The Evolution of a SPIDER

Fault Protection, Incremental Development, and the Mars Reconnaissance Orbiter Mission

Eric H. Seale
Lockheed Martin Astronautics Operations
P. O. Box 179 M/S S1008
Denver, CO 80201-0179
303-977-5024
Eric.H.Seale@lmco.com

Abstract—Spacecraft on planetary and deep space missions require a high degree of autonomy in responding to faults that occur during their flights. Meanwhile financial, staffing, and schedule constraints weigh heavily in programmatic decisions. The **SPIDER** (**SP**acecraft **I**mbedded **D**istributed **E**rror **R**esponse) emergent system architecture was originally conceived seven years ago to provide a robust fault protection capability, while minimizing the amount (and thus cost) of non-recurrent engineering required to adapt the architecture to subsequent missions.

The Mars Reconnaissance Orbiter (MRO) spacecraft is more complex and more expensive than any other SPIDER mission to date – with correspondingly lower programmatic risk tolerance. The layered and distributed nature of SPIDER's emergent system logic is proving to be an enabling factor in steadily improving SPIDER implementations over a series of missions. The course of fault protection development for MRO is showing that the SPIDER architecture can and will easily scale to accommodate the higher levels of autonomy required for this higher investment mission. By following a path of evolutionary improvement, significant robustness is being added to the architecture with only incremental additions to the well-tested heritage logic.

IEEEAC paper #1098, updated December 5, 2002
0-7803-7651-X/03/$17.00 © 2003 IEEE

TABLE OF CONTENTS

1. INTRODUCTION

As has been described in detail previously [1], the SPIDER fault protection architecture was originally conceived at Lockheed Martin Astronautics (LMA) in 1995 in support of the Mars Surveyor Program (MSP) 1998 Orbiter and Lander missions. These missions spurred SPIDER's development via two main factors – a contract requirement to code flight software in the C programming language, and the "Better-Faster-Cheaper" mandate of the mid-1990s.

The C-language mandate unwittingly provided a rare opportunity for a truly "clean sheet" approach to the flight software architecture and accompanying fault protection design. Meanwhile, the "Better-Faster-Cheaper" paradigm resulted in severe constraints on software development cost, hardware cross-strapping, and flight operations staffing -- resulting in a concerted effort to maximize software reuse, and to facilitate reuse on subsequent missions. In the process of pursuing an easily reusable architecture, it was discovered that the resulting logic displayed a number of emergent behaviors.

Subsequent to the MSP'98 missions, SPIDER has flown on the (Discovery 4) Stardust spacecraft, the MSP'2001 Odyssey orbiter, and on the (Discovery 5) Genesis

spacecraft. With each new mission came refinements to the architecture in order to support the given mission's unique requirements, and programmatic views of risk tolerance and spacecraft autonomy. We now are in the process of further refining the architecture to provide on-board fault protection for the MRO spacecraft, a platform of considerably greater cost and complexity than any previous SPIDER mission.

2. ARCHITECTURAL OVERVIEW

SPIDER's architecture can best be thought of in terms of three structural concepts. Originally inspired by the 7-layered OSI (Open Systems Interconnection) model of network architecture, SPIDER's functionality was partitioned into layers. At a more detailed level, SPIDER's structure is object-oriented. And while emergent behavior was not originally a design goal, it became clear, even during SPIDER's original development, that it displayed emergent properties. The emergent nature of SPIDER's behavior has proven to be a crucial factor in its subsequent evolution.

A quick look at layers

Layering is a widely used, and well-proven system structuring technique [2]. In a layered architecture, each layer is allocated a subset of system functions. Each layer relies on lower layers to perform more primitive functions, and to conceal the details of those functions. In turn, each layer provides services to the layers above it.

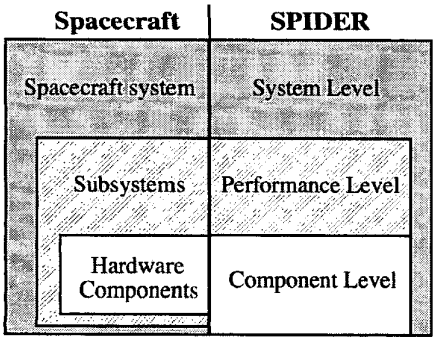


Figure 1 – Spacecraft vs. Spider

SPIDER's architecture is divided into three software layers analogous to the levels in a hierarchical decomposition of a spacecraft design (shown in Figure 1). Each layer of failure detection and response logic has a specific range of concern and an allowable range of responses. The layers cooperate to isolate and respond to any sensed failures. Note that in contrast to a level of spacecraft architecture, a given layer of SPIDER is not a subset of the layers above it.

Within each of SPIDER's three levels, fault detection logic implements a variant of the same basic decision process (shown in Figure 2). Fault detection attempts begin at the interface with hardware; logic within each successive layer then does what it can to resolve and respond to a failure. Parameters are selected to assure that lower levels of SPIDER's logic respond more quickly than upper layers.

If logic in a given level can't detect a particular failure, the failure "flows up" for the next highest layer to detect. If logic within a level detects a failure but can't fix it, the logic flags the failure's occurrence; this "indicated failure" flag then becomes fodder for a detection in the next level up.

Note that most fault data is passed up the hierarchy only on request (i.e., it is "pulled" by higher layers, not "pushed" by lower layers). This avoids saturating higher layers of the architecture with multiple detections of the same failure

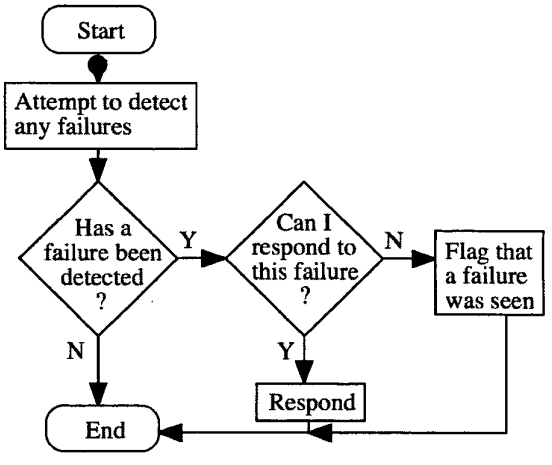


Figure 2 – Generic SPIDER Decision Process

from propagated symptoms. Actions of upper levels are then largely based on data from lower levels, although each subsequent level also has new detections based on increasingly abstract health measures.

Upper levels can force lower ones to perform actions (in particular, to reset their logic states), but do not "out prioritize" or overrule them. In this sense, SPIDER is related to, while not descended from, Brooks' subsumption architecture [3]. In Brooks' terms, SPIDER provides fault detection and response via "zeroth level" behavior, essentially analogous to a biological organism's reflexive pain avoidance behaviors.

Object orientation

An emphasis on reducing flight software reuse costs originally led MSP development personnel to maximize use of Object-Oriented Analysis (OOA) / Object-Oriented Design (OOD) software methods. When strictly followed, these methods lead to a flight software architecture (at least for component control functionality) that uses the spacecraft itself as a model.

As a result of object oriented methods, the SPIDER architecture essentially mirrors the design of a generic spacecraft. SPIDER's component level consists of distributed logic, implemented within "virtual component" software objects (each responsible for modeling, and interfacing with, a given hardware component). SPIDER's performance level consists of domain performance monitors, each modeling a given domain of spacecraft functionality. Finally, SPIDER's system level consists of objects performing on the behalf of the spacecraft as a whole.

SPIDER as an Emergent System

It was evident early on in SPIDER's development that it manifested emergent behavior. Lacking a consensus definition of either emergent behavior or emergent systems, let's work with those properties included in most definitions of emergent software systems:

- The system's logic is highly distributed, being implemented in a number of separate pieces scattered throughout the software system.
- Each piece of logic performs according to a set of pre-defined local rules, appropriate to its place and function within the system.
- The distributed pieces of logic each function independently, but in a collaborative fashion. In most cases, no central control is needed or provided.
- Desired system behavior comes from the integrated behavior of all the independent parts.

Emergent systems tend to display behavior that is non-deterministic, at least on small scales. Since non-deterministic behavior is not acceptable in the spacecraft design or operations communities (at least, not yet), SPIDER is designed to provide deterministic, and thus fully testable, logic. For lack of a better term, SPIDER can be described as a "linearized" emergent system. It should be noted that contrary to most emergent systems, only minimal communications take place between objects within a given layer of SPIDER.

3. SPIDER'S LEVELS IN DETAIL

Component Level Fault Protection

SPIDER's lowest level, Component Level Fault Protection (CLFP), consists of distributed logic to perform low-level error detections and responses. At this level, error detections are based on direct measurements from a given component's telemetry. Component level responses are not allowed to affect other objects; accordingly, many objects' component level responses consist of little more than setting a flag to indicate component failure. While a failure is indicated at the component level, the affected object is required to continue outputting "best guess" data, and to continue operations as best it can. This keeps the spacecraft operational while fault protection logic works to resolve the failure.

Component level logic is implemented in widely distributed fashion, as a part of "virtual components" in flight software (see Figure 3). In object oriented fashion, each virtual component contains all code needed to interface with its respective hardware component – both "nominal" component-control flight software as well as low-level fault protection logic. Each virtual component also maintains data on the state of its associated hardware component. This arrangement simplifies component state tracking, and reduces data flow between flight software objects. It also eases reuse of flight software; in many cases, component level software is reused from mission to mission without any changes.

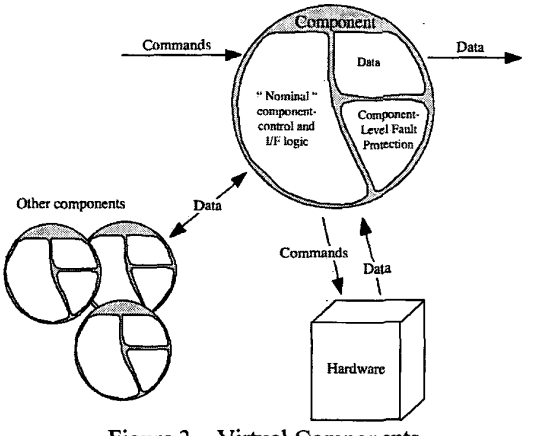


Figure 3 – Virtual Components

Performance Level Fault Protection

Performance Level Fault Protection (PLFP) consists of a set of monitors, each responsible for supervising the performance of a subset (domain) of spacecraft behavior. Each domain roughly corresponds to a spacecraft subsystem, although a few subsystems are subdivided into multiple domains, each protected by a separate monitor. There are 8 domains in most implementations of SPIDER:

- Attitude control
- Attitude knowledge (i.e., attitude determination)
- Articulation
- Communications (telecom)
- Data bus (inter-spacecraft communications)
- Momentum control

- Power control
- Science
- Software

Performance level detections are of CLFP-deduced object conditions, as well as more abstract derived indications of domain health. Performance level responses generally are not allowed to affect other domains, but may impact any component in the given monitor's domain. Responses here include component power cycling, component swapping (i.e., redundancy management), resets of I/O cards, and the like.

System Level Fault Protection

System Level Fault Protection (SLFP) consists of contingency mode executives (Safe Mode, uplink loss, and downlink loss) and various fault protection utilities. System level logic is allowed to manipulate anything on the spacecraft as needed to assure protection from failures. Historically, SPIDER's system level logic has been fairly unsophisticated (part of a legacy of constrained autonomy), and was an early target for improvements when the mission requirements of MRO were first seen.

4. Implementation History, Experiences

The first missions to implement SPIDER were the Mars Surveyor Program 1998 Orbiter (Mars Climate Orbiter, MCO) and Lander (Mars Polar Lander, MPL). Due to tight schedule and budget constraints, as well as understandable programmatic concern about a new hardware and software architecture, MCO and MPL flew with a fairly spartan implementation of SPIDER logic. Logic dead-ends were not yet explicitly treated, and the interaction between levels of SPIDER logic was still immature.

In this initial (admittedly rough) implementation of SPIDER, redundancy management logic was situated in various places within both the component and system layers, on an essentially *ad hoc* basis. Meanwhile, performance level monitors were the core of the architecture, "pulling" data from the component level but "pushing" requests for Safe Mode entry or avionics reboots / string swaps up to the system level. The ill-fated MCO and MPL spacecrafts' lifetimes were short, but experienced no problems due to this still immature version of SPIDER.

The Stardust spacecraft was launched only a month after the two MSP 1998 spacecraft, but was designed for a much longer mission than was either of those spacecraft. Accordingly, its fault protection logic contains a number of augmentations to that for the earlier missions -- primarily improvements in response logic using avionics string swaps. Simple logic dead-ends were, for the first time, explicitly treated. Now three years into its mission, Stardust is running smoothly, and while it has exercised SPIDER logic a number of times (Safe Mode seeing the most mileage), it has experienced no problems due to SPIDER.

Development of the Mars Surveyor Program 2001 (Odyssey) Orbiter started only shortly before the MPL, MCO, and Stardust spacecraft launched. This facilitated the inclusion of "lessons learned" in the Odyssey implementation of SPIDER. In particular, additional effort was expended to eliminate all logic dead-end scenarios, and improvements were made to the interface between CLFP logic for power distribution vs. that for other components. Odyssey was launched on 7 April, and reached Mars on 23 October, 2001. During its mission to date, Odyssey's implementation of SPIDER has been exercised only a few times, exposing no logic problems in the process.

Design of the Genesis spacecraft started shortly after that of Odyssey, so much of the two spacecrafts' fault protection development proceeded in parallel. Unlike Odyssey (or any other heritage SPIDER spacecraft) though, Genesis is a spin-stabilized spacecraft in order to support the collection of solar wind data via charged particle monitors and solar wind sample collection arrays. Genesis' sharp departure from its heritage proved to be a stout test of fault protection logic reuse, but was fairly easily accommodated by the object-oriented architecture [4]. Genesis was launched on August 8, 2001, and was inserted into its L-1 halo orbit 100 days later on November 16. SPIDER has seen little action on Genesis to date.

5. SPIDER and MRO

The Mars Reconnaissance Orbiter is scheduled for launch in 2005, and is tasked with analyzing the present and past climate of Mars to an unprecedented level of detail. The mission will further investigate hints of water uncovered by the Mars Odyssey spacecraft, and attempt to bridge the gap

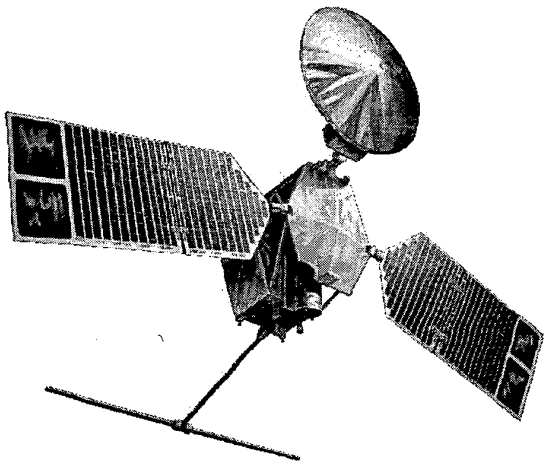


Figure 4 – The MRO Spacecraft

between surface observations and measurements from orbit (portions of the surface will be mapped to resolutions of 30 cm / 12 inches). The MRO spacecraft shares a significant amount of avionics, as well as some mission design elements, with spacecraft designs on which SPIDER has previously run. There are, though, a number of significant departures from heritage.

MRO is being designed for a 5.4 year mission life (vs. 3 years for both Odyssey and Genesis). As a result, MRO has significantly increased cross strapping compared to heritage missions. Figure 5 shows a schematic comparison of cross strapping for Odyssey vs. that for MRO.

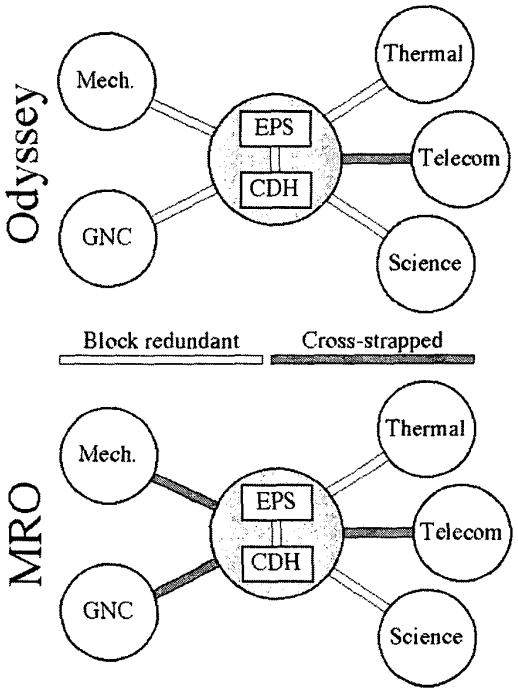


Figure 5 – Cross-strapping compared
(GNC = Guidance, Navigation & Control,
EPS = Electrical Power Subsystem,
CDH = Command & Data Handling)

MRO also is flying a much larger number of payloads than has its forebears – 6 science instruments and 2 engineering experiments vs. 3 science instruments for Odyssey and 4 for Genesis.

As would be expected with MRO's increased lifetime and science requirements, programmatic tolerance of risk is dramatically lower than that seen on heritage missions. One particular manifestation of this shift is that MRO must be able to recover from any credible single failure during the critical Mars Orbital Insertion (MOI) burn. In contrast, previous missions "waived" the requirement for fault recovery during such short duration mission critical events.

From a fault protection perspective, the increase in hardware cross-strapping and the MOI recovery requirement pose the two biggest challenges. In any event, work done on MRO to date indicates that both can be accommodated with straightforward enhancements to the heritage architecture.

Component level fault protection changes

As has been the case with previous missions, the only major changes at the component level come in response to new or modified hardware. For MRO, this primarily means new CLFP logic for science instruments, new avionics (gimbal motor controllers, star tracker), and new 1553 bus interface code.

Performance level fault protection changes

At the performance level, MRO's implementation of SPIDER will introduce two major changes. In order to accommodate the increase in cross-strapping for MRO avionics, a number of performance level monitors will see the addition of steps to their heritage component failure response logic. As this just adds a few lines of software in a few places, the fault protection impact of increased cross-strapping will be fairly minor.

A larger change (at least in architectural terms) is the reworking of the interface between performance and system level logic. Previously, performance level logic has had the responsibility for requesting either entry to Safe Mode, or an avionics string reboot / swap. This meant that some data had to be "pushed" up to the system level, complicating parameter generation and system level logic for some failure modes (as great care needed to be exercised to avoid and / or mitigate race conditions between performance level monitors).

MRO is addressing this issue by giving performance level monitors an interface analogous to that of component level objects. Monitors will now set indicators corresponding to any need for domain initialization (which generally means that Safe Mode is required), or to indicate a logic dead-end (which generally means that a reboot or string swap must be performed). It is then the responsibility of system level logic to "pull" this information up the hierarchy.

System level fault protection changes

SPIDER's system level logic will see, by far, the largest amount of change for MRO. Here, a new executive is being added (currently named, with no surplus of imagination, the "Fault Protection Executive," or FPE) to handle the new interface with performance level monitors, as well as to coordinate entries to Safe Mode and reboots / string swaps.

	MCO / MPL / Stardust				Odyssey / Genesis			MRO			
System	Initial				Rev. 1			Rev. 2			
Performance	Initial / Rev. 1				Rev. 2			Rev. 3 (baselined)			
Component	Initial				Rev. 1 (baselined)			Rev. 1			
	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005

Figure 6 – SPIDER's Evolution to Date

SPIDER implementations to date have been configured to exhibit fail-safe, rather than fail-operational behavior. Any major system reconfiguration or reinitialization results in the termination of any active on-board sequence(s), and entry to Safe Mode. It is then Safe Mode's charter to reinitialize the spacecraft state, and to maintain the spacecraft in a stable and sustainable configuration until ground personnel can intervene.

MRO's updated variant of SPIDER allows for more faults to be treated without resorting to Safe Mode, as well as allowing for fail-operational behavior during critical events. As a result of its privileged position at the top of the SPIDER food chain, FPE has sufficient visibility and authority to handle a restart of the MOI burn after fault recovery has completed at lower levels.

6. Looking Back, Looking Ahead

A planetary spacecraft's requirements for high levels of autonomy must contend in practice with a natural programmatic reluctance to accept (much less trust) full spacecraft autonomy. Given a tradeoff between the risk of potential logic flaws *vs.* the risk of insufficient autonomy, programs to date have decided to worry much more about the former than the latter. Recent missions, though, have shown a decided trend of increased demand for (as well as comfort with) spacecraft autonomy.

We have had the rare opportunity to build a series of spacecraft with similar avionics – giving us the ability to evolve the SPIDER architecture in a low-risk, incremental way. The resulting maturity of the SPIDER architecture allows programs to accommodate ever-increasing demands for autonomy without taking on undue development and logic risk. SPIDER's layered architecture allows us, almost literally, to build on what we've done in the past. Progressively "standardizing" logic one layer at a time also allows development effort to be focused on the refinement of the more complex (and ultimately capable) logic at the upper architecture levels (see Figure 6). A number of interesting options for further development of SPIDER's system level logic present themselves.

In the course of development of spacecraft autonomy, one of the most promising sources of design & development inspiration is often overlooked – namely, that embodied in the evolution of the human brain. Paul MacLean originated one of the more influential, as well as powerful, models of the human brain [5]. MacLean's Triune Brain model defines three distinct systems within the brain; the structure and chemistry of each segment reflects developments identified, respectively, with reptiles, early mammals, and late mammals. Under MacLean's model, the human brain really consists of three brains in one – each represents essentially an evolutionary stratum that has formed on the older layer below it.

MacLean's research suggests that most human behaviors are the result of complex interactions between these three systems. In most cases, upper levels predominate; when lower levels' needs are not met, though, the brain as a whole may downshift to basic, survival thinking. There are obvious parallels between this model and Maslow's hierarchy of needs [6], as well as between the model and Brooks' subsumption architecture. There is also an obvious parallel between this model and spacecraft behavior in the face of a major fault (shifting from a normal, "high-level" mode of operations to a "Safe Mode" or other contingency mode). I'll leave the pursuit of these tangents to the reader.

When it comes to applying the lessons of biology to system design, though, two primary points jump out of this:

> 1 – Microgenesis (building a beast one thin layer at a time) is a low risk and highly effective development approach.
>
> 2 – A hierarchical organization of functionality, with the lowest complexity (regulation of simpler functions) on the bottom, and the highest complexity (mediation of more complex functions) on top is a robust and proven architectural scheme.

Applying these concepts to the development of increased levels of spacecraft autonomy would essentially result in restating them as programmatic directives:

> 1 – An evolutionary perspective on the development of increased autonomy, not repeating the mistake of attempting to build some ideal autonomous system in a single leap on a single program.
>
> 2 – The implementation of increased autonomy via incremental layering of more capable (and likely

more complex) functionality "on top of" existing functionality.

Over the long run, this results in the phased introduction of more-sophisticated functions over time – adding the easier / simpler things first, and building on that foundation in future missions (an extended version of SPIDER experience to date). Following such a path, more capable non-reactive functionality can in the future be overlaid on the existing SPIDER architecture to meet a future program's requirements – either supplementing or replacing logic in MRO's system level of SPIDER.

7. SUMMARY

Development and flight experience with SPIDER's distributed emergent architecture have shown it to provide most of the benefits associated with monolithic AI fault protection architectures, while being easier to completely test, and having lower reuse costs than is the case for those schemes. Experience over a number of missions has shown that the architecture can be readily and easily adapted to a wide variety of spacecraft and mission designs. Furthermore, the distributed and layered nature of the SPIDER architecture eases the implementation of even substantial enhancements with time, as it allows for the low-risk incremental introduction of new and improved functionality.

ACKNOWLEDGEMENTS

This work was supported by the Mars Reconnaissance Orbiter program under contract to the Jet Propulsion Laboratory, contract number 1234906. In addition, the author would like to thank Jason Dates for his untiring assistance in implementing this architecture in flight software over the past 5 years.

REFERENCES

[1] E.H. Seale, "SPIDER: A Simple Emergent System Architecture for Autonomous Spacecraft Fault Protection," *AIAA Space 2001 Conference Proceedings*, AIAA 2001-4685, August 2001.

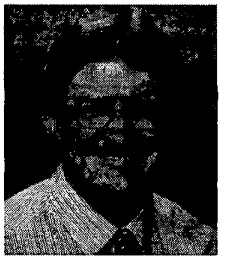
[2] William Stallings, *Data and Computer Communications, Second Edition*, Macmillan Publishing, 1988.

[3] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, RA-2, 14-23, April 1986.

[4] E. H. Seale, "Harnessing a SPIDER: Fault Protection, Software Reuse, and the Genesis Spacecraft," *AAS Guidance & Control Conference Proceedings*, AAS 2002-013, February 2002.

[5] Paul MacLean, *The Triune Brain in Evolution*, Plenum Publishing, 1990.

[6] Abraham H. Maslow *Toward a Psychology of Being*, D. Van Nostrand Company, 1968.



Eric Seale is the Fault Protection Architect for the Mars Reconnaissance Orbiter spacecraft. He invented and has led further development of the SPIDER architecture over a series of planetary spacecraft missions. He has worked in various capacities on the development and operation of seven planetary spacecraft (starting with Magellan) in his time at Lockheed Martin (formerly Martin Marietta). He has a BSAE and MSAE from the University of Colorado – Boulder.