and Charles Norton for their continued support for this work. We also thank Steve Allen for discussions about Proximity-1 communications change for Electra. This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. It was funded by an internal R&TD grant at the Jet Propulsion Laboratory.

REFERENCES

1. Rouquette, N. F., Neilson, T., and Chen, G. The 13th technology of Deep Space One. In *Proceedings of the 1999 IEEE Aerospace Conference*, pages 477–487, 1999.

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

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

4. Marr, J. C. Space Interferometry Mission (SIM): Overview and current status. In Shao, M., editor, *Proceedings of the SPIE*, volume 485, pages 1–15, 2003.

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

6. Samek, M. *Practical Statecharts in C/C++*. CMP Books, 2002.

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