

# Planning for the V&V of Infused Software Technologies for the Mars Science Laboratory Mission<sup>1</sup>

Martin S. Feather, Lorraine M. Fesq, Michel D. Ingham, Suzanne L. Klein  
Jet Propulsion Laboratory  
California Institute of Technology  
4800 Oak Grove Dr  
Pasadena, CA 91109

{Martin.S.Feather, Lorraine.M.Fesq, Michel.D.Ingham, Suzanne.L.Klein}@jpl.nasa.gov

Stacy D. Nelson  
NASA Ames Research Center  
Moffett Field, CA 94035  
& Nelson Consulting  
2006 Hwy 101 #135, Florence, OR 97439  
nelsonconsult@aol.com

*Abstract*— NASA’s Mars Science Laboratory (MSL) rover mission is planning to make use of advanced software technologies in order to support fulfillment of its ambitious science objectives. The mission plans to adopt the Mission Data System (MDS) as the mission software architecture, and plans to make significant use of on-board autonomous capabilities (e.g., path planning, obstacle avoidance) for the rover software. The use of advanced software technologies embedded in an advanced mission software architecture represents a turning point in software for space missions. While prior flight experiments (notably the Deep Space One Remote Agent Experiment) have successfully demonstrated aspects of autonomy enabled by advanced software technologies, and MDS has been tested in ground experiments (e.g., on-earth tests on rover hardware), MSL will be the first science mission to rely on this combination. The success of the MSL mission is predicated upon our ability to adequately verify and validate the advanced software technologies, the MDS architectural elements, and the integrated system as a whole. Because MSL is proposing a shift from traditional approaches to flight software, approaches to verification and validation (V&V) require scrutiny to determine whether traditional methods are adequate, and where they need adjustment and/or augmentation to handle the new challenges. This paper presents a study of the V&V needs and opportunities associated with MSL’s novel approach to mission software, and provides an assessment of V&V techniques, both current and emerging, vis-à-vis their adequacy and suitability for V&V of the MSL rover software.

## TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. MISSION RISKS DUE TO SOFTWARE.....	2
3. RISK MITIGATION: V&V APPROACHES .....	4
4. RISK MITIGATION: THE MDS APPROACH .....	9
5. FUTURE WORK.....	11
ACKNOWLEDGEMENT.....	13
REFERENCES .....	13
BIOGRAPHY.....	14
APPENDIX: SOFTWARE DEFECTS.....	15

## 1. INTRODUCTION

NASA’s Mars Science Laboratory (MSL) mission, currently being designed at the Jet Propulsion Laboratory (JPL), is a rover-based exploration mission scheduled for launch in 2009. MSL’s science and engineering goals expand on previous Mars rover missions, and include enhanced capabilities such as hazard detection and avoidance, goal-based commanding, on-board goal elaboration, for the purposes of long-distance traverse and science instrument placement. These enhanced capabilities take shape through the use of advanced software technologies and a state-of-the-art embedded software architecture, the Mission Data System (MDS). In addition to providing a modular component-based architecture, MDS provides MSL with a systematic process for capturing requirements, an approach to modeling states of the system and the relationships between states, and a mechanism for goal-based control of the spacecraft.

<sup>1</sup> 0-7803-8155-6/04/\$17.00©2004 IEEE

This paper describes our approach to addressing the V&V challenges introduced by the use of MDS and other advanced software technologies on a rover. In Section 2 we analyze the problem addressed through software V&V, namely, the prevention and discovery of software defects. Defects can be viewed as a risk to the mission, and we begin by understanding and assessing these risks versus other software risks. We present a comprehensive list of all software defects relevant to rover software, that is, embedded, real-time, mission-critical software. We also discuss limitations in traditional mission software design approaches.

In Section 3, we outline the traditional approach to verifying and validating mission software. We then describe surveys we have performed of the V&V field to identify additional tools and processes that are available to prevent or detect software defects. We also discuss state-of-the-art V&V tools that could help address the V&V challenges and opportunities posed by the use of advanced software technologies. Finally, we identify V&V techniques used by other industries that develop and deploy embedded, real-time, mission-critical software.

Section 4 introduces the MDS concepts that address the limitations in traditional mission software described in Section 2. We also examine the V&V needs and opportunities that are introduced by the use of MDS. Section 5 looks to the future and describes how we will seek to combine all of the information described in the earlier sections to perform cost-risk trade analyses using JPL's Defect Detection and Prevention (DDP) tool. This effort will allow the MSL project to assess the most cost-effective V&V techniques to apply to the rover software, in order to accomplish mitigating the greatest number of software defects, given its available resources.

## 2. MISSION RISKS DUE TO SOFTWARE

Projects have long considered potential hardware problems in their evaluation of project risks and have developed formal techniques, such as fault tree analysis, to aid in their evaluation. Examining software risks is relatively new and far less mature. Software risk examination is starting to come of age, however, as shown by a wealth of recent surveys that have been performed to determine the most prevalent risks in the software industry. For example, DeMarco has published a list of five core risks that are so prevalent throughout the software industry that they affect virtually every project that requires software [1]. We reproduce these core risks in Table 1.

The surprising thing about DeMarco's core software risks is how little they seem to relate to what most software implementers would consider software risks – namely potential defects in the software. Three of the core risks –

(faulty schedule/budget, employee turnover, and productivity) are not technical in nature and the other two risks appear to be associated mainly with early stage, high-level software technical development. But, in fact, defects in software “bubble up,” affecting schedule and budget (Risk 1). The number of software defects is a reflection of the number of employees involved, their degree of expertise, and their productivity (Risks 3 & 5). Software requirements inflation (Risk 2) and software specification breakdown (Risk 4) are influenced by the need to avoid potential software implementation problems that are already apparent in requirements and design. In essence, potential software defects are to a considerable extent at the heart of the core risks, and, as such, may be thought of as fundamental risks to any mission. For this reason, we have chosen to treat potential defects as a subcategory of software risk to which risk management techniques should be applied and to which proactive solutions (mitigations and preventions) need to be found. The next sub-section describes our efforts to capture and document a comprehensive view of software defects that enter into mission software. The following subsection discusses some limitations of the traditional approach to mission software; these limitations contribute to the level of risk associated with the software, and can be a source of potential defects.

**Table 1. Core project risks due to software [1]**

	Core Risks	Area	Description
1	Faulty Software Schedule or Budget	Planning	An error in the schedule or budget rather than in the software development and testing process; usually an underestimate of the cost and schedule.
2	Software Requirements Inflation	Requirement & Design	Significant increase in the number of software requirements over time.
3	Software Employee Turnover	Staffing	High number of programmers leaving a project – often not considered in budget estimates.
4	Software Specification Breakdown	Planning	A breakdown in negotiation; the majority of stakeholders can't agree on what they are building.

5	Poor Productivity	Experience & Teaming	Under performance due to an insufficient number of programmers with the required skills and experience.
---	-------------------	----------------------	---------------------------------------------------------------------------------------------------------

### Limitations of Traditional Mission Software

In this section, we describe some of the limitations of the traditional approach to mission software design that can cause defects and contribute to the level of risk associated with the software.

### Generic Software Defects

In order to determine which V&V techniques are most effective, it is important to understand what kinds of software defects can occur at each phase of the software development life cycle. The software life cycle "V-chart" (shown in Figure 1) identifies the phases typically found in developing software, as well as the concepts of verification and validation with respect to these phases. This lifecycle model is generic and can be applied to various software development methodologies including the iterative incremental methodology used for MDS.

Using the life cycle as a foundation, a comprehensive list of common software defects organized by life cycle phase was compiled based on an extensive literature search, the combined experience of the authors, and a survey of experts at NASA and the aerospace industry [2].

The software defects lists contain over 200 defects, organized into life cycle phases. Examples, one from each phase, are included in the table in the Appendix.

### Subsystem-based encapsulation

In traditional approaches to designing mission software, code is "compartmentalized" in various ways, e.g. flight vs. ground vs. test, and by subsystem (power, thermal, navigation, etc). As a result, each subsystem's software engineering and programming teams tend to apply customized solutions to problems in their subsystem, leading to minimal amounts of software reuse across the different subsystems. Furthermore, in addition to the problem of inefficiency, this approach leads to increased interface complexity between subsystems, as many important mission considerations, such as onboard resource limitations, introduce coupling across subsystem boundaries [3]. Getting the interfaces right generally requires many iterations on the part of multiple subsystem software teams.

### Lack of Explicit Representation of System Knowledge

In traditional approaches to embedded software, and spacecraft mission software in particular, programs are written such that they prescribe the desired state evolution implicitly, through low-level commands to actuators and

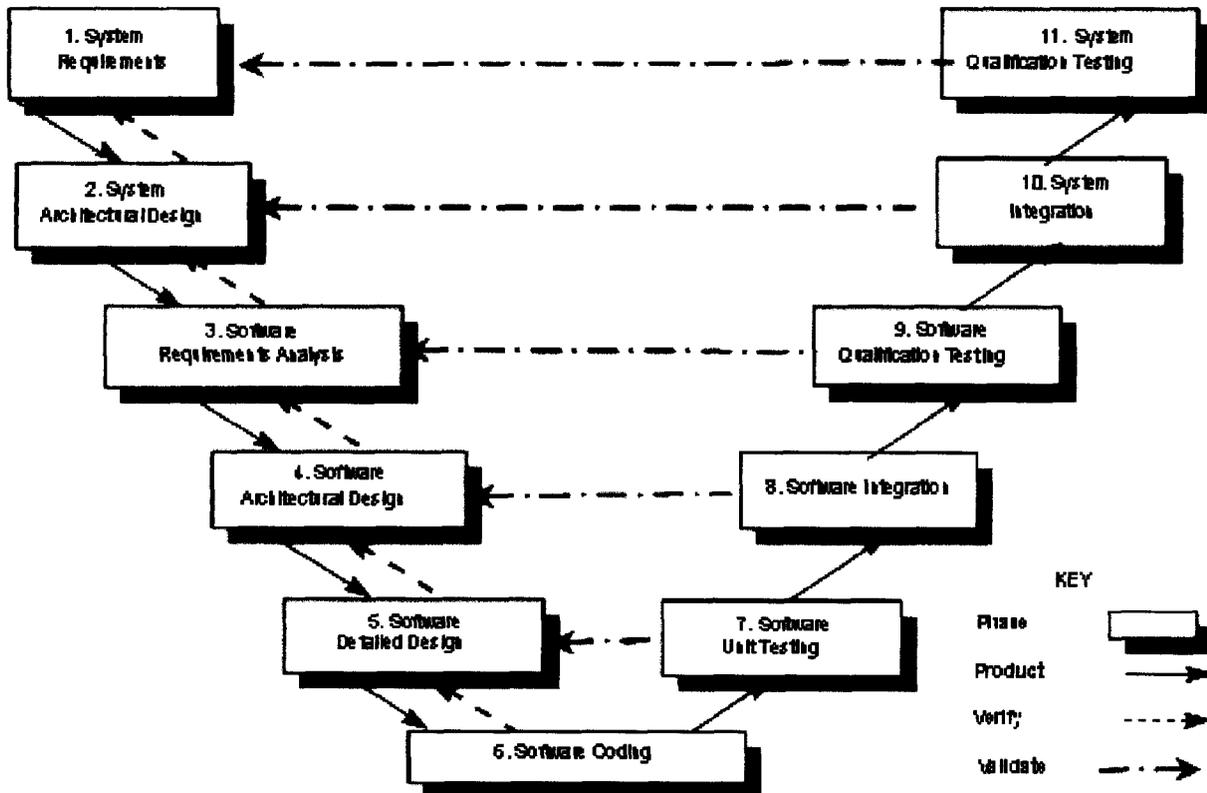


Figure 1. Software Development Life Cycle

references to sensors. Their implicit consideration of state makes these programs hard to encode, requiring translation of the system behavior as understood by the systems engineer into the flight code as written by the software engineer. This translation process also results in a verification challenge, by opening up the possibility for inconsistencies between the system specification and the flight code, and by making these inconsistencies particularly hard to identify during code reviews.

### **Open-loop Command Sequences**

The traditional approach to controlling spacecraft is through nominal command sequences, which are time-tagged lists of commands and macros. These sequences specify actions down to the level of detailed hardware commands, whose effects can potentially be felt across spacecraft subsystem boundaries. Once uploaded to the onboard flight computer, these sequences are executed by simply issuing the appropriate commands at their specified times, in open-loop fashion [4]. These low-level command sequences do not capture any notion of the operator's *intent*, resulting in non-robust behavior in the face of off-nominal execution. Thus, to build confidence that the spacecraft will behave as predicted, engineers must perform careful modeling and extensive testing of these sequences on sophisticated hardware-in-the-loop simulation testbeds, as well as on the flight hardware itself, prior to launch.

### **Mixed Estimation and Control**

In traditional mission software, control logic is frequently intermingled with state determination logic. This can lead to inefficiency and unnecessary redundancy, where controllers in multiple subsystems use multiple instances of the same estimation algorithms. More importantly, this approach allows for the possibility that different estimates of state could be used by different controllers in the system, potentially resulting in dangerous inconsistencies among control actions that should be correlated.

### **Fault Protection as an Add-On Capability**

For flight activities where more flexible event-driven execution is necessary, time-triggered command sequences are insufficient due to their inability to represent conditional response. Thus, timed command sequences are traditionally augmented with rule-based engines [5] or hard-coded state machines [6] running as concurrent processes, which periodically check the available onboard measurements for satisfaction of a trigger condition and issue predetermined commands or macros in response. These conditional execution mechanisms are also used for onboard fault protection. Off-nominal behavior is usually handled by putting the spacecraft into "safe mode," in which all non-essential functions are disabled and the spacecraft calls for help from the ground operators. These interventions can be costly, both in terms of ground

operations costs and the science opportunities lost while in safe mode.

During mission-critical activities, such as planetary fly-bys, orbital insertion and entry, descent and landing, the act of putting a spacecraft into safe mode would result in loss of the mission. In such situations, standard fault responses are usually disabled, and fault protection is provided by highly specialized dedicated sequences, which tend to be significantly more complex than non-critical sequences. Generating and testing these critical sequences is an extremely expensive process; this cost can dominate the mission operations budget even though these sequences represent a small fraction of the overall mission duration [7]. Furthermore, at execution time, the complexity problem is exacerbated by the very short time available for recovery from anomalies, and can result in "brittle," non-robust behavior in unexpected off-nominal conditions.

In this section, we have discussed two major sources of risk to a mission introduced by software; namely, software defects that can be introduced throughout any of the lifecycle phases, and limitations in the traditional approach to designing mission software. In the next two sections, we turn our attention to mitigation approaches that address these risks.

## **3. RISK MITIGATION: V&V APPROACHES**

Two approaches to V&V can be employed to mitigate the risks introduced by software defects: prevention, that is, providing mechanisms and processes that will prohibit defects from being introduced into the software, and detection, or the uncovering of defects that have been introduced. In this section, we survey the traditional V&V approaches used by the aerospace industry to prevent and detect software defects. We then provide a view into an abundance of tools and techniques that are available for use today, but not yet adopted as mainstream practices. We then describe the results of our survey of V&V practices followed outside of the aerospace industry by industries that develop similar software; i.e., real-time, embedded mission critical software. Finally, we look toward the future at a number of promising V&V tools on the horizon.

### *Traditional V&V Techniques*

Traditional V&V techniques are based upon methods such as fault tree analyses (FTA), failure modes, effects and criticality analyses (FMECA), design reviews, code reviews, and, above all, test plans and procedures whose aim is to verify and validate software requirements through the use of one of four methods:

- Comparison between actual and expected results
- Demonstration

- Inspection
- Analysis

of automated tools.

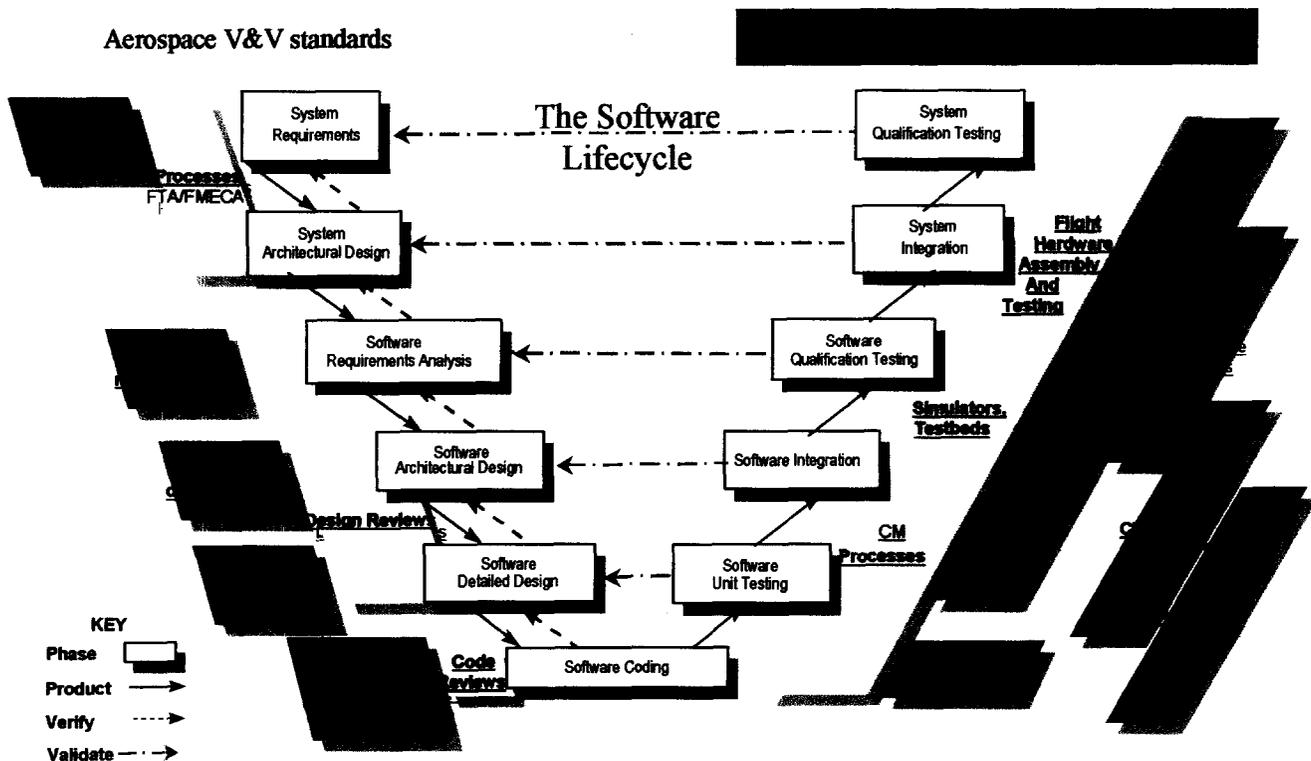


Figure 2. V&V techniques aligned with software lifecycle phases. Inner columns (light blue) show current aerospace practices; outer columns (green) highlight additional, commercially available techniques.

Comparison of expected with actual results is most often used for determining whether performance requirements have been met. Demonstration is typically used when the results are visually obvious; for instance, when verifying that the layout of a GUI meets specifications. Software inspections are strict and close examinations conducted on requirements, specifications, architectures, designs, code, test plans and procedures, and other artifacts. They check primarily for completeness, correctness, and consistency. Analysis is most often used when large amounts of data must be scrutinized before it is possible to verify that complex algorithms are being implemented correctly.

Informal verification is traditionally done during coding and unit testing. Formal qualification V&V is performed during software and system integration. The test plans and procedures for unit testing are typically drawn up and executed by the software implementers themselves and the tests are witnessed and the results verified, validated, and recorded by Quality Assurance (QA) personnel based upon the test plan. Formal V&V testing is traditionally done by independent testers (persons other than the implementers), and witnessed, recorded, and the results verified and validated by QA. Traditional V&V testing employs largely manual procedures with little or no use of techniques other than the four methods already indicated and with little use

#### Currently available V&V Tools and Techniques

A weakness of traditional V&V is that it makes little use of new, currently available techniques or of existing automated tools for verification, but rather relies on techniques that have been around for over 20 years and on occasion, “home grown” tools created by testers to verify specific software. The tendency to continue with the status quo is also surprising since a casual search of the web and of NASA sites reveals a number of techniques and currently available commercial tools that could be very useful. This situation motivates our current effort to explore the benefits that new and emerging tools could provide to the existing V&V process. Figure 2 depicts the traditional software lifecycle V-chart that was introduced in Figure 1, annotated with a selection of traditional V&V approaches currently used in the aerospace industry, and augmented with additional techniques that are either currently available or under development.

A number of useful commercial grade tools are presently available that are capable of automatically verifying design architecture or of scouring unit and integrated code for defects or both.

Among currently available commercial grade tools, most can be categorized by identifying the phase of the software life-cycle they address and the techniques they use (see Table 2).

**Table 2. Currently available V&V techniques categorized by software lifecycle phase**

LIFE CYCLE PHASE	V&V TECHNIQUE
System Requirements	Requirements Consistency Analysis
System Design	Requirements Consistency Analysis
Software Requirements	Requirements Modeling/Analysis
Software Architectural Design	Architectural Design Modeling/Analysis
Software Design	Detailed Design Verification
Software Coding	Auto-Code Generation
Software Unit Testing	Auto-Test Case Generation

We provide a brief overview of these techniques below.

**Requirements Consistency Analysis.** Tools that use this technique check requirements definitions. They are capable of performing requirement consistency checks that uncover problems such as subtle partitioning errors, logic errors, algorithmic errors, and data dictionary variable definition defects.

**Requirements Modeling/Analysis.** These tools perform either static or dynamic analysis at the requirements level, and usually include requirement model animation and requirements model property checking.

**Architectural Design Modeling/Analysis.** These tools are capable of checking the architectural design for consistency and can perform behavior checking and reachability analysis via simulation.

**Detailed Design Verification.** Few of the detailed design verification tools verify exactly the same detailed design attributes. And none are complete design verifiers. As a result, to cover detailed design verification and validation, a suite of design verification tool may be required. Capabilities of the design verification tools include the following features:

- Verification of safety and “liveness” properties
- Verification of user correctness requirements
- Verification of parameters
- Validation of protocol systems (e.g., communication)
- Validation of the design model during simulated execution

- Limited performance verification support
- Traces of logical design errors in distributed systems design
- Reachability analysis at a more refined level
- Detection of race conditions and deadlocks
- Detection of timing violations
- Detection of concurrency errors

**Auto-Code Generation.** The objective of these tools is not to simply check the design or to check for defects that may have occurred, but to prevent defects in the first place. The approach is to change the developer's point of view from writing code, or assembling software components, to modeling the system to be built. The design then is implemented through automatically generated code. These tools operate on the premise that if the design is correct, and the design can be automatically verified, then the generated code will defect-free. These tools tend to have the following additional features:

- Requirements traceability
- Model Checking (checks for completeness)
- Validation through design-level debugging

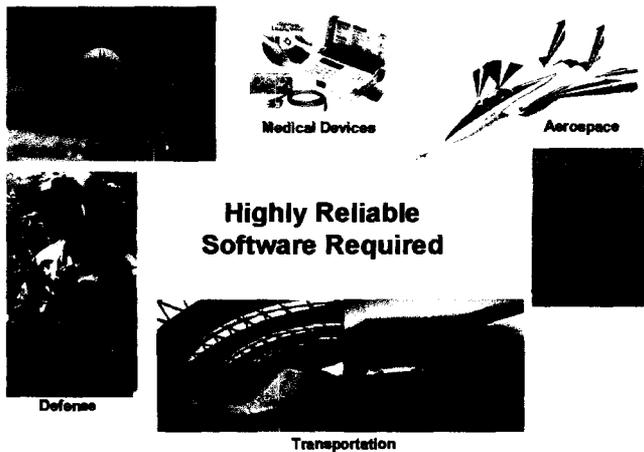
**Auto-Test Case Generation.** Test code generation tools generate test cases based upon minimum, maximum and mid-range values. Most include a range of additional features such as:

- Automatic generation and compilation of test stubs and driver programs
- Interactive point-and-click and script-generated test case construction
- Test case modification and re-execution without re-compilation
- Automatic regression testing
- Standards compliant test report generation
- Basis path analysis and cyclomatic complexity
- Test execution on both host and embedded target development systems

After compiling a comprehensive list of V&V tools, techniques and processes available, we examined them and identified the relevant defects that were addressed by each. A sample of this cross-correlation between each identified software risk (i.e., the defect) and the possible ways to mitigate these risks (i.e., the V&V tools, techniques and processes) is shown in the Appendix.

*Survey of V&V techniques for Mission critical Software*

Software plays an increasingly crucial role in all aspects of modern life from flight to driving to power generation to defense to medical devices, etc (see Figure 3). Therefore, we must be able to trust that software is reliable and will act according to intended design.



**Figure 3: Industries Requiring Highly Reliable Software**

For the purposes of this paper, the following definitions apply [8]:

- **Mission-critical software:** loss of software capability can lead to possible reduction in mission effectiveness
- **Safety-critical software:** software failure or design error could jeopardize human life

Currently, key facets of reliable software depend upon the thoroughness of the software development process, i.e., the software life cycle. Software life cycles vary across industries and across projects within the same industry, but the overall idea is the same: assemble a team of competent software developers to determine the intended software behaviors (requirements), develop code to accomplish these behaviors, then submit the requirements and code to a team of V&V specialists who check them via a variety of techniques including testing, simulation, and formal methods.

Finally, the code is evaluated by an independent team of software development experts who review the software during formal review sessions to decide whether it meets its objectives. If the software is deemed safety-critical (has potential for loss of life), the reviewers generally ask themselves whether they would be willing to use the software. They consider questions such as: Would I risk my life to fly on an airplane with this digital flight control system? Would I drive an automobile with this anti-lock brake design? If the answer is yes then the software is submitted for system certification. Generally, software does not receive a stand-alone certification. Only integrated systems including hardware and software are certified.

If the software is mission-critical (potential for loss of spacecraft, mission data, etc) then reviewers consider whether test results indicate a significant likelihood of mission success. If yes, then the software is approved for

implementation. Approving software is a rigorous process. To make the approval decision, reviewers must believe, based on the facts presented, that the software has been thoroughly checked.

This section summarizes key processes used across industry and government in the United States and Europe to determine whether software is safe and reliable. These processes reveal the following common themes:

- Standards exist, containing lessons learned from prior development projects to promote safer, more reliable software.
- Review boards make decisions about the software safety and reliability based on trust in the development team, demonstration of key software capabilities in high-fidelity simulators and rigorous and thorough V&V (including testing).
- Software sometimes fails despite best efforts to verify and validate capabilities.
- Formal methods can uncover hard-to-find errors like race conditions.
- Software reliability metrics generally consist of keeping track of the number of issues (bugs). For example, the Space Shuttle IV&V team computes the following metrics [9]:
  - Number of Issue Tracking Reports (ITRs) per software release;
  - Number of Days an ITR remained open – a measure of complexity;
  - Severity of Open and Closed ITRs.

The following techniques have proven to be necessary for developing safety-critical software across all industries:

- Testing based on key scenarios designed to check that software works as intended.
- Simulation beginning on low fidelity testbeds and occurring on higher fidelity testbeds until final tests occur on the actual hardware. This promotes cost containment by allowing developers to find and correct anomalies early in development before exposing expensive hardware to possible failures.
- Demonstrations of working software to qualified review boards in accordance with industry standards. Certification or approval by review boards is consistent across all industries. Therefore, individual projects succeed or fail based on the aptitude of these review boards.

While ANSI/IEEE standards [10] contain a plethora of metrics, review boards in the United States currently

emphasize the following criteria to determine whether software is safe and reliable:

- Test results
- Demonstration of software in high-fidelity test beds
- Trust in the experience and expertise of the development and verification/validation teams

Review boards in Europe and Canada supplement reliance upon experienced teams and demonstrations with effective use of formal methods to prove software correctness properties.

Unfortunately, software errors still occur. The following additional techniques (in alphabetical order) were used across at least three industries. The industries are noted in parentheses:

- Formal Methods (Canada and European nuclear power and transportation);
- Information Flow Analysis (aerospace, defense and nuclear power);
- Partitioning (aerospace, nuclear power and transportation);
- Risk/hazard assessment based on severity and likelihood (aerospace, defense and transportation).

The aerospace, nuclear power and transportation industries rely upon Fault Detection and Diagnosis as a safety net to respond in the event of an unforeseen error resulting from either V&V oversight or unexpected environmental conditions.

To supplement the traditional life cycle, the FAA and SAE recommend building safety or reliability cases (justification) as part of software development.

As software becomes more sophisticated, more software failures are likely. The following section describes advanced techniques that are emerging and have been used in experiments (NASA, industry and academia) to improve V&V of highly reliable software with promising results.

#### *Emerging V&V Techniques*

Emerging V&V Techniques are largely based on Formal Methods [11], [12]. The term “Formal Methods” refers to the use of techniques from logic and discrete mathematics (discrete structures like set theory, automata theory, formal logic as opposed to continuous mathematics like calculus) in the specification, design and construction of computer systems and software. The objective of Formal Methods as a V&V technique is to reduce reliance on human intuition and judgment by providing more objective and repeatable tests.

For discussion purposes, Formal Methods are often

categorized into runtime monitoring, static analysis, model checking and theorem proving. An overview of each category follows, along with associated benefits and challenges:

**Runtime Monitoring** - evaluating code while it runs or scrutinizing the artifacts (event logs, etc) of running code.

#### Benefits

- Requires a relatively small incremental effort over traditional testing.
- Combines the ease of testing with the power of Formal Methods.
- Can locate difficult to find error potential for problems that test engineers may overlook.

#### Challenges

- Logic-based monitoring can add overhead to the normal execution of programs.
- While detecting difficult-to-find errors, error pattern runtime analysis can detect problems that do not exist (false positives).
- Runtime monitoring observes the current program execution, but does not observe all possible runs so coverage is limited to actual program execution.

**Static Analysis** - detects runtime errors and unpredictable code constructs without executing code.

#### Benefits

- Verification can begin earlier in the software life cycle resulting in early detection/resolution of problems and reduction in development cost.

#### Challenges

- The biggest challenge for Static Analysis is generation of false positives. However, new tools use statistical methods to reduce the number of false positives. An analogy can be made between the results of static analysis and the results of a compiler where subsequent errors result from an initial or upstream error. Correcting the initial error can eliminate some false positives.

**Model Checking** - automated technique for verifying finite state concurrent systems. The model checker evaluates the model by beginning with the initial states and repeatedly applying transitions to reach all possible states.

#### Benefits

- Fast, automated method for exploring all relevant execution paths of non-deterministic systems. This is very important because it is virtually impossible for humans to conceive of every test scenario

required to verify a non-deterministic system in a plausible time frame for software development.

- Can backtrack to explore alternative paths from a common intermediate state, avoiding the costly reset between tests required in traditional scenario-based testing.
- Detects problems in the early stages of development, thereby greatly reducing overall development costs.

#### Challenges

- Models must be translated into special model checking language.
- A model checker can run out of memory before exploring the entire state space of the program.

**Theorem Proving** – use of logical induction over the execution steps of the program to prove system requirements. In other words, system requirements can be captured in complex mathematical equations and solved by verification experts. Solving these equations proves that the system is accurate.

#### Benefits

- Can use the full power of mathematical logic to analyze and prove properties of any design.

#### Challenges

- Requires significant effort and expertise. Currently suitable only for analysis of small-scale designs by verification experts.

### 4. RISK MITIGATION: THE MDS APPROACH

As mentioned above, the Mission Data System (MDS) is an embedded software architecture, currently under development at NASA JPL. Its overarching goal is to provide a multi-mission information and control architecture for the next generation of robotic exploration spacecraft, that will be used in all aspects of a mission: from development and testing to flight and ground operations. In the process of achieving this ambitious goal, the MDS team has rethought the traditional mission software lifecycle, and has adopted a vision that acknowledges and leverages the intimate coupling between software and systems engineering: “Software is part of and contributes substantially to a new systems engineering approach that seamlessly spans the entire project breadth and life cycle.”[13]

The central themes of the MDS approach address many of the limitations of traditional mission software designs described earlier:

- **State and models as the architectural foundations** – MDS is a state-based architecture, where state is defined as the momentary condition of a dynamic system. State is accessible in a uniform way through state variables, instead of through local variables as in traditional flight software. MDS emphasizes the separation of application-specific knowledge, in the form of *models* that describe how *state* evolves, from reusable general-purpose code that operate on the models to track and control state. MDS models can be expressed in any convenient form, e.g., tables, functions, rules, state machines, etc. The novelty in this approach is that models are used explicitly, rather than being “hidden” in the details of the flight code. This leads to easier portability from mission to mission, as only the models need to be updated with domain-specific knowledge.
- **Modular component framework** – MDS strives to identify common problems across embedded software applications and provide common solutions, in the form of shared core architecture components, such as estimators, controllers and schedulers. This decreases the amount of redundant (and potentially conflicting) code written, and promotes consistency through common resource coordination services provided by the architecture. A simplified interpretation of the MDS architecture is shown in Figure 4.

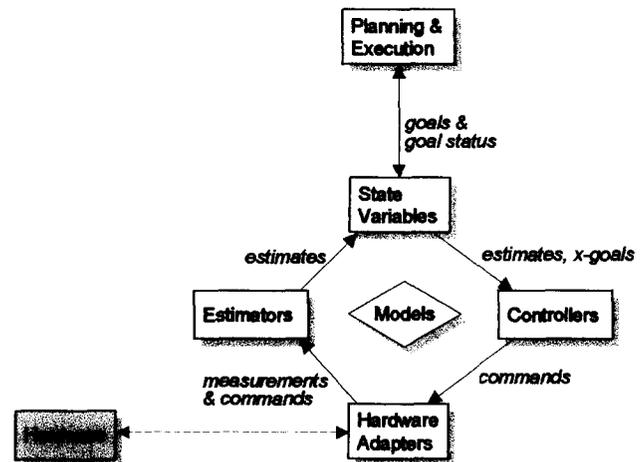


Figure 4. MDS Architecture

- **Goal-directed closed-loop operation** – Instead of issuing low-level open-loop commands, control sequences issue goals that indicate intent in the form of desired state. Goals are easier to specify than the actions needed to achieve them, and result in more compact specifications of desired behavior. Furthermore, goal-directed operation goes hand-in-hand with closed-loop control, because goals can be thought of as set points for onboard controllers, which are then given the latitude to decide how best to achieve the goals.

- *Separation of State Determination from State Control* – Unlike traditional approaches to embedded software, in which control logic is intermingled with state determination logic, MDS advocates making a clear separation between these two key functions, which are coupled solely through state variables. Taking this approach allows state knowledge to be updated in a unified, consistent way, for use by any control function in the architecture. For instance, multiple controller components might need to access the same state variable. Keeping the state determination and state control functions separate ensures that all active controllers use a consistent estimate of state. Furthermore, this type of increased modularity simplifies component-level testing of various state determination or control algorithms, and allows for minimally invasive upgrades to individual state determination and control components.
- *Integrated fault protection* – The goal-directed nature of the MDS paradigm leads to intrinsic “fault awareness” in the system; that is, fault detection, diagnosis, and recovery are an integral part of the design of the architecture. Intrinsic fault-awareness is enabled by providing the control system with knowledge of the operational intent (in the form of state goals), and the ability to derive appropriate actions by reasoning about the state of the system, instead of by edict [14]. In MDS, fault states are included in the behavior models and are treated just like any other state. Fault detection, diagnosis and recovery are thus performed in the same loops as the nominal estimation and control processes. Another key element of robust operations is the consideration of knowledge uncertainty. In the MDS framework, state knowledge uncertainty is tracked in an explicit way, within the state variables. MDS also provides the ability to issue goals on knowledge quality/certainty, a capability that is generally not built into traditional flight software architectures.

#### *V&V Needs and Opportunities of the Mission Data System*

Given that the MDS approach corresponds to a departure from traditional mission software design, it is necessary to consider how the use of MDS as the mission software architecture for MSL will impact our V&V approach. Are conventional V&V methods appropriate and sufficient for an MDS-based system? If not, what aspects of the MDS system require extensions or modifications to existing V&V methodologies?

Another important question that should be posed is: does the MDS framework enable or enhance the level of V&V that can be applied to the system? The use of explicit state-based models allows MDS to capture, in one place and

using a uniform representation, types of mission requirements that have traditionally been documented in a variety of documents using a variety of representations. Furthermore, it enables MDS to explicitly capture assumptions about system behavior that traditionally reside in the minds of systems engineers, and are captured only implicitly in the flight software. Thus, MDS provides the opportunity to formally validate and verify these heretofore informally-specified requirements and assumptions, through the application of V&V to these models.

We have performed an analysis to identify the V&V needs and opportunities that are particular to MDS, as adapted for the MSL rover mission. In this section, we describe four of the key V&V needs/opportunities that have emerged from our analysis.

#### 1) Verification of the MDS component architecture:

MDS defines a set of rules that specify how different types of components in the architecture (Figure 4) must be connected. Verification of the component architecture for a particular mission adaptation (e.g., the MSL adaptation) involves checking for compliance of the component connections with respect to the set of architecture rules. Given a specification of the rules in a machine-checkable form, and of the architecture as a formal model, this verification may be performed automatically using state-of-the-art theorem proving technology, for example.

#### 2) Validation of the State Effects Models:

MDS is based on a systems engineering approach called *State Analysis*, which provides a process for system modeling via state discovery. The primary product of the State Analysis process is a *State Effects Model*, which captures the physical model describing the evolution of each state in the system, as well as the physical relationships between different state variables in the system. The State Effects Model compiles information traditionally documented in a variety of systems engineering artifacts, including the Hardware Functional Requirements, the Failure Modes and Effects Analysis, the Scenario Description Document, and the Hardware-Software Interface Control Document. Information from the State Effects Model is used throughout the mission software, including in the elaborations of goals into subgoals on related states, and in the estimation and control algorithms. As an explicit, formal representation of this type of critical system information, the State Effects Model provides the opportunity to perform a more thorough validation of the physical assumptions embedded in the mission software. Validation of the State Effects Model consists of checking the model for completeness (e.g., has a relevant state been omitted? has a relevant influence relationship been omitted between states in the model?) and correctness (e.g., have we included an incorrect influence in the model? are there any undesired “loops” in the

influence model?). Whereas correctness checking may be enabled using some form of state-of-the-art model-checking, completeness checking most likely requires inspection by mission systems engineers.

### 3) V&V of Goal Elaborations and Goal Networks:

As discussed above, an MDS-based system is controlled by issuing goals on state variables, rather than the traditional approach of issuing low-level commands. A goal is formally defined as a *constraint on the values of a state variable over a time interval*. In MDS, each goal elaborates into a network of subgoals on related states, as specified in the State Effects Model. These *goal elaborations* form the building blocks for the goal-based “sequences” that are executed onboard the spacecraft, called *goal networks*. Currently, goal elaborations are explicitly specified by systems engineers, and translated into executable form using a Goal Elaboration Language (GEL). Goal elaborations must be both validated and verified. Validation of a goal elaboration consists of checking the correctness and completeness of its specification: the goal elaboration must include correct subgoals on all appropriate related states in the State Effects Model. Verifying a goal elaboration consists of determining whether the GEL code associated with a specified goal elaboration correctly maps to the systems engineer’s specification. Goal networks are generated by compiling and combining goal elaborations for multiple goals in support of the execution of an operational activity. Goal networks can either be pre-specified on the ground and uplinked to the spacecraft, or generated onboard from a set of stored goal elaborations. An operations scenario (or “plan”) expressed as a goal network must be checked for correctness, which would include checking that it does not specify any conflicting or adversely interacting goals.

### 4) Verification of the real-time execution mechanisms:

Given a plan corresponding to a set of executable goals, MDS executes the plan by dispatching the goals to various components (estimators and/or controllers) for achievement. To accommodate components that run at different frequencies and with different priorities, MDS’ real-time execution mechanism specifies a set of threads, with each thread corresponding to a particular frequency and priority. MDS employs a soft real-time component scheduler, to manage the execution of its multiple threads. In this design, uncertainty in the actual run-time of the software components can result in missed deadlines (where a thread fails to start by its specified start time or fails to end within its specified time window) and execution over-runs (where a component takes more CPU cycles to complete than the scheduler had anticipated). Verification and validation of MDS’ real-time execution mechanisms consists of: (1) checking that the behavior of the current component scheduler design is consistent with its stated requirement: *even in the presence of misses and over-runs,*

*computational integrity of the MDS execution mechanisms is maintained* (i.e., execution continues, even when miss and over-run events occur); (2) validating a variety of miss and over-run handling mechanisms; and (3) evaluating the performance of the MDS execution mechanisms with respect to the specification of “ideal” behavior: *even in the presence of misses and over-runs, no scheduled goals fail due to cycle slips.*

We have initiated the process of identifying applicable research and state-of-the-art V&V technologies, such as those described earlier in this section, that can be brought to bear on our list of MSL/MDS needs and opportunities.

## 5. FUTURE WORK

The objective of this paper is to serve as the first step in assessing software risk to MSL and to identify appropriate V&V techniques that can be used to prevent or mitigate these risks. The ultimate goal is the attainment of sufficient confidence in the reliability of the mission-critical software resulting from the development process. The selection of which V&V techniques to apply, where to apply them, and how thoroughly to apply them, is key. In this section, we propose using the JPL-developed Defect Detection and Prevention (DDP) process to perform trade analyses on the defects we have enumerated and the possible ways to mitigate them.

### *DDP approach to trade analyses for assessing software risk vs. mitigation cost*

Development of MSL’s software, like any other software development effort, is resource-constrained: schedule, budget and other resources (e.g., availability of personnel, of testbeds, of copies of the hardware that the software is to operate) are limited. Selection of V&V activities must therefore be done judiciously, taking into account both benefits and resource costs. For this purpose, our intent is to make use of DDP for cost- and benefit-based risk-informed decision-making. Custom software supports the application of DDP.

DDP manipulates three sets of information: “objectives” (the desired properties of the artifact being developed), “risks” (loosely speaking, all the things that would detract from attainment of objectives), and “mitigations” (options for preventing, reducing and/or mitigating the likelihood and/or severity of risks). DDP’s key ideas are to (1) explicitly relate risk to objectives (for evaluation of risk severity, and permitting tradeoff decisions among objectives), and (2) explicitly relate risk to the options for risk-reducing actions that might be taken during development and operation. DDP computes the benefit of a selection of “mitigations” in terms of expected attainment of objectives (taking into account the risk-reducing effects of those mitigations), the cost of a selection of mitigations,

and the repair costs for problems that they uncover (e.g., when applying a unit test, there is both the cost of the test itself, and the cost of repairing any bugs that test reveals). For more details, see [15]. The use of DDP for software assurance planning is discussed in [16].

For the purposes of V&V planning for MSL, software defects (examples of which are listed in the appendix) are represented as DDP's "risks", and the V&V activities are represented as DDP's "mitigations". The components of the mission software (e.g., Algorithms, Fault Protection) are represented as DDP's "objectives" (this allows us to capture the distinction between a defect in one component vs. in another).

The current status of the software defects information is that the links between items (e.g., relating the traditional V&V technique of "Boundary Value Analysis" to the defect "Array Overrun") indicate *some* connection, but not the *strength* of that connection. In order to make use of DDP's quantitative reasoning, we will need to augment these links with quantitative information (e.g., if "Boundary Value Analysis" is applied, *what proportion* of "Array Overrun" defects would it detect?). While some information of this form is available (e.g., data on the efficacy of inspections at uncovering defects, reported in [17]), we will likely have to rely extensively on experts' estimates. Nevertheless, past experience using DDP has shown that such estimate-based inputs can lead to valuable insights.

In the interim, we are able to use DDP to scrutinize the connectivity of the defects and V&V information. This allows us to quickly see which kinds of defects are addressed by which techniques, and vice-versa. The example in Fig. 5 shows a partial view representing the kinds of defects, alongside which are listed all the techniques applicable to preventing and/or detecting such defects.

For the purposes of illustrating DDP's quantitative capabilities we make the following simplifying assumptions:

- all defects are assumed to be equally detrimental to the components of mission software
- all components of mission software are assumed to be equally important (in practice we will need to assess these, e.g., we will likely weight highly software fault protection of the spacecraft hardware, since its correct performance is critical when called upon.)
- all risk-reducing actions are assumed to be equally effective, namely capable of preventing or detecting 50% of the kinds of defects to which they are related in our tables.

We stress that *these assumptions are for illustration only*, and not intended to represent our understanding of software risk and risk mitigation!

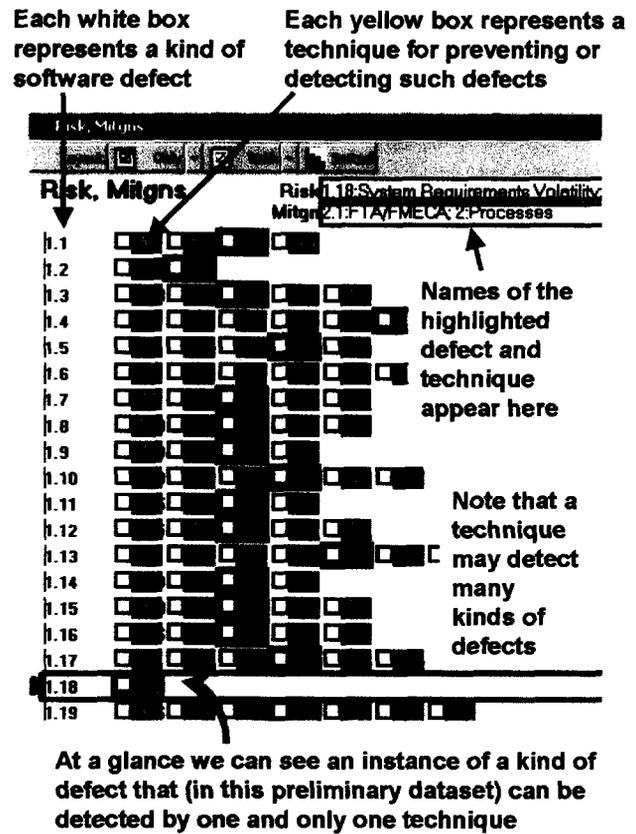


Figure 5. Defect-to-techniques view

Given these assumptions, Fig. 6 shows the DDP bar chart displaying overall risk status when all the traditional processes and V&V techniques that were described in the "Emerging V&V Techniques" sub-section, are being applied.

If, in addition, all the state-of-the-art V&V techniques are applied, then the risk levels drop further, as shown in Fig. 7, where the yellow portions indicate the drop in risk due to

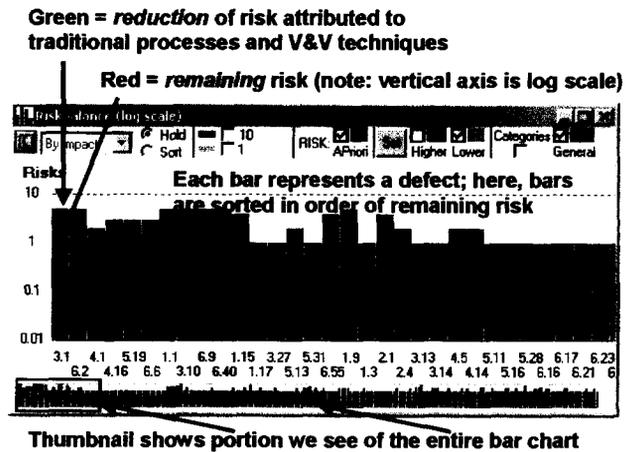


Figure 6. DDP bar chart of risk levels

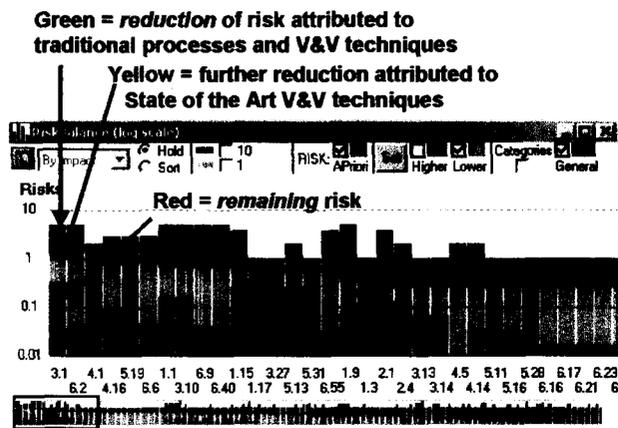


Figure 7. DDP bar chart of changed risk levels

those state-of-the-art techniques.

These kinds of views allow us to quickly scrutinize the net risk-reducing effects of candidate selections of techniques. Budget, schedule and personnel limits preclude the application of all possible state of the art V&V techniques to every portion of the MSL software, so it is important to be able to gauge their effectiveness when making this selection. MSL's needs, coupled with the work that has taken place to gather software defect information and relate it to existing and emerging V&V techniques, offer an interesting and challenging case study on which to apply this approach.

Once any new V&V technologies have been identified, we must address the issue of technology infusion. The MSL program is working on a formal approach to addressing this issue, and is publishing guidelines for infusion of new technologies that are in the pipeline for flight on MSL, as well as V&V technologies that will support acceptance of these new technologies.

## 5. CONCLUSIONS

NASA's MSL mission is planning to launch a robotic rover to Mars in 2009. The aspirations of this mission include adopting a new approach to architecting and developing mission software, and introducing capabilities such as goal-based commanding and long-range traverse that exceed previous Mars rovers. Along with these enhanced capabilities come V&V challenges that must be met in order to gain sufficient confidence in this software-centric interplanetary traveler. This paper reports on the beginning of our journey in charting MSL's V&V course. Risks to a mission due to software stem, in part, from software defects that are introduced at each phase of the software lifecycle, as well as limitations of traditional mission software designs. Current practices used by aerospace and other industries to verify and validate the software products tend to rely heavily on the use of testing to uncover defects that were introduced in earlier phases of the software lifecycle. Numerous additional tools and

techniques are either available today or will be ready for use in the near future to detect defects further upstream in the software development process and to prevent others from being introduced in the first place. While MSL's adoption of the MDS architecture squarely addresses and overcomes the inherent limitations in traditional mission software design identified in this paper, it introduces new V&V challenges that must be explored, understood, and addressed. In addition, the issue of risk mitigation due to software defects still must be systematically addressed and managed. JPL's DDP tool provides a mechanism to perform risk vs. cost trade analyses to assess and identify comprehensive, yet cost-effective plans to ensure the quality, integrity and robustness of MSL's rover software system.

## ACKNOWLEDGEMENT

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

## REFERENCES

- [1] T. DeMarco, "Waltzing With Bears: Managing Risk on Software Projects," Publisher: Dorset House; (March 2003)
- [2] "What Makes Code Review Trustworthy?" S. Nelson and J. Schumann, *Hawaii International Conference On System Sciences*, HICSS '04, January 5-8, 2004, Hawaii.
- [3] D. Dvorak, "Challenging encapsulation in the design of high-risk control systems", *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, 2002.
- [4] E. Gat and B. Pell, "Smart executives for autonomous spacecraft", *IEEE Intelligent Systems*, 13(5):56-61, 1998.
- [5] D. Watson, "Model-based autonomy in deep space missions", *IEEE Intelligent Systems*, 18(3):8-11, 2003.
- [6] N. Rouquette, P. Gluck, and R. Kanefsky, "The 13th technology of Deep Space One", *Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS'99)*, 1999.
- [7] E. Gat, "The MDS autonomous control architecture", *Proceedings of the Fourth World Automation Congress (WAC)*, International Symposium on Intelligent

Automation and Control (ISIAC'2000), 2000.

[8] Interview with Dale Mackall, Sr. Dryden Flight Research Center Verification and Validation engineer on January 16, 2003

[9] M. Zelkowitz & I. Rus, "Understanding IV&V in a Safety Critical and Complex Evolutionary Environment: The NASA Space Shuttle Program", Proceedings of the 23rd International Conference on Software Engineering, pp. 349-357, May 2001.

[10] ANSI 982.1-1989 and 982.2-1989: Measures to Produce Reliable Software

[11] NASA/CR-2002-211402 "V&V of Advanced Systems at NASA", Stacy Nelson and Charles Pecheur, NASA Ames Research Center, April 2002.

[12] "Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion" [NASA-GB-001-97], 1997.

[13] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software architecture themes in JPL's Mission Data System", Proceedings of the AIAA Guidance, Navigation, and Control Conference, 1999.

[14] R. Rasmussen, "Goal-based fault tolerance for space systems using the Mission Data System", Proceedings of the IEEE Aerospace Conference, Big Sky, MT, 2001.

[15] S.L. Cornford, M.S. Feather & K.A. Hicks; "DDP - A tool for life-cycle risk management", *Proceedings of the IEEE Aerospace Conference*; Big Sky, Montana, Mar 2001, pp. 441-451.

[16] M.S. Feather, B. Sigal, S.L. Cornford & P. Hutchinson: "Incorporating Cost-Benefit Analyses into Software Assurance Planning", *Proceedings of the 26th IEEE/NASA Software Engineering Workshop*; Greenbelt, Maryland, Nov 2001. IEEE Computer Society.

[17] Shull, F., Basili, V.R., Boehm, B., Brown, A.W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M.V., "What We Have Learned About Fighting Defects", *Proceedings of the 8<sup>th</sup> International Software Metrics Symposium*, Ottawa, Canada, IEEE 2002, pp. 249-258.

## BIOGRAPHY

[AUTHOR BIOS LISTED HERE IN ALPHABETICAL ORDER]

**Martin Feather** is a Principal in the Software Quality Assurance group at JPL. He works on developing research



ideas and maturing them into practice, with particular interests in the areas of software validation (analysis, test automation, V&V techniques) and of early phase requirements engineering and risk management. He obtained his BA and MA degrees in mathematics and computer science from Cambridge University, England, and his PhD degree in artificial intelligence from the University of Edinburgh, Scotland. For further details, see <http://eis.jpl.nasa.gov/~mfeather>



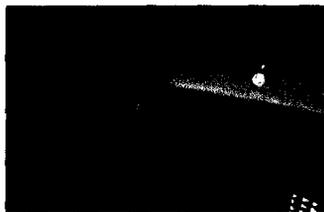
**Lorraine Fesq** is a Principal Engineer in the Software Systems and Operations Engineering Section at JPL. Her passion is to raise the intelligence and increase the capabilities of spacecraft. She is leading MSL's effort to identify and mature promising new V&V technologies that will provide confidence the rover's behavior. She obtained her Bachelor's degree in mathematics from Rutgers University, and her M.S. and Ph.D. degrees in computer science/artificial intelligence from UCLA.

**Michel Ingham** is a Software Engineer and member of the Senior Technical Staff at JPL, where he is a member of the software architecture team for the Mission Data System. His roles include the development of the State Analysis modeling and design methodology, and the application of this methodology to the MSL mission. He received his Sc.D. and S.M. degrees from MIT's Department of Aeronautics and Astronautics, and his B.Eng. in Honors Mechanical Engineering from McGill University, in Montreal, Canada.



**Suzanne Klein** is a senior flight software engineer and test lead at NASA JPL. She is currently working on the Software Quality Improvements Project, an initiative designed to improve software throughout the lab through the introduction of ISO and CMMI capabilities and practices.

Stacy Nelson is a technology infusion consultant for NASA specializing in software verification and validation.



She has successfully applied state of art V&V technology to 2<sup>nd</sup> Generation Reusable Launch Vehicle, Intelligent Flight Control Systems, Autonomous Rotorcraft Project and is currently working on validation of Mars Science Laboratory.

### APPENDIX: SOFTWARE DEFECTS

The software defects table in this appendix contains sample defects for each life cycle phase. It has five columns:

- Life Cycle Phase
- Defect Description
- List of traditional V&V techniques used to find this defect
- Processes used to find this defect
- State of art techniques to find this defect

Life Cycle Phase	Potential Defect	V&V Techniques	Processes	V&V Start of the Art Tool
System Requirements	Inadequate Test Case Coverage	Partition Testing, Coverage Analysis	FTA/FMECA	UBET, Software Synthesis Tool
Software Architectural Design	Processor Resets Not Supported. Flight software not designed to support processor resets during mission-critical events (like entry, descent and landing)	Analytic Modeling, Critical Flow Analysis, Event Tree Analysis, Prototyping	Design Reviews	LTSA, ACME Rapide, SPIN
Software Detailed Design	Performance Issues (memory and timing);	Analytic Modeling, Control Flow Analysis, Prototyping, Simulation, Sizing and Timing Analysis	Design Reviews	LTSA, Software Synthesis Tool, ACME Rapide, SPIN, UBET
Coding	Off-By-One Errors. Any off-by-one iteration errors	Boundary Value Analysis	Code Reviews	Coverity, Insure++, Polyspace, Uno, VectorCAST
Software Integration	No Integrated Regression Testing	Regression Analysis and Testing	CM, CCB, Problem Tracking	FeaVer, Pathfinder, LFF
System Qualification Testing	System Response Time Threshold Untested. Threshold in terms of system response time was not tested	Back-to-Back Testing, Performance Testing, Sizing and Timing Analysis, Stress Testing	Simulators, TestBeds, Rovers	