

Model Checking Investigations for Fault Protection System Validation

K. J. Barltrop
JPL

barltrop@jpl.nasa.gov

P. J. Pingree
JPL

ppingree@jpl.nasa.gov

Abstract

Following the trend of creating flight code with state chart modeling and auto-code generation, NASA's Deep Impact (DI) project uses Stateflow® for its Fault Protection Flight Software development. In a parallel task, model checking, a powerful and formal verification technique, is being used to partially validate the DI fault protection responses against a set of core behavioral correctness properties. The HiVy Toolset described in this work enables model checking for state chart-based designs by providing a formal method-based capability for automated state chart translation from Stateflow® into Promela, the input language of the Spin model checker. The results of this approach are compared against a traditional test script-based validation using the Matlab® environment. Additional work is in progress to validate the responses in the context of a complete flight system model with its accompanying mission-derived correctness properties. Special attention is given to methods for reducing the model to a form that yields a tractable search space.

1.0 Introduction

Proper design validation, which seeks to ensure the correctness of a design at the earliest stage possible, is a major challenge in any responsible software development process. Over the years, the complexity of space missions has dramatically increased with more of the critical aspects of a spacecraft's design being implemented in software. Fault Protection (FP) is autonomous flight software (FSW) that provides the robustness and autonomy needed to ensure survival of a space mission in the event of detected on-board failures. The design and validation of FP software systems is complex and its functionality in flight is critical, thereby making it a good candidate for more rigorous design and verification methods.

1.1 Deep Impact Mission Overview

The Deep Impact (DI) mission will send a pair of mated spacecraft to encounter the comet Tempel-1 in July of 2005. One day before closest approach, the single-string Impactor spacecraft separates from the Flyby spacecraft to collide with the comet, excavating a football field-sized crater. Meanwhile, the dual-string Flyby spacecraft records the resulting ejecta trajectory and spectra to provide insight into the internal structure of the comet.

The project has fault tolerance design requirements that lie somewhere between a lower-cost experimental mission (DS-1) and a higher cost flagship mission (Galileo). Furthermore, these design requirements must be met using a very small team for both design and test. JPL shall deliver all fault protection flight code to Ball Aerospace.

To help meet these requirements, Deep Impact has turned to formal state machine representation coupled with automatic code generation. Auto-coding directly ties generation and testing of the flight code back to system design documents. By using this approach the DI team has been able to automatically create flight code and test scripts, and perform interface compliance checking and automated test analysis.

1.2 DI Fault Protection Design Process

The DI auto-coding design originates in the Pathfinder mission to Mars. The core fault recovery engine design from Pathfinder was adapted to support the Deep Space-1 project as an experiment in automatic code generation. The DS-1 team diagrammed fault detection and recovery logic with Mathworks' Stateflow tool and then used a custom-modified version of Stateflow's autocoding capabilities to produce flight code.

The Deep Impact project reworked the fault recovery engine as a generic module that could be reused across multiple platforms. It also replaced the customized version of the Stateflow tool with Mathworks' improved off-the-shelf version. Further levels of auto-coding and

checking were added to ensure requirements compliance and to facilitate automated testing.

Although a similar process was applied for both fault detection and fault recovery development, the focus here will be on the recovery aspect. The steps followed in the fault recovery design development were:

1. *System Analysis*
2. *Design and Initial Verification*
3. *Implementation and Verification*
4. *Integration and Verification*
5. *System Validation*

The Matlab state chart development environment has allowed Deep Impact to complete rigorous design debugging early in the design process. The result of applying this method for fault *detection* so far has been that all verification (but not necessarily validation) issues were identified and corrected prior to system integration, and all but one or two have been fixed prior to unit testing on the target platform. Although still in progress for the fault *recovery* design, similar results are expected.

2.0 Illustrative Design Case

A look at a representative example will help illustrate the process and structure for the recovery behavior design.

2.1 System Analysis

Analysis of failure mode symptoms at the interface between the device and the system was performed via interviews with the design engineers. The information was captured in a form similar to that of classical failure modes analysis.

A fault tree analysis was then used to derive a system dependency model as illustrated in Table 1. An *element* can be a function, hardware item, or state. The column labeled *Req't* describes the dependency relationships while the *indicator* and *input* columns identify the system data needed to estimate the state of the parent element. This data includes information such as whether an element is actively used in supporting the spacecraft (e.g. member of the prime set).

<i>Flight System Elements</i>	<i>Req't</i>	<i>State Indicator Mnemonics</i>		
Parent element mnemonic	needs X	FaultAlarm	PrimeState	HeathState
Child element #1 mnemonic				
Child element #2 mnemonic				
Child element #3 mnemonic				
...				
Child element #n mnemonic				

Table 1. Element Model Structure

Corrective actions for failure modes were also collected during interviews with design engineers and during system level discussions of recovery. Attention was given to constraints on the conditions under which

actions may be taken. Table 2 illustrates a corrective action requirements analysis.

<i>Purpose</i>	The Device Repair Response shall recover device functionality after a fault has been detected.
<i>Location</i>	* Prime Computer * Backup Computer
<i>Tiers of action</i>	1. Reset 1553 RT 2. If not at encounter cycle/reload device electronics 3. Swap to backup electronics 4. Exhaust
<i>Interfering</i>	No
<i>Comments</i>	none

Table 2. Corrective Actions and Constraints Analysis

Eventually this information was combined with the dependency model to create a tier description table as shown in Table 3. The tier description table becomes the source design document for verifying that recovery behaviors comply with many aspects of design intent.

Chart ID	Table Size				FP Sequence Calls				
	Fyby	Element	Prime	Avail	Sync	Reset	Cycle	Escape	Isolate
					Self Avail	Next Avail			
Behavior #1	4	Bus	Bus	Bus	1
Behavior #2	4	Cbh	Cbh	Cbh	1	2	.	.	.
Behavior #3	2	RfCmd	Sdst	Sdst	1	.	.	2	.
...	3	BatCell	BatCell	BatCell	1
Behavior #4	2	Gimb	Gimb	Gimb	1	2	3	.	4

Table 3. Tier Description Table

Each sequence call is classified according to the type of action it performs. For DI five actions categories were defined. Choosing the correct category is a matter of engineering judgment, but the choice of category has a real impact on later assessment of valid behavior.

Sync = re-command state to desired state;

Reset = perform hard reset of element;

Cycle = perform power cycle of element;

Escape = go to alternate configuration without this element;

Isolate = go to alternate configuration and make sure this element is physically or electrically isolated.

Each action also specifies some requirement on whether the resource in question or its backup needs to be marked as available by ground ops.

2.2 Design & Initial Verification

Detailed design of recovery behaviors is eventually captured in the state chart drawing. Figure 1 illustrates a typical layout for a response's active state. Within the active state the response may be in one of several sub-states. All but the last sub-state usually corresponds to the execution of a sequence of spacecraft commands. In terms of software behavior, the last sub-state initiates an exit from the response function. In a system sense the last state represents a period during which a triggering fault is given time to re-trigger if the problem has not actually been fixed. After a timeout period, the response is set to its idle state (not shown).

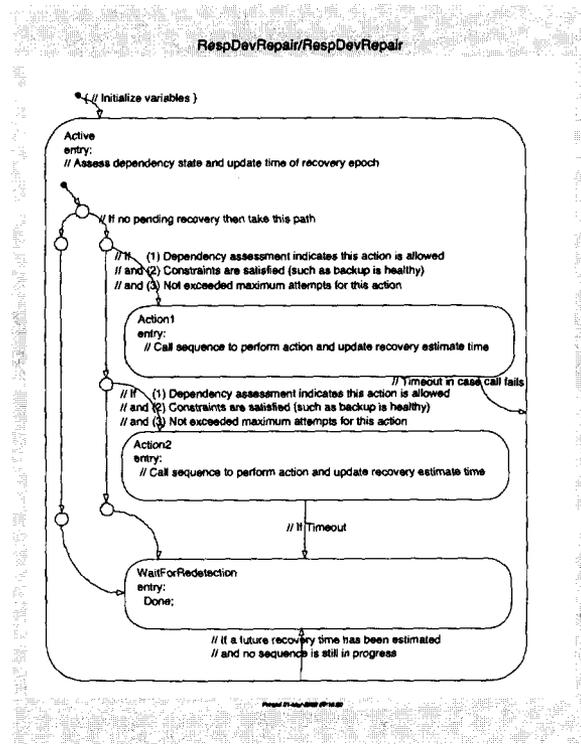


Figure 1. Illustration of State Chart

Spacecraft state information along with local data items are captured in a data dictionary for the state chart as shown in Table 4. This data dictionary is crosschecked against the information in the tier table. For example, the *InputDevPrime* and *InputDevAvailable* entries must appear as entries in the tier table.

Initial design testing begins with instrumenting the state chart to interface with Matlab scripts, and with the creation of scripts to exercise the chart. A test generation tool uses the data dictionary along with several look-up tables to construct a script that traverses the data space for the chart. Data items are classified into several groups:

Masks – discrete states that can inhibit access to a chart transition.

Events – temporal data items that can inhibit access to a

chart transition.

Selects – data items that are used to compute an index into a table.

Parameters – data items that we can safely assume will maintain a single value for the mission.

Local Data – data items that are only used locally by the chart for computation support.

Data Name	Scope	Type	Size
Id	Local	uint8	
IsSubResp	Local	boolean	
SysTime	Local	double	
SysTimeInput	Local	double	
Urgency	Local	uint8	
EnabledAction1	Param	boolean	3
EnabledAction2	Param	boolean	3
IdDev	Param	uint8	
MaxRetryAction1	Param	uint8	
MaxRetryAction2	Param	uint8	
TimeoutAction1	Param	uint16	
TimeoutAction2	Param	uint16	
Epoch	Persist	double	
CalledAsSubResponse	Shared	boolean	
FpSeqActive	Shared	boolean	
InputDevAvailable	Shared	boolean	4
InputDevPrime	Shared	uint8	4
MaxDev	Shared	uint8	
MinDev	Shared	uint8	
SeqIdAction1	Shared	uint16	4
SeqIdAction2	Shared	uint16	4
TimeOfLastRun	Shared	double	256
TimeOfRecoveryDev	Shared	double	4
ValueDev	Shared	uint8	

Table 4. Data Dictionary Illustration

The generated test script contains tables describing this information about the data space. The tables are then referenced as part of a nested loop structure to test combinations of masks, events, and selects. Figure 2 shows an illustration of the script layout.

Behavioral analysis is tabulated by checking the behavior against a set of rules such as “No response shall execute a recovery action for an element if a prior recovery on that element is already in progress.”

Testing of simple responses on a 400 MHz PC require about 15 minutes to complete. Testing of complex safety-net responses requires about 36 hours to complete. The output of this testing is a behavior summary report that lists assertions about behavior that can be verified for rule compliance.

```

Data Description Tables
Parameter Initialization
Select Item Loop
  Select Item Value Loop
    Mask Loop
      Event Loop
        Time Loop
          Response Call
          end
        Behavior Tabulation
      end
    end
  end
end
Behavior Analysis Report

```

Figure 2. Matlab Test Script Layout

Stateflow's animated debugging environment made it very easy to track down unexpected behaviors. The insight gained by watching the execution led to a few significant chart layout redesigns.

To illustrate a typical design defect caught by the test approach, a state chart has been modified to exclude a required enable check on a transition. The analysis report for the unmodified chart produces the excerpt:

- *GyrEnabledEscape(1) inhibited nothing*
- *GyrEnabledEscape(2) inhibited nothing*
- *GyrEnabledEscape(3) inhibited nothing*

The analysis report for the modified chart produces:

- *GyrEnabledEscape(1) inhibited*
SeqGyroEscape(1) 2 times when NoSelect = 0
- *GyrEnabledEscape(1) inhibited*
SeqGyroEscape(2) 2 times when GyrIdGyro(1) = 1
- *GyrEnabledEscape(1) inhibited*
SeqGyroEscape(3) 2 times when GyrIdGyro(1) = 2
- *GyrEnabledEscape(1) inhibited*
SeqGyroEscape(4) 2 times when GyrIdGyro(1) = 3
- *GyrEnabledEscape(2) inhibited*
SeqGyroEscape(1) 2 times when ValueGyro(1) = 1
- *GyrEnabledEscape(2) inhibited*
SeqGyroEscape(1) 2 times when ValueGyro(1) = 2
- *GyrEnabledEscape(3) inhibited nothing*

The two excerpts show differences in the assertions about the affect of changing the *GyrEnabledEscape* variable. In this case the desired behavior is that changing the value of the variable should inhibit execution of the recovery sequence for the respective urgency value of 0 through 2. In the modified version the recovery sequence is never inhibited, thus violating a requirement that the sequence only be executed when enabled.

2.3 Implementation & Verification

Once Matlab testing is completed, the charts move to the auto-coding phase. Stateflow's off-the-shelf code generator provides the initial "C" language version of the response behavior. Additional post-processing tools reorganize the "C" code into a "C++" class equivalent that is compatible with the flight system coding standards and architecture.

The generated code is linked with a unit test harness that provides a subset of the overall flight software services and a TCL interface. Additional tools then create a TCL version of the original Matlab test script. Running the TCL scripts provides an identical summary report to the Matlab versions. This allows quick comparison of auto-generated code behavior against the Matlab-tested behavior.

2.4 Integration & Verification

After completion of unit testing, the fault protection software is sent to the contractor for integration with the flight software. Integration is performed on a software test bed, which provides engineering versions of the core avionics fed by reasonably high fidelity simulations of the flight system. Test scripting executes in real time under the Oasis [Oasis-CC] test environment.

At this point it becomes impractical to apply the same scope of testing that was used at the unit level. The Oasis test scripts are designed to cover only the unit test cases that show an expectation of a behavior change. For example, if changing a particular disable in unit test demonstrated that it always inhibited a particular tier then the Oasis case would skip over event tests for that tier whenever the disable is applied. This is expected to still provide code statement coverage, but not the extensive data space coverage of the unit tests.

2.5 System Validation

During system validation the project seeks to demonstrate that under mission-like conditions the design gives results consistent with mission level objectives. "Pass" criteria are less clear-cut than in verification and often rely on judgment calls about the appropriateness of the behavior. For example, recovery in a particular scenario eventually occurs, but one might observe that it needlessly changed the state of many flight elements in the process.

System validation follows the traditional approach of previous missions. Significant mission scenarios are collected and analyzed for state changes. In this case we do make use of the flight system model to identify such points. Fault injections are then applied at critical transition points (before, during, after) and the system behavior recorded to confirm that everything "looks OK." Although more formal metrics may be derived to assess

the quality of a recovery, the DI project has not scoped such an effort.

3.0 Model Checking Investigations

3.1 Beyond Scripted Test Cases

Deep Impact believes that the above verification and validation process is very thorough with respect to the standards of typical deep space missions. However, in the absolute sense it by no means completely tests all of the states and paths. Although it would be possible to upgrade existing test scripts to explore all combinations of states and inputs, it is time-prohibitive within the DI test environments. To complete testing within a manageable amount of time, the auto-generated test scripts take short cuts such as exploring the range of only the first four and last two elements of any data array. They also have a baseline state definition of which at most three or four off-nominal states are ever simultaneously explored.

Such an approach provides useful information to the extent that one believes that all unexplored combinations are sufficiently equivalent to explored ones. Under ideal circumstances we would identify a basic set of state cases and know that all unexplored cases are valid combinations of the explored cases. For example, engineering analysis concludes that the system will always behave identically when either star tracker A or tracker B is providing healthy data, so having confirmed that A recovery and B recovery behave identically, we may use tracker A states as a sufficient representation of all tracker states. By finding system elements with external interactions that are invariant with respect to their own internal states one can reduce the search space. In the tracker example, by observing that the attitude estimator element interacts with the star tracker element without a specific requirement for A or B tracker, one can reduce the search space requirement to vary only tracker A states in conjunction with varying other system states.

In practice it becomes prohibitively expensive to accurately identify these invariance scenarios for a real system. It's quite easy to find invariance with respect to two elements, but upon considering the wider picture, find a lack of invariance with respect to other elements. In the star tracker example, although attitude estimation may not care whether it gets its valid data from tracker A or tracker B, the thermal system may care that having tracker A on tends to cause a warming trend that tracker B does not. When one gets to system level validation, where real-time testing of complex scenarios becomes extremely time consuming, one is continually forced to make a best estimate about where uncoupled behavior can be assumed.

It is a goal, however, to actually perform this sort of comprehensive analysis and testing, thereby eliminating the need to make best guesses at what constitutes a sufficiently representative set of test cases. By encoding the design of the flight system using an environment that allows rapid exploration of state changes, such as model checking, one can perform a more exhaustive search of the system behavior. Engineering decisions about where to reduce the search space no longer limit the verification and validation. Instead the fidelity of the flight system physics and software behavior representation, and the raw faster-than-real-time computing power available become the limiting factors.

3.2. The HiVy Toolset

The HiVy toolset provides model checking for state charts ([SFUG]). This is achieved by translating state chart specifications into the input language of the Spin model checker ([Hol97]). The HiVy toolset transforms output of the commercial tool Stateflow provided by The Mathworks. HiVy can also be used independently from Stateflow. An abstract syntax of hierarchical sequential automata (HSA) is provided as an intermediate format for the toolset [Mik02]. The HiVy toolset programs include *SfParse*, *sf2hsa*, *hsa2pr* and the HSA merge facility.

State chart design representations are captured in Stateflow model files. Two programs of the HiVy toolbox: *SfParse* and *sf2hsa* are used to prepare the model file for translation. If parsing is successful, a file is produced that contains an ASCII representation of the abstract syntax tree in HSA-format.

Once the components of the system are parsed in HSA HiVy generates Promela input for the Spin model checker. As an interim step, if the model consists of several files, then they may be merged into one HSA file before translating into Promela for Spin using the HSA merge facility program *hsacomplete*. To preserve Stateflow scope when merging a subcharted state chart with its parent, all state names are extended by the name of the root state of the subchart. This resolves potential name clashes in the merged state chart. It is important to know these naming conventions because during verification, the user is provided with propositions that refer to renamed states. These propositions are the means for formalizing linear temporal logic (LTL) properties about the state chart model for Spin.

The program *hsa2pr* is used to generate Promela code from the HSA file. The following files are generated by *hsa2pr*:

- *stmodel.pr*: the Promela model of the original state chart.
- *propositions*: contains names and definitions of propositions. One proposition is generated for each state and each event.

- `prop_list`: contains just the names of propositions (not their definitions). These proposition names are suitable for automatic generation of LTL properties during verification.

The auto-translated file `stmodel.pr` contains an include statement for a file named `never`. This file contains the Spin “never claim” to be verified. The never claim is not generated by `hsa2pr` and must be created before applying Spin to the generated model.

3.3 Creating a Valid Promela Model

HiVy implements the new HSA format that accurately represents Stateflow semantics. The translation method associates with each state chart a hierarchical sequential automaton that is semantically equivalent to the source state chart. A hierarchical sequential automaton consists of a finite set of cooperating sequential automata that can be implemented as parallel processes in Promela. Referring to Figure 3, we establish equivalence between the semantics of the source state chart, intermediate HSA and the resulting Promela code.

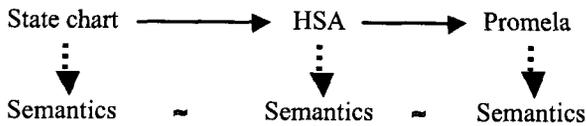


Figure 3. Two step translation of State charts

Within the auto-translated code there is one Promela process for each Stateflow OR-state. This Promela process corresponds to one automaton in HSA. State chart states, events and variables are encoded as Promela variables and Promela processes change the values of these variables in order to simulate state changes, event generation and variable changes according to the semantics of Stateflow. The observable behavior is defined with respect to the variables representing state chart states, events and variables.

In order to achieve a closed model for the code generated from the FP response state charts, we manually extend the automatically generated code by user-written Promela code, which will model the response environment. Both the model of the environment and its integration with the generated code must yield a closed system that is a valid model of the real system.

We contend that the clockwise execution order of the Stateflow semantics might give rise to design faults that are hard to detect because the semantics determine one specific execution order to be taken based on the graphical representation of transitions. We compensate for this semantic restriction by expanding the translation of HSA to Promela such that every execution order is considered when a property of interest is verified. Stated more formally, this change is a generalization of the original

semantics to consider not only the original execution order but alternative execution orders as well.

In the integration of user written Promela code with translated code a closed system is created. The translated code interface allows the user-written code to pass control and data to the translated code. Whenever translated code is executed the user-written code waits for this execution to be completed and vice versa. Here we adopt the ideas of the synchrony hypothesis: the controller program (here represented by the translated FP response specification) reacts infinitely faster than the environment and therefore the environment can be considered as “waiting” for the controller to complete. This construction is valid only if the controller program is responsive; i.e. eventually reacts to each environment input. [PMHSD02] discusses how we verify responsiveness in our translated models.

The coding conventions for integration of translated code with user-written Promela code are the following:

- The user-written Promela code may set values of event, boolean and integer variables.
- The user-written Promela code may pass control to the translated code `execute_root` by setting the activation condition of this function. Similarly, after activation the user-written Promela code waits for the corresponding notification condition to be satisfied.

Implemented in this way the flow of control between the user-written Promela code and the translated code guarantees mutual exclusiveness in their execution.

Reactive systems are characterized by infinite interaction with the environment, e.g., in the case of state charts by receiving events from the environment and by responding to them. Since the FP responses designed in Stateflow are playing the role of the controller in a reactive system our translation and integration method ensures that the Stateflow model indeed interacts with its environment infinitely. We call a control program *responsive* if every execution passes the state where the notification condition is satisfied infinitely often. This property can be expressed as a formal Linear Temporal Logic (LTL) property and can be verified using Spin.

3.4 Model Checking with Spin

Model checking is a powerful formal methods approach which can detect defects in designs that are typically difficult to discover with conventional testing approaches. The Spin model checker accepts input of a closed-loop model of the validation article of interest and specification of LTL correctness properties against which the model can be validated.

The DI state charts implement two categories of temporal constraints by comparing recorded time tags of certain events. Some chart transitions act on whether temporal relationships are satisfied. For example, it is determined that recovery is in progress by comparing the estimated

time of conclusion of a recovery with the current time. Of primary importance to the chart behavior is whether that comparison indicates one of two states: (1) A prior recovery in progress OR (2) No prior recovery in progress. Because Spin model checking does not support a concept of time per se, this chart implementation presented a problem for conversion. To get around this, the charts were pre-processed to replace each time comparison with an equivalent single boolean state indicating which of the two possible comparison results were to be tested. For example, the expression *TimeOfRecoveryX < SystemTime* would be translated to the single boolean *TimeOfRecoveryXLessThanSysTime*.

Next we applied the following correctness property (CP) to the closed-loop Promela model of the Deep Impact device response previously described:

```
!([](Urgency=WantIt || Urgency=NeedIt) || RunFPSeg=False)
```

This Spin “never claim” seeks to verify the following DI FP requirement where “[]” in the property means that it is always the case that:

“No repair response shall attempt recovery actions for an element unless the corresponding urgency has been assessed as either ‘Need It’ or ‘Want It’.”

We can induce a bug by manipulating the variables in the model pertaining to those in the property. However, the Spin results verifying that this property holds in the correctly designed response are:

```
----- start -----
!([](Urgency=WantIt || Urgency=NeedIt) || RunFPSeg=False)
--Building the verifier
--Verification
(Spin Version 4.0.0 -- 9 May 2001 -- Lucent Proprietary Code)
Hash-Compact 4 search for:
  never-claim      +
  assertion violations + (if within scope of claim)
  acceptance cycles + (fairness disabled)
  invalid endstates - (disabled by never-claim)
State-vector 120 byte, depth reached 0, errors: 0
  1 states, stored
  0 states, matched
  1 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
5.060 memory usage (Mbyte)
```

```
real    0.1
user    0.0
sys     0.0
```

```
----- compare results: expected - actual -----
State-vector 120 byte, depth reached 76, errors: 0
```

In the event of an error (i.e., a failed correctness property), Spin produces a counterexample file that documents the property conflict with the Promela model of the system.

We are in the process of building a library of correctness properties for this code, from which we plan to derive formal Spin “never claims” that can then be verified by the model checker. The final system should allow us to check the Stateflow specifications within their intended context with a thoroughness that is virtually impossible to achieve by other means.

Once the design is validated we will also investigate how to correlate the validated design (a conservative abstraction of the real system) to the real implementation, i.e., are all errors of the design reproducible in the model checking implementation? We will report on the system design and validation of its implementation in future work.

4.0 Summary

The HiVy Toolset permits the validation of complex mission critical software designs with the exhaustive exploration techniques of model checking. The HiVy translation method preserves the Stateflow semantics of the software design to guarantee direct association of the auto-translated Promela model for validation with the flight software code that is auto-generated from the state chart specification using Stateflow Coder. Spin’s capability to express correctness properties in explicit LTL statements provides a clear connection between design requirements and verification artifacts, removing the need to invent some of the verification mechanisms such as Deep Impact has employed. When the state chart is the source of both the flight code and the Promela model, this automated approach ensures design and validation integrity of the implemented code.

5.0 Acknowledgments

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.

The authors also acknowledge E. Benowitz, D. Dams, M. Feather, E. Mikk, A. Oyake, J. Powell & M. Smith for their contributions to the results presented in this paper.

6.0 References

- [Hol97] G.J. Holzmann. The Model Checker Spin. IEEE Trans. on Software Engineering, 23(5):279-295, May 1997. Special issue on Formal Methods in Software Practice.
- [Mik02] E. Mikk. HSA-Format, *private communication* 2002.
- [Oasis-CC] Operations and Science Instrument Support – Command and Control, <http://lasp.colorado.edu/oasis/oasis.html>
- [PMHSD02] P. Pingree, E. Mikk, G. Holzmann, M. Smith, D. Dams, Validation of Mission Critical Software Design and Implementation Using Model Checking. The 21st Digital Avionics Systems Conference, October 2002
- [SFUG] The Mathworks Stateflow Users Guide, <http://www.mathworks.com>