Survey of NASA Mars Exploration Missions – Software Fault Protection Architecture

Mars Climate Orbiter, Mars Reconnaissance Orbiter, and Mars Exploration Rovers

© Dennis Clark, Christopher Brennan, and Megan Jaunich
Software Reliability and Safety Engineering (SSW 689), Spring 2010 Stevens Institute of Technology
5/3/2010
# Table of Contents

Introduction.......................................................................................................................... 4

General Comments on NASA Spacecraft System Safety Evaluations .................................. 5

Overview of Attitude Control System, Orbit Determination Methodology, and Considerations of MCO Fault Protection Architecture ................................................................. 10

Orbit Determination Process .................................................................................................. 11

Angular Momentum Desaturation Files .................................................................................. 11

Categorization of Fault Protection Capabilities ...................................................................... 12

Human Management ............................................................................................................. 12

Fault Avoidance .................................................................................................................. 12

Fault Tolerance ................................................................................................................... 13

Failure Tolerance/Management ............................................................................................ 13

Accident Avoidance ............................................................................................................. 14

Testing of Fault Protection Capability .................................................................................. 14

Effectiveness of the Fault Protection Architecture ................................................................. 15

Mars Reconnaissance Orbiter ............................................................................................... 16

Fault Protection Architecture ............................................................................................. 17

Categorization of Fault Protection Capabilities ..................................................................... 20

Human Management ........................................................................................................... 20

Fault Tolerance .................................................................................................................. 20

Failure Tolerance /Management .......................................................................................... 20

Accident Avoidance ............................................................................................................ 21

Testing of Fault Protection Capability ................................................................................. 21
Effectiveness of the Fault Protection Architecture .......................................................... 21
Mars Exploration Rovers .................................................................................................... 23
Fault Protection Architecture .......................................................................................... 23
Categorize the fault protection capability ...................................................................... 25
  Human management ........................................................................................................ 25
  Fault avoidance ............................................................................................................... 25
Fault Tolerance .................................................................................................................. 26
Failure Tolerance/Management ......................................................................................... 26
Accident Avoidance .......................................................................................................... 26
Testing of Fault Protection Capability ............................................................................. 26
Effectiveness of the Fault Protection Architecture ....................................................... 27
Conclusions ....................................................................................................................... 28
**Introduction**

Today’s complex systems are increasingly software-driven, with software controlling safety-critical functionality or responsible for monitoring the state of safety-critical systems. In modern space systems, much of the functionality of a spacecraft or satellite is automated, making software dependability an important factor in the overall system safety and mission success of a spacecraft system.

An updated software dependability model (Laird 2010) identifies the attributes of software dependability as reliability, availability, robustness, safety, security, and resilience. Threats to dependability are faults, failures, hazards, and external systems (including humans). The means of achieving dependability are through avoidance, tolerance, and/or management of the dependability threats. This paper will discuss software fault protection architecture and its capabilities and effectiveness in “stopping the flow” from latent faults, to failures in operation, to hazardous conditions, to accidents. The Accident Process Model (APM, Figure 1) (Laird 2010) is utilized in considering the fault protection architecture and categorizing the fault protection capabilities. Fault avoidance, fault tolerance, failure management, and accident avoidance are each deemed “Chokepoints” and each is an opportunity to utilize software to mitigate or prevent a mishap or accident. Laird calls this software reliability and safety architectural solution as “Stop-the-Flow Framework.”

In particular, this paper will apply these concepts to three NASA Mars missions – the Mars Climate Orbiter (MCO), the Mars Reconnaissance Orbiter (MRO), and the Mars Exploration Rovers (MER). NASA and Air Force system safety design requirements are introduced and the typical NASA system hazard analysis process is described in order to provide context. Failure case studies illustrate specific examples of how the “Stop-the-Flow-Framework” may have been useful in preventing a costly mishap. Lastly, the Goal, Question, Metric (GQM) approach is utilized for each Mars spacecraft/mission in order to evaluate the fault protection architecture (Basili 1994). This is an approach that considers the conceptual level, or goal, the operational level, or question, and the quantitative level, or metric. It is a structured way to define the goals of a system, and compare them against the operation of the system in a measurable way.
General Comments on NASA Spacecraft System Safety Evaluations

Note: Much of this section is based on the experience of M. Jaunich, who works as a system safety engineer evaluating spacecraft system design, including safety critical functionality controlled by software, for NASA’s Launch Services Program at Kennedy Space Center, FL. This is a spacecraft system-level design evaluation approach and considers the software as-written in the context of the entire system.

In general, there are two key requirements documents with which all NASA satellites must comply – one is NPR 8715.3 NASA General Safety Program Requirements, and the other is the safety documentation of the launch site or range from which the spacecraft will be launched (of course, there are many, many more standards and policies with which compliance is mandatory or required). In the case of all three missions discussed here, the applicable launch site safety document is Eastern and Western Range Safety Requirements (EWR 127-1). EWR 127-1 is being phased out and replaced with a very similar document, Air Force Space Command Manual Range Safety User Requirements (AFSPCMAN 91-710), which will be referenced in this study. These documents outline, among many other topics, system safety requirements for spacecraft design, with particular attention to safety-critical systems. This includes software-intensive spacecraft subsystems with safety-critical functionality. To provide a common diction for the discussion that follows, terminology used in these requirements documents is explained below.

The following definitions come from NPR 8715.3:

- Inhibit – Design feature that prevents operation of a function.
• Failure – Inability of a system, subsystem, component, or part to perform its required function within specified limits.

• Failure tolerance – the ability of a system to perform its function(s) or maintain control of a hazard in the presence of failures of its subsystems. Failure tolerance may be accomplished through like or unlike redundancy.

The following definitions come from AFSPCMAN 91-710:

• Failure – the inability of a system or system component to perform a required function within specified limits.

• Fault – the manifestation of an error in software that may cause a failure.

• Fault tolerance – the built-in ability of a system to provide continued correct operation in the presence of a specified number of faults or failures

• Inhibit – an independent and verifiable mechanical and/or electrical device that prevents a hazardous event from occurring; device has direct control and is not the monitor of such a device.

We will utilize the following definitions for consistency:

• Fault – the manifestation of an error in software that may cause a failure.

• Failure – Inability of a system, subsystem, component, or part to perform its required function within specified limits.

• Fault tolerance – the built-in ability of a system to provide continued correct operation in the presence of a specified number of faults or failures

• Failure tolerance/management – the ability of a system to perform its function(s) or maintain control of a hazard in the presence of failures of its subsystems. Failure tolerance may be accomplished through like or unlike redundancy.

• Inhibit – an independent and verifiable mechanical and/or electrical device that prevents a hazardous event from occurring; device has direct control and is not the monitor of such a device.
Both NPR 8715.3 and AFSPCMAN 91-710 have similar requirements for number of inhibits and failure tolerance.

Utilizing the more stringent requirement between NPR 8715.3 and AFSPCMAN 91-710, the number of inhibits required is:

- Operations have three inhibits where loss of life or loss of system can occur.
- Operations have two inhibits where personal injury, illness, mission loss, or system loss or damage can occur.

Utilizing the more stringent requirement between NPR 8715.3 and AFSPCMAN 91-710, the number failure tolerance/management requirements are as follows:

- If a system failure may lead to a catastrophic hazard (loss of life or system loss), the system must be designed such that the combination of two failures and/or operator errors (fail-safe, fail-safe as a minimum) will not result in loss of life (dual failure tolerant).
- If a system failure may lead to a critical hazard (system loss/damage or personal injury), the system must be designed such that the occurrence of a single failure or operator errors (fail-safe) will not result in personal injury or system loss/damage (single failure tolerant).

Strictly speaking based on these definitions and requirements, software is not considered to be an inhibit to any hazardous conditions in the system design. In other words, even though well-designed software may be utilized in order to prevent a hazardous condition (for example, may be used to monitor safety critical circuits and automatically change the state of that circuit if it inadvertently transitions into an incorrect state), it is not considered sufficient to be categorized as an inhibit. On the other hand, failure of software is considered when assessing the failure tolerance of a spacecraft system or any of its subsystems. So, if a single software failure could lead to a critical hazard (system loss/damage or personal injury), there must be at least one other failure required prior to that hazard being present. This failure could be of a mechanical component, electrical circuit, or process (i.e. human error).

Whether software should be considered as an inhibit in system safety design is a philosophical discussion which will not soon be resolved, as many system safety professionals are not familiar...
enough with software to understand how dependable software (which includes aspects of reliability, safety, availability, resilience, robustness, and security) (Laird 2010) can contribute to the overall emergent safety of the system. However, it is well understood in the NASA system safety community that software is critical to system safety and mission success (CITATION). 
NPR 7150.2A (2009) NASA Software Engineering Requirements
Laird states that “hazard analysis is rarely applied to software systems.” Based on the experience of this author in evaluating spacecraft systems hazard analyses, this is true. From the spacecraft system level, software is not explicitly assessed using the HA technique as a hazard is typically directly attributed to those components/subsystems which physically manifest the hazard. Hazard analyses performed on a spacecraft are typically carried out as follows:

- List hazards and categorize based on the potential severity and likelihood of occurrence (AFSPCMAN 91-710, Figure 3.2).
- Per the hazard classification, perform a detailed hazard analysis on those in the highest third of hazard classification.
- Identify the subsystem that presents the hazard and provide a description.
- List the hazard causes – this is one place where we see software indirectly assessed in the hazard analysis process. Hazard causes could be latent faults that lead to a failure in operation. For the purpose of a qualitative hazard analysis, often a hazard cause may be simply stated as “software failure” and not be developed further.
- List the hazard controls – these can include software controls and this is another place where software is indirectly assessed in the hazard analysis process. Hazard controls could incorporate the “chokepoints” suggested by Laird (Figure 1). At the stage when the spacecraft system hazard analyses are performed, the system architecture may be set such that incorporating fault avoidance may not be an option. However fault tolerance, failure management, and accident avoidance are all techniques that can be incorporated to some extent whether in the software architecture or in the physical one.
- List the method of verifying the hazard control has been implemented and the status of the verification.
It is evident from this description that at the system level, software may be considered as a way to control hazards and perhaps as a potential failure point in the functionality of safety- or mission-critical software. From the system level, it is almost assumed that software will work since it does not succumb to the wear and tear of traditional hardware models, and perhaps also because many engineers do not have a good grasp of what “safety,” “reliability,” and “dependability” mean for software. Therefore, the onus is on the software development community, along with the system engineers, to ensure that a thorough independent verification and validation program be implemented on safety/mission-critical software prior to integrating software with the fully assembled system. Stakeholder needs must be flowed into system requirements, which must be implemented through design and testing. Appropriate input and output ranges must be specified in order to identify equivalence classes which will test the normal operation of the system as well as the boundaries of normal inputs and outputs. Expected outputs must be compared against as-run output data based on a known/given input set in order to validate algorithms/functionality of the system.

Mars Climate Orbiter

The Mars Climate Orbiter (MCO) was part of the Mars Surveyor ’98 program which also included the Mars Polar Lander. Both spacecraft were launched separately on Delta II rockets from pads 17A and 17B of Florida’s Cape Canaveral Air Force Station (CCAFS) (NASA Press Kit, 1998) during the Mars orbit insertion launch window, which only occurs approximately every two years (26 months). The broad objective of the Mars Surveyor program was to study the climate of Mars and to gain insight into the weather, presence of water and carbon dioxide, and the behavior and role of volatiles (chemical elements/compounds typically found in the crust or atmosphere of an orbital body that have low boiling points at the atmospheric pressure, e.g. nitrogen, hydrogen). This information could be used to ascertain the occurrence and nature of climate changes on the planet (Mars Climate Orbiter Fact Sheet, 1998).

MCO was managed by NASA’s Jet Propulsion Laboratory, and Lockheed Martin was the primary contractor responsible for design, manufacturing, and integration operations. Unfortunately, MCO was lost after its nine month trek to the red planet, most likely crashing into Mars after approaching too close to the surface. The NASA Mishap Investigation Board Phase I Report (1999) cited the root cause as “Failure to use metric units in the coding of a ground
software file, ‘Small Forces,’ used in trajectory models.” Contributing causes included “Undetected mismodeling of spacecraft velocity change” (i.e. undetected inconsistency between metric and English units) along with other systems engineering, organizational, training, and staffing factors. A Lockheed Martin report (Euler 2001) on the MCO and MPL failures added that the “dependence on heritage design and operations eventually became a contributing factor in the loss of the MCO.” Others reported on (David 2000) or have researched (Sauser et al 2009) how managerial aspects may contribute heavily to or even be considered the root cause of project failure, in particular the loss of the MCO. However, the scope of this section will be to investigate the software fault protection architecture of the MCO, categorize fault protection capabilities, discuss efforts made to test the fault protection capability, and finally to consider the effectiveness of the fault protection architecture using the Goal, Question, Metric (GQM) approach.

**Overview of Attitude Control System, Orbit Determination Methodology, and Considerations of MCO Fault Protection Architecture**

Research of the Mars Climate Orbiter mishap has not led to a significant amount of insight as to the overall software fault protection architecture of the spacecraft. Similar software is being utilized that was heritage on Mars Observer (MO)/MGS. Because failures was primarily related to a software fault, there is a sufficient amount of literature to discuss where the fault protection architecture was insufficient, and where incorporation of improved avoidance, tolerance, and/or management techniques might have improved the dependability of the software in the overall “stop-the-flow” framework.

The details of the attitude control system (ACS), orbit determination process, and are well documented (Euler 2001) and summarized below:

All spacecraft require some sort of Attitude Control System (ACS). MCO could be controlled in roll, pitch, and yaw in order to adjust the attitude during the trans-Mars cruise (e.g. for thermal control, nadir pointing, inertial slewing), in order to perform trajectory control maneuvers (TCMs), and to achieve Mars Orbit Insertion (MOI). The ACS was comprised of a sun sensor, inertial measurement unit (IMU), star camera, reaction wheels (RW).
In order to reduce the overall launch mass, the amount of propellants was limited. In order to achieve the functionality needed (i.e. travel to Mars and enter orbit), aeropass and aerobraking techniques were used. This is where the spacecraft enters at the upper layers of the Mars atmosphere and used the drag and gravity force to its advantage in order to adjust to the appropriate velocity to obtain a desired orbit and altitude. Rocket engine modules (REMs) were utilized to achieve this, and the REMs also served to dissipate any angular momentum that accumulated over time in the RWs, and to provide additional attitude control capability during the TCMs and MOI. The REMs had four aft thrusters (pitch and yaw) for the TCMs and 4 roll thrusters, all with small minimum impulse bits (meaning they could be fired for a short amount of time and provide precise motion control). The aft thrusters could be fired in alternating pulses when pitch or yaw adjustments were necessary and to rid the RWs of any stored angular momentum.

MCO performed nominally from launch through the second Trajectory Control Maneuver. This included some uplinked “housekeeping” commands, slight adjustment maneuvers, and RW desaturation (dissipation of stored angular momentum).

**Orbit Determination Process**

As mentioned above, RW desaturation is accomplished by thruster firing (aft or yaw) in order to dissipate residual angular momentum. These thruster firings (occurring about every 17 hours), and other small unbalanced forces (e.g. solar wind) to which the spacecraft was subjected make it necessary to calculate exactly how much the spacecraft acceleration was. This is not directly measurable using the IMU because its accelerometers did not have sufficient precision to detect the very small changes. So, the number of pulses fired from each and every thruster (aft or roll), the corresponding duration, the corresponding pulse, and thruster alignment to determine the delta V.

**Angular Momentum Desaturation Files**

Each pulse fired as part of the RW desaturation process was counted as a data packet by the onboard software. These data packets (approximately 60 per RW desaturation) were downlinked. These data were calculated and downlinked with the appropriate units of measure.
The Spacecraft Performance Analysis Software (SPAS) “small_forces” was used as Ground Equipment to calculate the Angular Momentum Desaturation (AMD). Files were saved on a common server. SPAS did not utilize the correct units of measure (off by a factor of 4.5, N to lb-f conversion) in making its calculations which resulted in the MCO over-shooting its target.

**Categorization of Fault Protection Capabilities**

*Human Management*

NASA ground operations played a vital role in the failure of the MCO. Humans were involved in the calculation of the Automatic Momentum Desaturation and in loading this information to the server. The information pulled off the server was then utilized by the navigation team, and then uplinked to the MCO. Although the human teams did not make the error or cause the fault, they were a potential place where the fault could have been detected in the operational environment. One potential way to improve the human-system interaction in this case would have been to provide the team with valid output ranges for data based on given input, or having an independent software module through which to run the input data and verify that the output was accurate.

*Fault Avoidance*

The “smoking gun” in the MCO software was the lack of a conversion factor between lb-f and Newtons, which was included in the previous version of the software used on another spacecraft but was somehow not carried over to the MCO software. Intuitively, the system should be built without faults, and re-used software should be carefully inspected and regression tested. This, however, can be difficult and fault-free software is improbable.

Teams should do their best to design fault-free software. But, since not all faults will be found and removed, one way to avoid faults per the “stop-the-flow” framework is to avoid humans and external systems, or at least keep them “out of the critical loop (Euler 2001).” Based on these techniques, dependability of the system could have been improved by having a system architecture which did not rely on human operators to that extent. Also, reducing external systems (e.g. eliminating SPAS or “small_forces”) or diligently ensuring that they talk to one another is key.
As described above, the angular momentum desaturations were computed onboard the spacecraft and this information (basically the velocity and the number of times that each thruster was fired) would be downlinked to ground. When this information got to ground, the Spacecraft Performance Analysis Software (SPAS) “small_forces” was used to calculate the Angular Momentum Desaturation (AMD) value. This “external system” had been previously used on Mars Global Surveyor (MGS) and was where the fault lay. Whereas MGS included the lb-f-N conversion factor, this was somehow not carried over to MCO. So, when the “small_forces” calculated the delta-V value from each pulse (thruster firing), it was not correct. Better IV&V performed on the re-used software might have uncovered this fault.

This information was provided to the MCO navigation team, who then used it for future maneuver planning that would later be uplinked to MCO. Clearly, there are two external systems, the small_forces and the navigation team, who could both contribute to mission failure. By reducing or eliminating the reliance on external systems, faults could have been avoided.

**Fault Tolerance**

It is pretty much impossible to design a perfect system, so fault tolerance should be built in. One architectural choice that manifests as a fault tolerant technique (but could also be for failure management or systems engineering), would be some sort of input checking on data uplinked to the spacecraft. If MCO could calculate its own position and then cross check that against what the navigation team was uplinking, the difference would be noted and at least the presence of a problem would be known.

**Failure Tolerance/Management**

Recommendations from the MCO failure board said that for future spacecraft, it is important to utilize redundant, independent components (HW or SW). Utilizing different software modules to perform the same calculations and then performing some sort of voting or incorporating software redundancy could be a solution. Using different and independent systems to determine the position of the spacecraft could also be utilized.
**Accident Avoidance**

The accident in the MCO case was when the spacecraft was lost, either by crashing into the planet or flying by it. Accident avoidance should be incorporated when possible as a last resort to deal with faults that have led to failures which have then not been managed.

In addition to the “safe mode” mentioned above, the MCO failure board suggests that future spacecraft have an onboard emergency suite of maneuvers. This gives some flexibility in designing the system. For example, incorporating a safe mode or safe state when a problem is detected automatically or presumed by the human operating team could have avoided the accident. MCO did have an in-cruise “safe” which could allow it to recover after an anomaly, but there was no way for the system to automatically enter safe mode simply from getting too close to the Mars surface. One method could be to use independent methods of position determination. For example, if radar determined that MCO was within 100 km of Mars surface (or within a lower boundary limit), that an emergency maneuver could be performed so that it could enter a higher orbit than required. Although not the intent of the MCO, this would be a recoverable failure.

**Testing of Fault Protection Capability**

Anyone who works for or around NASA has heard “Test like you fly.” In this case, we have the benefit of hindsight or “hindsight bias” (Woods 1999) when performing case studies. However, this example is a case of failure of re-used software. A simple constant converting between lb-f and Newtons, which was in the previous version, was not carried over to the MCO software. Software engineers, software developers, and system engineers, must ensure that a thorough independent verification and validation program be implemented particularly on re-used software. Regression testing must be performed, and changes in external systems must be considered. It is imperative that all anticipated input and output ranges are specified and tested, using equivalence classes, in order to test the normal operation of the system as well as the boundaries of normal inputs and outputs. Expected outputs must be compared against as-run output data based on a known/given input set in order to validate algorithms/functionality of the system.
Clearly, this software (that on MCO and that of small_forces, which not on the spacecraft was part of the software architecture) was not tested sufficiently to enable mission success.

**Effectiveness of the Fault Protection Architecture**

Utilizing the goal, question metric approach, we see that the MCO achieved some of its goals, but not all. Because it was not able to ultimately perform its desired task, it was not successful overall.

- **Goal**
  - Mars Climate Orbiter reaches Mars
  - MCO performs Mars Orbit Insertion maneuver
  - MCO enters orbit around Mars with sufficient propellant for slight orbital adjustments
  - The MCO communication system architecture functions
- **Questions**
  - Is the orbit determination process sufficiently precise to get MCO within range of Mars Orbital Insertion? Yes (even with errors in the Angular Momentum Desaturation coding).
  - Is the orbit determination process sufficiently precise for MCO to enter orbit around Mars, with sufficient propellant for slight orbital adjustments? Yes, but there was only room for error in one direction (i.e. coming in too far from Mars acceptable; too close is failure).  
  - What happens if MCO comes too close to Mars’ surface? Nothing and herein lies the problem. There was no safe mode incorporated in the fault protection architecture to enable MCO to enter a safe state and avoid coming in too close to the surface.
- **Metrics**
  - Was the goal of reaching the vicinity of Mars met? Yes.
  - Was the goal of Mars Orbit insertion achieved? Yes.
  - Was the goal of MCO entering orbit around Mars achieved? No. MCO came in too close to the surface of Mars, and was lost. The orbit determination process
and angular momentum desaturation (AMD) processes were flawed. The latent fault (lb-f to Newton conversion factor) was never found, detected, identified, or fixed, and this led to an inconsistency between where the MCO team thought the spacecraft was and where it actually was.

- Did the MCO communication system architecture function as required? Yes.

**Mars Reconnaissance Orbiter**

The $720 million dollar Mars Reconnaissance Orbiter launched August 12, 2005 to become one of five spacecrafts entering Mars’ orbit. It was designed and launched to continue NASA’s strategy of exploring and examining the red planet, especially in NASA’s overall objective of finding the existence of water on its surface. The orbiter’s primary function is image processing to survey the planet by utilizing six onboard instruments for probing Mars’ atmosphere, surface, and subsurface (Space Topics 1). NASA accomplishes the orbiter’s goal of capturing detailed imagery of the planet by using an advanced technique to keep the object stable in space, called aerobraking. They accomplish stability by allowing the orbiter to drop down lower into Mar’s atmosphere to slow down the orbiter hence making it more stable to capture the needed photography.

All these objectives and complex functions of the orbiter make the probability of failure high. Besides performing all these tasks without fault, the orbiter also has an objective of a 5.4 year functional life. This demands a highly stable system that needs to be designed to accomplish as close to 100% up-time throughout its mission timeline as possible. Essentially, as close to 100% availability is needed, meaning the system must be designed for reliability or dependability. Designing such a system is extremely difficult as all assumed and unknown failures must be handled to prevent the system from losing crucial functionality, especially in scenarios that would cause a complete mission failure. In analyzing the mission requirements it is highly probable that mission failure would be reached before success with the countless minor factors that could lead to catastrophic malfunctions.
**Fault Protection Architecture**

For MRO to handle the complexity of the system and to prevent anything from causing a massive failure the orbiter uses a semi-autonomous Fault Protection software known as SPIDER (Spacecraft Imbedded Distributed Error Response). This section will discuss the capabilities of the software fault protection architecture (flight and ground software) including fault avoidance, fault tolerance, failure tolerance/management, and accident avoidance.

This architecture was designed to be highly reliable and can handle extensive redundancy and cross strapping tasks which are prime complexity requirements for the orbiter. SPIDER was developed using the C-language and was designed to be reusable to be used by various spacecraft missions (Seale 2002). Even though SPIDER is a very advanced architecture it still can not function alone, as it still must rely on ground operations to manually set the system back into a normal state when in a safe state mode and relies on ground operations also to handle threatening conditions that are unrecognizable by the system. Even though ground operations is required the fault protection plays one of the most crucial roles in the orbiter system as it acts as a first responder for error handling, this is especially important in many cases when it can handle failures before the operators can even recognize an issue exists. As a result it prevents the system from having mass failure hence causing mission failure without the ground mission operations crew having to react. The fault protection software is able to put the system into a safe state, cutting off any open threats that may harm the system, depending upon the failure that is encountered.
Throughout the majority of MRO’s mission phases the fault responses are generic. An example of a generic fault response is one in which the system will automatically turn to safe mode and cut off any uncritical equipment and functionality if a fault is detected. The system can then only be set back into an operational state by NASA’s ground operations using required protocol. Unique responses are only used during phases that need to be accomplished the first time around, i.e. for critical mission phases. In such phases (e.g. Mars Orbit Insertion) a simple fail-safe responses are not sufficient. Rather, they need responses that will ensure functions will result in full completion of the mission.

Having a clearly thought-out and well-designed Fault Protection Architecture, the incorporation of SPIDER and ground operations was augmented by fully tested contingency plans that can be used instantaneously during the mission. This architecture has been very well designed and has had an outstandingly great outcome for the orbiter as it has survived many mission critical issues, which, without the fault protection architecture in place, could have resulted in the orbiter becoming nonfunctional.

There are many reasons why the SPIDER fault protection is a robust design for the survival of the Orbiter, one of which is its high level of reliability. The system meets the requirement of the Orbiter to be fully single fault tolerant (Note: the level of fault tolerance required on a spacecraft is typically related to the acceptable risk level for the payload, which is tied to the size, cost, and criticality of the mission. Ground processing safety may require that some systems be dual fault
tolerant if their failure could result in death or a catastrophic event). SPIDER contributes to MRO being capable of achieving this requirement because the architecture was designed heavily around redundancy and cross strapping. Basing the system architecture on redundancy and cross strapping enabled the Orbiter to handle random failures at pretty much any given time, presuming that no other system failures were occurring. Some different types of redundancy that play a major role in the system’s architecture are outlined in Bayer 2007):

- **Functional Redundancy**: allows the system to handle one failure in multiple ways
- **Block Redundancy**: allows for parallel elements to function to fix a given failure
- **Cooperative Redundancy**: allows a function of the system to be split into portions so that a given function can succeed even though one portion may fail
- **Cross Strapping**: allows a system to handle multiple failures at a given time

SPIDER’s architecture consists of three software layers: Component Level Fault Protection, Performance Level Fault Protection, and System Level Fault Protection. These layers are placed in a hierarchy structure that allows failures to flow up the levels to be handled by the appropriate logic that can detect the given failure. Through this method of implementation, the upper levels in the hierarchy are capable of calling lower levels to perform specific tasks that they are capable of executing. As high levels may call functionality in lower levels they are still not able to override and take priority of tasks already being run in those levels. The purpose of each level is (Bayer 2007):

- **Component Level Fault Protection**: distributed logic, is used to communicate with the orbiter’s hardware
- **Performance Level Fault Protection**: set of monitors, each used to oversee the performance of each subsystem
- **System Level Fault Protection**: contingency mode executives, contains logic to prevent failures from occurring
Categorization of Fault Protection Capabilities

Considering the Mars Reconnaissance Orbiter in the “Stop the Flow” framework (Laird 2010):

**Human Management**

NASA ground operations play a vital role in the Fault Protection Architecture in MRO as it cannot return to an operational state without the ground operators following and implementing the needed contingency plan to bounce the system back to a fully functional state. Ground Operations also play a major role when the system is unable recognize a failure.

**Fault Tolerance**

The system is tolerant to faults (single fault tolerant throughout) and also can manage them when they do occur. The three levels are designed to execute the appropriate response in the given phase when a fault does occur. If a response is flagged, NASA is notified and the fault is contained so it is not capable to perform further damage.

**Failure Tolerance /Management**

The MRO system also is able to tolerate failures that may occur and has been designed to avoid a domino affect if one failure occurs. It also has a set of rules and functions it can execute to manage a failure from occurring. It is also able to monitor and communicate back to NASA the failure that has been recognized.
**Accident Avoidance**

The MRO system can enter into safe mode when necessary to avoid accidents.

**Testing of Fault Protection Capability**

To test the MRO systems Lockheed Martin was able to use MathWorks tools to not only design its systems but also used them to create real-time simulations. With the use of both MATLAB and Simulink the company was able to effectively create advanced algorithms and use numerous variations of parameters to efficiently simulate deep space situations. Simulations incorporated input from sensors including inertial measurement units, star trackers, and sun drives (Lockheed Martin 2006). Thousands of control parameters for various phases and scenarios were input and used during the testing phase. Lockheed Martin also developed an Orbiter Test Bed for running real time hardware in the loop simulations.

**Effectiveness of the Fault Protection Architecture**

The Mars Reconnaissance Orbiter has faced numerous failures during its time of being operational. With the accomplishment of having a well-designed fault protection architecture and dedicated ground operations staff they have been able to recover from each of these failures without jeopardizing the mission. Such failures that occurred during MRO’s operation have been memory corruption, star tracker sensitivity discrepancies, downlink connectivity failure, and many other critical failures to the spacecraft. Even though the failures could have threatened the mission, the fault protection implemented prevented a system-wide failure from occurring. In the majority of these cases the fault protection software has been successfully able to put the system into safe mode allowing ground operations to diagnose the failure at hand and do any necessary repairing to the system. During MRO’s mission the fault protection architecture and process implemented has been highly effective to reach mission success.

The GQM approach for evaluating the software of MRO is utilized below:

- **Goal**
  - Design a system to reach the objective operational lifecycle of 5.4 years
  - MRO must survive multiple instrumental malfunctions
- The MRO communication link between Ground Operations must stay active
- MRO must successfully utilize digital instruments to take detailed imagery of Mars

**Question**

- Will the Orbiter be able to handle multiple repetitive malfunctions through its lifecycle? Yes, MRO’s fault protection is able to handle multiple failures at a time, with the option of shutting down any noncritical components to save the orbiter from mission failure.

- Will MRO’s architecture be able to maintain an efficient communication link to NASA Ground Operations? Yes, if the uplink/downlink to NASA is unexpectedly disabled MRO will immediately switch to safe mode to repair the connection and isolate any failure occurring.

- Will MRO be able to successfully stay stable enough and reach a close enough distance to Mars to capture efficient readings and images? Yes, MRO was designed to perform highly advanced aerobraking maneuvers, which has also been worked into the Orbiter’s high tolerant software architecture.

**Metrics**

- Was the Goal of MRO reaching its lifecycle time reached? Yes, still active. Has also been able to handle numerous mission critical failures

- Has the uplink/downlink to NASA stayed efficiently active? Yes, there have been several failures that have occurred with the downlink however each failure has been resolved for high rate of up time for the communication link

- Has MRO successfully reached the goal of capturing detailed images of Mars? Yes
Mars Exploration Rovers

NASA launched twin rovers on June 10\textsuperscript{th} and July 7\textsuperscript{th}, 2003. The rovers landed on Mars on January 3\textsuperscript{rd} and January 24\textsuperscript{th} 2004. The primary mission of the rovers was to search for and characterize rocks and soils that potentially contains information about past water activity on Mars. The rovers were designed to operate for 3 months. Both rovers have remained in operation for a much longer time period and one of them is still operational and it is possible that the other rover will become operational again. The MER rovers used VxWorks from Wind River. This was a commercial operating system. (Biesiadecki and Maimone 2006)

Fault Protection Architecture

The fault protection objectives changed based on the phase of the mission. While cruising to Mars, there were no time-critical events and the focus was on putting the “spin-stabilized spacecraft in a power positive, communicative, and thermally stable state (Neilson 2005).” During EDL (Entry, Descent and Landing) time is critical and the fault protection system is “designed to use all available resources to ensure a safe landing (Neilson 2005).” For rover operations on the surface of Mars, time was also very important to complete all mission science objectives. The rover was designed to last 90 sols and each sol was needed to complete the project objectives (a sol is a Martian day, 24 hours 37 minutes long). The focus of this section will be on the fault protection for rover operations on the surface of Mars.

It is important to note that in the JPL approach, fault protection is an integral part of the system design (Dvorak et al 2000). Nielson (2005) states that “Fault protection for these rovers was incorporated into every subsystem, built into hardware and software, and system engineered to make sure it all works together.”

The most critical fault protection objective to be achieved on a daily basis is sufficient power for the rover to operate. The rovers needed to enter a “sleep mode” every Martian day (each sol) to recharge the batteries. In sleep mode, the rover avionics and CPU were powered off. During this time, the hardware needs to maintain safe thermal and power states. The rover must wake from sleep mode and initiate communication and this must be done without communication from the operations team. There is an algorithm that controls the autonomous shutdown and wakeup
behavior. Nielson (2005) explains that the algorithm “wakes up the rover when the solar arrays can support the loads required for communication (in receive mode), and it puts the rover back to sleep once the solar arrays can no longer support those loads.” The autonomous shutdown mode is initiated if “there is no communication window active and either the solar array current is lower than a configurable parameter or the vehicle has been awake too long (Neilson 2005).”

There are many aspects of fault protection architecture as the flight software performs a number of functions. If a server error was detected during the rover’s initialization process, the software would postpone resetting the system for a pre-determined amount of time. If the read-back state did not match the predicted state, the flight software would try to establish a hardware state several times. There is a software health function that continuously checks for unresponsive or suspended tasks. “If severe errors are detected, the flight software will force a reset of the flight computer, causing a flight software initialization. The system response (after initialization) turns off science instruments and warm-up heaters. Survival heaters are reinforced on and all sequences are deactivated. After a flight software reset, the vehicle continues to perform scheduled communication windows in autonomous shutdown mode (Neilson 2005).” Fault protection is also provided for the telecom system. If problems occur when pointing the High Gain Antenna, then the Low Gain Antenna with lower data rates is used. (See figure 1)

The flight software also acts as an arbitrator if there is a conflict between fault responses. The local-level fault protection and the system-level fault protection are illustrated in Figure 1 (Neilson 2005).
Categorize the fault protection capability

This fault protection architecture has the following fault protection capabilities:

**Human management**

The rovers are monitored on a daily basis by various NASA teams on earth. The teams monitors the information sent (or not sent) by the rovers. If the information indicates major or minors errors, the anomaly team decides whether or not to intervene.

**Fault avoidance**

The rovers have an algorithm that controls the autonomous shutdown and wakeup procedure. This algorithm avoids a low energy fault and ensures that the rover has sufficient power to operate.
**Fault Tolerance**

The system is shutdown every day and benefits from software rejuvenation.

**Failure Tolerance/Management**

The autonomous shutdown and wakeup procedure check for adequate power. If a fault occurred that prevented the rover from getting enough power, the system does not wakeup and avoids the failure of starting up without adequate power.

**Accident Avoidance**

On long drive days, the Spirit followed a path determined by rover planners at JPL combined with “an autonomous navigation portion, during which the rover watched ahead for hazards and chose its own path to avoid them (NASA Facts 2010).” Path-planning software also assisted in hazard avoidance (“In-situ Exploration…”). Finally, the rovers had some hazard tolerance. If it was impossible to avoid a rock, for example, the suspension system would enable the rover to be maneuvered over the rock (“In-situ Exploration…”).

**Testing of Fault Protection Capability**

Before the launch, there was a “FIDO field test”. This was a rover used for testing on Earth. The mission criteria were met for the FIDO test but it was sometimes not possible to replicate the target environment. This is particularly true when the target environment is another planet (Mars in this case). The rovers’ fault protection capability was tested most comprehensively beginning with the launch. The goal of the fault protection design during the cruise to Mars was to put the spacecraft in a “power positive, communicative and thermally stable state (Neilson 2005).”

In the Entry, Descent and Landing (EDL) phase, the goal of the fault protection design was to ensure a safe landing. After the landing on Mars, the goal of the fault protection design was to “protect the mission objectives without compromising vehicle health (Neilson 2005).”

How the fault protection capability was tested varied depending on the phase. A successful transit to Mars is how the cruising protection capability was tested. In the EDL phase, a successful landing is how the fault protection capability was tested. After landing, the fault protection capability must ensure that the system has sufficient power to perform the many tasks
necessary for the rover mission. The fault protection design worked well for the first 17 sols. After the 17\textsuperscript{th} sol, the DTE (direct-to-Earth) signal suddenly dropped out. The anomaly team received enough information to determine that the rover was rebooting over and over and not successfully accessing the FLASH memory. The anomaly team had also determined that the rover has not shutdown overnight. This was actually another failure in the fault protection system. The team sent a command to place the rover in “crippled mode” (this command was structured to not use FLASH memory). It took about a week for the anomaly team to determine that there was a design error in the DOS file system library code. A representation of the file system structure (Table of contents – TOC) is in RAM to optimize performance. Nielson (2005) stated that when “a file is deleted from the file system; the TOC is changed to reflect that the file is deleted but the size of the TOC does not shrink.” Cruising through space for seven month combined with EDL and a week of deployments had caused the TOC to consume all the available RAM. The fault protection design had failed to adequately address unexpected growth in the TOC in RAM. Testing before the launch also did not identify this issue. Fortunately, this did not result in mission failure.

**Effectiveness of the Fault Protection Architecture**

The GQM approach is utilized to assess the effectiveness of the fault protection architecture.

- **Goal**
  - Rover survives in Martian environment for 90 days to complete scheduled tasks
  - System gains enough power each day to sustain it for operations driving distance goal – 600 meters (Mars Exploration Rover Mission Press Release March 24, 2010)

- **Questions**
  - How much power do we have? Generates about 140 watts of power for up to 140 watts per sol
  - Is this power sufficient for operations? Yes – Rover needs 100 watts of power to drive (Spacecraft: Surface Operations: Rover)
  - What happens if power is not sufficient for operations? Autonomous shutdown mode is invoked
• Metrics – Was the goal of 90 days of operation met? Yes
  o Opportunity – January 25, 2004 to 2010
  o Has the rover failed to operate on some days? Yes – After 17 days of operations, Spirit failed to operate for about two weeks. Spirit also stopped operating on March 22nd, 2010 and has not resumed operation. These are two examples of failed operations.
  o Was the distance goal of 600 meters met? Yes Spirit – 7730.5 meters Opportunity 20553.25 meters (as of April 20th, 2010). Mars Exploration Rover Mission Press Release March 24, 2010

Conclusions

The Mars Climate Orbiter is a classic example of the importance of software testing and system-level testing. A minor fault (an omitted conversion factor) resulted in total mission loss. While this simple error should have been found, the overall fault protection architecture did not prevent mission loss. There was no tolerance for this fault (conversion factor), there was no way to manage the failure (coming in too close to Mars), and ultimately no way to avoid the accident (presumably crashing in to Mars). This paper has attempted to point out areas in the “stop-the-flow” framework where additional fault protection capabilities might have prevented the ultimate failure – in the areas of human management, fault avoidance, fault tolerance, failure management/tolerance, and accident avoidance.

The Mars Rovers met and exceeded all mission goals. The fault protection design was sound and very effective. The rovers operated successfully for over 6 years! The fault protection provided by the autonomous shutdown and wakeup behavior enabled the rovers to gain power each day. This provided the energy needed to drive on the surface of Mars and gather the information about past water activity on the planet. The fault protection also played a key role in the cruise to Mars and the entry, descent, and landing on Mars. While “one of the leading causes of system outages is human beings (Laird 2010),” the NASA teams corrected the FLASH memory problem
and prevented it from reoccurring. The fault protection design was robust and has enabled continued operation of at least one of the rovers even to this day. However, the fault protection design has some shortcomings that have been overcome with the assistance of the NASA teams on earth.

The Mars Reconnaissance Orbiter also proved to be a success. Throughout its operational phase it has provided NASA with thousands of key images and readings from Mars’s atmosphere. It has faced numerous communication and sensor issues however no issue has resulted in a failure of mission success. With its advanced software and hardware architecture backed by a complex and flourishing fault protection design it has managed to stay operational throughout all its failures. The SPIDER fault protection software has been proved to be a very reliable and reusable architecture through the MRO mission. MRO’s architecture will certainly continue to be a standard throughout all NASA spacecraft missions.

Future work could include determining precisely how additional spending might buy-down risk by improving the fault protection architecture. For example, hardware and software redundancy (many techniques have been discussed in class) could be incorporated, but not for free. Determining how much it would cost to “buy” overall dual fault tolerance, or how much to have redundant independent systems could be useful. It was mentioned in the literature about the Mars Climate Orbiter (and also Mars Polar Lander) that the “Better, Faster, Cheaper” approach ended up leaving the mission management team, and navigation teams were only one-person deep (Euler 2001). In that case, there was no redundancy in the human element of the fault protection architecture. As with fault detection and code inspections, there will be a point where the costs no longer outweigh the benefits (i.e. additional fault protection capabilities will provide only marginal improvement in overall software system dependability). Knowing where this benefit becomes minimal would be very useful for any mission- or safety-critical software applications.
Sources


Bayer, T.J. (2007). In-Flight Anomalies and Lessons Learned from the Mars Reconnaissance Orbiter Mission. IEEE.


Subramanian, N. and Chung, L. Software Architecture Adaptability: An NFR Approach. (???) Applied Technology Division Dept. of Computer Science Anritsu Company University of Texas, Dallas


