The Curiosity Mars Rover's Fault Protection Engine

Ed Benowitz
Jet Propulsion Laboratory, California Institute of Technology
La Cañada Flintridge, CA
eddieb@jpl.nasa.gov

***Abstract*—The Curiosity Rover, currently operating on Mars, contains flight software onboard to autonomously handle aspects of system fault protection. Over 1000 monitors and 39 responses are present in the flight software. Orchestrating these behaviors is the flight software's fault protection engine. In this paper, we discuss the engine's design, responsibilities, and present some lessons learned for future missions.**

Keywords—fault protection; MSL; Curiosity Rover; Mars Science Laboratory; flight software; mars rover

I. INTRODUCTION

The flight software on the Mars Science Laboratory mission (MSL) [1] is active during launch during, the cruise to Mars, during Mars entry, and while the Curiosity rover roams Mars. Throughout all of these phases, if a fault occurs, flight software must autonomously take appropriate action. The portion of flight software that checks for faults and decides what action to take in response is called the fault protection engine.

In this paper, we will discuss the MSL fault protection engine. We will begin by discussing prior work on similar missions. We then provide background overview of the MSL mission itself. We then discuss the interfaces between the fault protection engine and other parts of flight software, namely monitors and responses. We then delve deeper into the engine itself, discussing details on how the engine chooses what actions to take. Finally, we discuss some lessons learned and future directions.

II. PRIOR WORK

Deep Impact [2, 3] had a centralized fault protection engine. It mapped from symptoms to faults, and then from faults to responses. Its engine used a queue to ensure that faults are serviced in a first-come-first-serve basis. Only a single response could run at a time. Some responses were interruptible. After a response completes, during a designated recovery time, their engine is free to service another response triggered by a different monitor.

The fault protection engine on MER [4] provided a resource arbitration scheme in the event of a resource conflict between multiple fault responses.

An alternative architecture [5] suggests integrating systems fault protection through the entire system design, and not necessarily having a centralized fault protection engine. Functions should be preserved, not just protected. Fault protection should not be separated from the nominal operation of the same functions. The emphasis is on modeling and obtaining state knowledge, and then closing the loop around state.

This work is sponsored by the Jet Propulsion Laboratory (JPL), California Institute of Technology under a contract with the National Aeronautics and Space Administration (NASA)

III. MISSION AND FLIGHT SOFTWARE OVERVIEW

A. Mission Overview

The Curiosity rover has the goal of exploring the habitability of Mars [1]. The mission was launched on November 26, 2011, and landed on Mars on August 6, 2012. The mission goes through 3 main phases, each of which uses a subset of the hardware. Note that the fault protection behaviors needed for each phase are different.

During the cruise phase, the spacecraft travels from Earth to Mars. During cruise, trajectory correction maneuvers are performed, refining the trajectory. As the spacecraft nears Mars, the cruise stage hardware is then discarded.

Next is the Entry Descent and Landing (EDL) phase, which lands the rover on Mars. During this phase, a combination of guided entry, a parachute, powered descent, and finally a sky crane are used to deliver the rover safely to the Martian surface. By the time the rover has reached the ground, EDL hardware is discarded. System fault protection is disabled in this phase, as the EDL software fully is in control.

During the final phase, the surface phase, the Mars rover drives on the surface of Mars, performing scientific investigations with its suite of instruments. The rover carries ten instruments, a robotic arm, various cameras, and a drill.

Particularly relevant to fault protection, the MSL spacecraft has redundancy in a number of hardware areas. Specifically, it carries two flight computers, only one of which is prime at a time. As further examples of this redundancy, the spacecraft also carries dual sets of avionics, sensors, heaters, and has multiple telecommunications options.

B. Flight Software Overview

The MSL flight software runs on the VxWorks operating system. It is broken up into a number of modules, each of which runs in its own VxWorks task. Each module typically communicates with another module via IPC. IPC messages get put on a priority queue, and each priority can be individually enabled or disabled. As the queue is primary means of communication between tasks, semaphores are not typically

used. In most scenarios, a client will send an IPC message, wait for a reply IPC message, and then proceed.

IV. Fault Protection Architecture

We identify the key players that participate in the fault protection implemented by flight software. Monitors identify a persistent problem. The monitors may initiate a local response. A local response is code that resides solely within the same module as the monitor, and attempts to correct an issue by using only a small portion of the system.

The fault protection engine is responsible for checking for problems periodically, and for initiating a response if appropriate. The engine periodically polls the monitor states. When the engine notices that a monitor has turned red, the engine maps the monitor to a system response. The engine then initiates a system response. A system response is code that makes large, system level changes in an attempt to address a fault condition. A system response typically calls multiple software modules.

A. Monitors

A monitor is a portion of software that is responsible for detecting an anomalous situation. Monitors are spread throughout the flight software near the software that has knowledge of a particular device. On MSL, most monitors were implemented with a library that was used by many modules. MSL has a total of over 1000 monitors. However many of them share the same code, especially for thermal monitors.

A persistence count is typically used within a monitor, so that it takes multiple occurrences of the erroneous reading to provoke a response.

A monitor goes through a number of states which are visible in telemetry. The black state indicates we do not have any data yet. In the green state, we have received data indicating there is no error. In the yellow state we have received erroneous data for a long enough persistence that we need to run a local fault response. Note that a local response is handled within the module, and the fault protection engine is not involved. A local response can be enabled or disabled via ground command.

When the erroneous data reading persists even longer, the monitor turns red, indicating that the monitor needs to get system fault protection involved. A key provision here is that once a monitor gets to red, the monitor stays red until it resets. Even if good data arrives, the monitor stays red. When we describe how the engine behaves in a later section, we will explain why this behavior was implemented. This contrasts with the approach in [2], where monitors do not latch red.

Although rarely used if ever, monitors provide the capability to disable detection via a ground command. This setting can be stored in non-volatile memory. Monitors can also be temporarily disabled from flight software, but this change is only noted in RAM. We found this was necessary to prevent unintended monitors from tripping as a consequence of hardware being in an intermediate state during a response.

B. Responses

We now provide a description of system responses. All responses begin by stopping any ongoing autonomous behavior onboard. For example, this includes stopping any currently running onboard sequences. We allow responses to have multiple tiers, so that as a problem recurs, more severe actions will be taken in a response. Also, because many of the internal response actions are similar, it is quite common for responses to be implemented in terms of other responses. That is, a response may be implemented in part by calling a portion of another response as if it were a library.

A typical response will have a menu of options to take in various degrees of severity. Some examples of responses actions include changing the telecommunications configuration, swapping redundant devices, power cycling a device, or in an extreme case, switching to the other spacecraft main computer.

Most responses do not involve much branching behavior, and are typically implemented as a simple state machine that executes a step, waits for response back, and then executes the next step. A response tells the engine when it has finished.

MSL has a total of 39 responses. Further examples of typical responses are provided in [6].

V. Fault Protection Engine

We first provide a high level view of the fault protection engine behavior, and then proceed to look into the details.

- Once per second, check to see which monitors have just turned red.
- In a table, look up the responses associated with the newly red monitors
- Choose the highest priority response; this will be our new candidate response
- If no responses are currently running, start running the candidate response
- If a response is already running, but is lower priority than the candidate response, abort the currently running response, then start running the candidate response.
- When a response has finished, reset all monitors associated with the response.

A. Mapping from monitor to response

The engine maintains a hard-coded table, containing an entry that maps a single monitor to a single response. This array is indexed by monitor, and returns the appropriate response. Note that the table is not dependent on the spacecraft configuration, be it cruise or surface. So if a behavior depends on the spacecraft mode, the change in behavior must be done within the response, not the engine.

Within the table, it is possible to have multiple monitors that trigger the same response. But we require that a single monitor will only trigger one response.

B. Polling approach

Recall that we require that monitors stay red until the monitor is reset. It is this property of monitors that allows polling to work. If a monitor could independently move between red and green, a polling engine might miss a change if it happened too quickly. By having the monitors stick to red, polling is able to notice that a monitor has turned red.

There are some advantages to the polling approach. The monitor modules do not need to have knowledge of or maintain a handshake with the engine. The engine knows about monitors within the clients, but the clients do not have to know about the engine. Architecturally, we believe this is a positive, because it prevents a cyclical dependency between the engine and its clients. Additionally, we believe this approach simplifies the monitor implementation.

Let us contrast it with an alternative which sent an IPC fault announce message every time a monitor turned red. The client and the engine would have to coordinate to ensure that fault announce messages can never overflow the engine's IPC queue.

However, in fairness there are some disadvantages to the polling approach. Because of our polling approach, the engine does not handle the monitor that occurred first, chronologically. Instead, by polling, the engine acts on the first highest priority response that it examines. It may be more likely that the first monitor to go off chronologically was the root cause, and thus should be handled first. Our polling approach does not honor this. Also the polling approach would need modification for a system in which the engine modules could not access the memory of the monitor modules.

C. Priorities

Each response is assigned a hard-coded priority. Higher priorities are assigned to short-running responses that would need to run first due to the hardware architecture. For example, a response that fixes a bus would be higher priority than a response that fixes a device that sits on that bus.

We also note that there is an implicit, second level priority present. Consider the case where multiple responses have the same explicit priority. The implicit, second level priority turns out to be the order that the responses are examined within flight software. In other words, if there are two candidate responses of the same priority, the engine will execute the first response it examines. Thus, to predict which response will run given a list of candidate responses, the ground needs to know the explicit response priority levels and the implicit flight software response ordering.

D. Aborts

Suppose a lengthy low-priority response is already running, but the engine has detected a higher priority response which needs to run. The engine has the capability to abort a lower-priority response in the middle of its run, to allow a higher priority response to start. To implement this, the low-priority response receives an abort message from the engine. The response then waits for its current step to complete. When the step is complete, instead of starting a new step, the response tells the engine it is done aborting. Upon receiving the abort done message from a response, the engine is then free to start the high-priority response.

There was debate within the team about whether we should only abort at specific points within a response, or whether we should abort as soon as the current step completes. The advantage of aborting upon step completion is that we get to a higher priority response more quickly. The disadvantage, however is that it is not feasible to test the entire space of abort locations. By contrast, if we had limited the number of possible abort locations within a response, it might have been feasible to test them all.

E. Enables and disables

The engine provides ground commands to the operations team to limit the engine's activities. The ground team is allowed to prevent a particular monitor from tripping its response. Note that we do not allow the ground to disable a single response entirely. Instead it is the response associated with a single monitor that is disabled or enabled via the ground. The ground operator can be specific, enabling or disabling during only specific spacecraft configurations.

Additionally, certain responses need to dead-end. That is, responses are only permitted to run a given number of times, and after that point, the response must not run again. The engine was modified to allow a response to dead-end itself. The dead-ending can be undone with a ground command.

F. Response cleanup

After a response has finished running, control returns to the engine. The engine then uses a table to determine which monitors are associated with the response. The engine then clears all of these monitors. The monitors reset their persistence values, and return the black state. At this point, monitors are no longer stuck in the red state. After the monitors are reset, they are free to redetect, and will eventually turn red again if the problem persists.

G. Design for Testability

A number of features were added to the engine to assist with testing. Testers wanted the ability to start a response directly on command without waiting for a monitor turn red. In essence, this was used to test the response, not the engine itself. So the engine added a utility command to directly execute a single response. Additionally, testers wanted a way to exercise and test the engine, without being dependent on a particular monitor implementation. To facilitate this, the engine added a second utility command. This command forces the engine to detect that a particular monitor turned red. It bypasses the actual state of the monitor. These two commands allowed testing the responses and engine in isolation. So when it came time to test the behavior of monitors, the engine, and responses in end-to-end scenarios, we had more confidence that the individual pieces would work.

H. Important Telemetry

During a fault scenario, it is typical to switch to a very low data rate. Because of this we must choose carefully what telemetry to send, to help the ground diagnose the current situation. Some of telemetry MSL selected for this situation comes from the fault protection engine.

MSL has two main types of telemetry: periodic channelized telemetry channels, and prioritized event reports (EVRs). The fault protection engine is responsible for tracking and storing the last n fault monitors to turn red. It also tracks and stores the last n responses to run. This information is stored persistently in a special memory that is shared between both flight computers. So on boot and whenever this information changes, the engine pushes the last n monitors and responses out to a telemetry channel. In addition, the engine contains a number of high priority EVRs that indicate the last n monitors and last n responses, and get generated whenever one of the lists change. We also maintain counts for the total number of monitors that turned red, and for the total number of responses that have run over the course of the mission.

VI. Lessons Learned and Future Directions

A. Future directions

We believe the MSL fault protection engine could be used in other missions. The primary areas that would change to the engine code proper would be hardcoded mapping tables. These tables are isolated, and contain information that lets the engine map monitors to responses. The tables also let the engine know what monitors to reset when a response has finished.

The other main side effect of using the engine in another mission would be the way the engine impacts the new mission's monitor designs. A new mission would need to design monitors that latch red. Additionally, the new mission would need to have all monitors implement a uniform API for the engine to use for polling and resetting the monitors. Additionally, a new mission would have to design responses such that they could be aborted.

B. Lessons Learned

Several changes were made to the engine later in development, and serve as lessons for future missions. Several responses had requirements to dead-end. That is, the responses would execute a finite number of times, and afterwards, they would not be allowed to execute again. To facilitate this, the engine was modified to track a response's dead-end state in non-volatile memory.

Additionally, we found during response testing that several responses ending up causing additional monitors to trip in the middle of response execution. Additional engine capabilities were added, allowing a response to tell the engine to temporarily ignore certain monitors during a small portion of response execution.

As further advice for future fault protection designers, we suggest they carefully plan the fault protection design for times when the spacecraft goes through a major reconfiguration change. The caution is noted because the software that performs a reconfiguration may conflict with or attempt to run simultaneously or at cross purposes with the system fault protection. Consider the protocols and restrictions on interactions between the fault protection engine and other services that reconfigure the system. Make sure to plan for scenarios with a fault during a configuration change, or a scenario with a configuration change during an already running fault response.

VII. Conclusions

We have discussed our polling approach in MSL's fault protection engine. We've showed how information flows from a monitor to the engine, and showed how the engine prioritizes and then runs responses. We have also showed our strategy for aborting responses. We presented engine changes that were discovered later in the development process, and discussed several other lessons learned that are applicable to future projects. We feel that with a change to hard-coded mapping tables, the engine could be re-used in future missions.

Acknowledgment

We would like to thank the entire MSL flight software team, led by Ben Cichy, for their contributions. We recognize the outstanding designs and leadership contributions of fault protection systems engineer Tracy Neilson. We thank the systems fault protection test team headed by Mary Lam.

References

[1] J. Grotzinger, J. Crisp, A. Vasavada, R. Anderson, C. Baker, R. Barry, D. Blake et al. "Mars Science Laboratory mission and science investigation," Space science reviews 170, no. 1-4, 2012, pp. 5-56.

[2] K. Barltrop, E. Kan, "How much fault protection is enough – A deep impact perspective," Proceedings of the 2005 IEEE Aerospace Conference, 2005, pp. 1-14.

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

[4] T. Neilson, "Mars exploration rovers surface fault protection", 2005 IEEE Conference on Systems, Man and Cybernetics, Vol. 1, pp. 14-19, 2005.

[5] R. Rasmussen, "GN&C fault protection fundamentals," Proceedings of the 31st Annual AAS Guidance and Control Conference, AAS 08-031, 2008.

[6] P. Morgan, "Fault protection techniques in JPL Spacecraft," Proceedings of the First International Forum on Integrated System Health Engineering and Management in Aerospace (ISHEM), 2005.

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