A model-based Architecture for a small flexible Fault Protection System

Garth Watney
California Institute of Technology, Jet Propulsion Laboratory, 4800 Oak Grove Dr. Pasadena, CA 91109

Introduction

Space mission flight software typically relies on a Fault Protection software system to monitor, detect, diagnose and mitigate anomalous events. Due to the long communication lag for deep space missions, the spacecraft has to operate with enough autonomy to detect faults and respond appropriately. The traditional fault protection implementation has utilized a distributed detection and centralized mitigation strategy. (see figure 1). Distributed monitors reside in various subsystems and detect faults that are either handled with a local fault recovery or a system-level fault recovery. A “small” fault protection system rather than a “large” system has been the focus of this paper for a couple of reasons. Firstly, the software infrastructure to support this proposed architecture is small – utilizing a few hundred lines of framework code as the underlying communication protocol and state-machine model implementation. The auto-coding tools described here are small home-grown light-weight tools that perform simple mappings from a modeling language syntax to design patterns in C/C++. The second reason a “small” system was specified is because larger systems in the future may utilize a completely different software architectural paradigm that is goal based. In JPL’s Mission Data Systems (Ref. 6), fault protection in an integral part of the design and not a separate subsystem as it is treated in this paper. This paper describes the more traditional fault protection architectures and proposes a small flexible architecture that fits within this context. The centralized Fault Protection Manager is responsible to diagnose the fault and conduct the appropriate system-level response. Each mission may choose a unique fault protection strategy. The Cassini AACS Fault Protection supported the execution of parallel recovery responses. Other missions like the DS1 (Deep-Space One) and Deep-Impact supported sequential interruptible fault responses based on a priority scheme. Each mission has utilized unique software architectures to implement the specific details of their required fault protection behavior. The software required to implement fault protection behaviors has been fairly complex and error prone, requiring extensive testing before deployment. Correctly translating system requirements (specified as “shall” statements in a requirements document) into correct flight code is a difficult and error-prone process.

In this paper we will give a brief overview of how different missions implemented the fault protection application and the improvements along the way. We will then propose an architecture that supports the direct implementation of statechart models into flight code. These statechart models can be used to formally and naturally specify the behavior of all the major fault protection components – monitors, fault protection engine, and fault responses. The goal is a flexible, light-weight implementation of a traditional fault protection software system that is inexpensive to implement, reliable and understandable for both system and software developers.

Figure 1: Local Detection, Centralized Mitigation Fault Protection (diagram by Dan Dvorak)
Fault Protection Architectures for Space Missions

Cassini AACS (Attitude and Articulation Control Subsystem)

The Cassini AACS Fault Protection system was implemented without the use of software implementation models. All the Fault Protection components were manually coded using the Ada Programming language. Distributed monitors throughout the system would generate symptom events which fed into a series of Diagnostic Rules. These rules would generate fault events which in-turn would activate the Responses based upon a series of Activation Rules. This implementation would allow multiple responses to be active in parallel. Responses could be prioritized and time-sliced. This heavy-weight rule-based system finally worked very well, but not without extensive testing and many “brain storming” sessions to flush out a working set of rules. The advantages of this approach are the following:

1. Final product was a well-tested and extremely capable fault protection system capable of responding to all conceivable anomalies.

The disadvantages of this approach are the following:

1. Extremely high implementation cost
2. Final product is a “pile of code” that is not easily analyzable or understandable across disciplines
3. Entire Fault Protection software was tightly coupled to the mission and was not malleable to change or reuse across missions.

Deep Space One (DS1)

The DS1 Fault Protection implemented a much simpler software system than Cassini. Responses were not allowed to run in parallel, but were queued up waiting for their turn to execute. A two tiered system was implemented that allowed shorter (more urgent) responses to interrupt longer (less urgent) running responses. Distributed monitors throughout the system would also generate symptom events. Symptoms would be mapped to Faults which in-turn would be mapped to Responses. This was a many to one mapping which enabled the fault to be isolated and a targeted response to be activated. A software component called the Fault Protection Engine (implemented manually in C) would be responsible for this mapping and the orderly execution of responses. The innovation in the DS1 approach was the use of software implementation models to auto-generate C code. (Ref 4). All Monitors and Responses were developed by system engineers using the Mathworks Stateflow tool. This tool allowed a system engineer to specify the behavior with Statechart models. Stateflow would also auto-generate C code from these models. A suite of home-grown post processing tools would transform the auto-generated C code into flight C code by applying various software wrappers and other transformations of the code. The advantages of this approach are the following:

1. One system engineer could develop Monitor and Response models which could be faithfully translated into flight code resulting in an enormous savings of software development and testing.
2. Statechart Models were used to capture Monitors and Responses which can be understandable across disciplines.

The disadvantages of this approach were the following:

1. The Statechart Models did not conform to the UML (Ref 1) Statechart semantics but had their own Stateflow proprietary semantics.
2. The Stateflow tool saved the models in a proprietary format which restricted the ability for us to parse and analyze
3. The Stateflow tool generated C code that was not easily understandable and did not conform to the projects coding standards or design patterns.
4. The Fault Protection Engine software component was tightly coupled with the DS1 mission and so was not malleable to change or reusable across missions.

Deep-Impact

The Deep-Impact Fault Protection system was based on the legacy design of DS1. The same fault protection strategy of queuing responses was used. Matlab Stateflow models were also used to specify Monitors and Responses. The flight code however was implemented in C++. Once again a suite of post processing tools transformed the auto-generated C code into flight C++ code. An additional requirement for Deep-Impact was the reusability of the fault protection system for different missions. The Fault
Protection (FP) Engine component was then refactored to be mission agnostic. Monitors and Responses would register to the FP Engine and a system/response mapping grid was uploaded to the software. The advantages of this approach were similar to DS1 but added:

1. A refactored reusable FP Engine software component that was mission agnostic.

The disadvantages of this approach were similar to DS1 with respect to the Stateflow tool but added:

1. The FP Engine software component was still just “a pile of code”. Even though it was designed for reusability across missions, the code was still fairly complex and not malleable to changes in the fault protection strategy. Code, no matter how well written, is still not easily analyzable or understandable across disciplines.

Space-based Interferometry Mission (SIM)

For SIM (Ref. 7), a model-based software architecture was developed which directly supports model execution. The SIM software architecture can be viewed as a middle model-based framework layer on top of the lower RTOS layer. This middle layer directly supports the application layer which can be described as a collection of interacting state-machines. (see figure 2). All dynamic behaviors from high level flight mode controllers to the low level device drivers are expressed in UML Statechart models. Because SIM adopted a model-based software implementation process, the formally specified state-machine is at the heart of every software component capturing the dynamic behavior. One of the innovations in the SIM software development was the use of non COTS tools to auto-generate flight code from UML Statecharts.

(Ref. 2). Statechart models are expressed using a UML drawing tool (i.e. MagicDraw) and the model saved in an XML file. The home-grown autocoder reads this XML model and maps the UML Statechart syntax directly to tried and tested software design patterns that are readable and conform to flight software coding standards. Advances in UML drawing tools which save the model in a non proprietary format allow the model to be read and analyzed. The SIM flight software architecture directly supports the instantiation, execution and communication between the models using a Publish and Subscribe protocol (often described as a software bus). To implement the Fault Protection software system, all aspects of the system – Monitors, Responses and the Fault Protection Engine are expressed as a collection of interacting UML Statechart models. The fault protection strategy which is encapsulated in the FP Engine model could conform to the Cassini strategy, the DS1/Deep-Impact strategy or some other variant strategy. The advantages to this approach were the following:

1. All FP software components were explicitly modeled and flight code generated directly from these models
2. The model-based software development process was not tied to a vendor’s autocoding or drawing tools.
3. The auto-generated code was readable and understandable since the UML Statechart syntax was directly mapped to tried and tested software design patterns
4. The fault protection strategy was explicitly captured as a model enabling the specific strategy to be malleable to changes.

The disadvantages to this approach were the following:

1. Abandoning the previous Matlab Stateflow tools for generating and autocoding Statechart models reduces the ability of system engineers to be directly involved in building software implementation models. In the SIM process, system engineers would be involved in reviewing the models, but typically would not generate the models themselves.
2. Scalability problems begin to emerge with multiple monitors and responses hanging off the software bus (see figure 3). This problem is discussed in more detail in the following sections.

Implementing State-chart models using the Quantum Framework

The Quantum Framework (QF) is a small light-weight software framework for implementing UML Statecharts semantics into C/C++ code. It is primarily intended for real-time embedded applications. The Quantum Framework was adopted to elegantly map UML Statechart semantics to flight software design patterns for the SIM flight software. The Quantum Framework was developed by Miro Samek and is fully described in his book (Ref. 8). The entire framework consists of approximately 800 lines of code and comes with both C and C++ implementations. The underlying architecture consists of Active objects. Active objects are hierarchical state machines that communicate with each other via an exchange of event instances. The framework is extremely flexible in that the user may decide to utilize just the base class for hierarchical state machines, or Active objects with the Quantum framework’s Publish and Subscribe mechanism for event communication. The framework also comes in different flavors – a simple one thread or multithreading environment. When developing code based on this framework, the developer thinks at a higher level - in terms of states, transitions and events, not convoluted if-then-else or switch statements. We found the architecture of the Quantum Framework to be very elegant, understandable and efficient. We experimented with different design patterns to implement some of the more complex UML state-machine features such as orthogonal regions, junctions and history states. A good Statechart Framework allows for an easy mapping of UML Statechart models into the application code.

The SIM Flight Software chose to utilize the Quantum Framework for its inter-process communication method (publish and subscribe) and for the implementation of all state-machines. As previously mentioned, SIM utilized a model-based software architecture where the system can be viewed as a collection of interacting state-machines. The executive scheduler for SIM was based on rate groups. Each state-machine in the system was assigned to a particular rate group and executed periodically. Published events are placed in the input queues of state-machines that subscribed to the particular event. Events are drained from the input queues and dispatched to the state-machine during its particular rate group.

As previously discussed, the Fault Protection system for SIM incorporated Monitors, Responses and a FP Engine software component which are all specified as formal state-machines and utilize the Quantum Framework’s Publish and Subscribe framework. (see figure 3). Scalability problems soon started to emerge with this approach. For a typical Fault Protection system, we need to implement anywhere from a few dozen to a few hundred state-machines. Every Monitor and every Response is captured as a state-machine. The SIM flight software adopted a rate-group driven architecture, as apposed to an event-driven architecture. This then required every state-machine to be assigned a particular rate group. The state-machine is required to be called periodically at its rate group in order to consume and process events on its input queue. The implications of this design decision mean that every state-machine has a certain amount of processing throughput overhead. State-machines communicate at a global level by publishing and subscribing to global events. Using this architecture would mean that every Response in the Fault Protection system would need to be active (executed periodically), even if there were no anomalies. Another problem emerges with the implementation of Monitors. Every Monitor is simply an instantiation from a common Monitor state-machine. (all monitors share a common design pattern where the monitor is
in either a Green, Yellow or Red discrete state). Essentially we have one Monitor state-machine that is instantiated multiple times and distributed throughout the system. Having multiple instantiations of any one state-machine poses a problem using the SIM Architecture. Every one of these instantiations will respond to the same event. How can events be directed to a particular instantiation and ignored by other instantiations? This problem was solved by adding guards to every state transition. The guard function checks an identification parameter in the published event. In this way, only one monitor state-machine would respond to a common event. The solution however is rather cumbersome in that extra processing is required to add and remove and check the event from every monitor’s state-machine input queue. In addition extra complexity is added to the model itself. We started to look for a more elegant solution to this problem.

Light-weight State-chart Architecture

Using the SIM Flight Software architecture directly for a Fault Protection system has some shortcomings as previously described – mainly in the area of performance throughput and managing multiple instantiations of a single state-machine. To make improvements to the SIM Fault Protection subsystem we continued to use the SIM State-based Architecture, but augmented it with a “light-weight” state-machine implementation. Light-weight state-machines are almost identical to the regular Quantum Framework state-machine design pattern, with the added capability to operate at the local level. In figure 3 we illustrate the fault protection components communicating via a global Publish/Subscribe software bus. We grouped state-machines together into subsystems that can communicate with each other at a local (subsystem) level. The subsystem component communication can be isolated from the rest of the system. Multiply-instantiated Monitors that are already distributed amongst subsystems can be prevented from all responding to a globally published event. Figure 4 illustrates this grouping of state-machines into subsystems. Localized state-machines have the following restrictions: 1) Local state-machines cannot subscribe to events and 2) local state-machines must be encapsulated by another global interface component (i.e. Fault Protection Manager). This global interface component is responsible for publishing events directly to its local state-machines via a point to point publish capability. This point to point publish capability is the only needed addition that was made to the Quantum Framework. The throughput performance problem is solved since the Fault Protection Manager tracks what responses are active and only invokes the active response/responses periodically. The Monitor state-machines which are instantiated throughout the application are also local (light-weight) state-machines. Because they operate at a local level, this solves the problem of multiple Monitor instantiations responding to the same event. This particular architecture is applicable not just for fault protection, but other subsystems as well. For instance – multiple instantiations of the same device driver. The scalability problem will always emerge as the number of state-machines
grow in the system. This solution of light-weight or localized groupings of state-machines is a way to manage that complexity.

**Advantages of Modeling**

Our experiences in explicitly modeling software components and architecting the system as described has revealed a number of advantages:

1. The FP Engine provides a blueprint template that can be customized for any future mission to accommodate mission-specific functionality
2. The FP Engine can also be extended into a general purpose Autonomy Engine that responds to non fault events
3. Models in general provide a common communication medium between system and software engineers. (Ref. 5)
4. Formal and explicit models reveal whether a software engineer has understood the intent of the imposed requirements.
5. Models help to nail down ambiguous or loosely worded requirements
6. Executable models can be used as a prototype to demonstrate and test early behavior
7. Models can be used to analyze and document the design
8. Models can be used to automatically generate large portions of the flight software code

Because the interaction of multiple Statechart models can produce unexpected behavior, especially in a multi-threaded environment, it is important to formally verify that these models are correct. The SPIN verifier (Ref. 2) is a tool for analyzing the logical consistency of concurrent systems. SPIN has its own model specification language called Promela. Just as we have mapped UML Statechart semantics into C/C++ code, so we have also mapped these same semantics into Promela code. We have started building verification models that check internal consistency in the design and prove that various requirements are always satisfied.

The advantages of utilizing our own autocoding tools to generate flight code from the UML Statecharts have been the following:

1. Increased efficiency for the software engineer
2. Increased maintainability of the software product (rapid turn-around of specification changes to builds)
3. Fewer defects are introduced in the auto-generated code
4. Auto-generated code always conforms to the specified UML semantics
5. Complete control of the tool suite. Not locked into a vendor’s modeling or autocoding tools
6. Complete control of the auto-generated output code. We can utilize our own design patterns and generated code that interfaces to our flight software architecture.

**Shortcomings**

When using this light-weight statechart architecture for implementing a Fault Protection system we have identified some general areas for improvement. Specifying dynamic behavior in a local software component level, statechart models are a natural and expressive means to efficiently design and auto-generate flight code. However, at the global level, component interaction and the executive real-time scheduling are not currently being modeled. Our experience has shown that most of our defects arise from component interfaces and other issues pertaining to the real-time scheduling of tasks. Very few defects arise from the dynamic behavior of a component that has been auto-coded from a statechart model – even when the statechart model was excessively complicated (deeply nested states with multiple orthogonal regions). Components can communicate with each other through the easy interface of Publish and Subscribe. This allows software developers to adopt the “don’t know and don’t care” attitude pertaining to the communication of events. (a component simply publishes an event without specifying the recipient of the event). Simple spelling errors in event names often lead to defects in the code where events are published without any recipients and recipients subscribe to events without any publishers. Other interface defects arise when a state-machine publishes events at a higher rate than the recipient due to state-machines running in different rate groups. A need arises to explicitly model the component communication and the
execution threads. An explicit formal model will allow us to automatically analyze the global interaction of software components and remove these defects at build time or before. An Architectural Description Language like AADL (Ref. 9) is being considered to describe a higher software architecture description of the software system.

**Conclusion**

In this paper we have given brief overviews of various flight Fault Protection implementations. At one extreme is the Cassini AACS which has a large and complicated fault protection system that was implemented without models and all the code manually generated. Our goal has been to create a software architecture for a small mission that is highly flexible in its particular fault protection philosophy (i.e. concurrently running responses, sequential responses, interruptible responses etc). In addition, we have desired to involve system engineers in the fault protection software development process. By capturing the explicit fault protection philosophy in an FP Engine state-machine model that is formally specified and auto-coded we have shown that the behavior is much more flexible to change. By capturing the Monitors and Responses as state-machine models, we have found a common medium by which system and software engineers can engage in reviews and design sessions together. We have also identified shortcomings in this approach where the global behavior is not being explicitly modeled which leads to defects in the communication between software components. This area can be fortified with the use of modeling the high level architecture using an architectural description language. In general, by focusing on implementation models to capture the fault protection strategy, monitoring and responses, we have greatly streamlined the design and implementation process leading to a light-weight fault protection software system that is more flexible and less prone to software defects.

**References**


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.