Software Risk Identification for Interplanetary Probes

Robert J. Dougherty [1], Periklis E. Papadopoulos. Ph.D.[2]

[1] Lockheed Martin Space Systems, 1111Lockheed Way, Sunnyvale, CA 94089,USA, Email:robert.j.dougherty@lmco.com

[2] Professor, Mechanical and Aerospace Engineering, San José State University, One Washington Square, San Jose, CA 95192,USA, Email: ppapado1@email.sjsu.edu

ABSTRACT

The need for a systematic and effective software risk identification methodology is critical for interplanetary probes that are using increasingly complex and critical software. Several probe failures are examined that suggest more attention and resources need to be dedicated to identifying software risks. The direct causes of these failures can often be traced to systemic problems in all phases of the software engineering process. These failures have lead to the development of a practical methodology to identify risks for interplanetary probes. The proposed methodology is based upon the tailoring of the Software Engineering Institute's (SEI) method of taxonomy-based risk identification. The use of this methodology will ensure a more consistent and complete identification of software risks in these probes.

1. INTRODUCTION

Software is a critical component for command, control, and communications in interplanetary probes. Many probe failures and other problems have been traced to software errors. Future probe software will become more complex as it is used to perform novel tasks such as the optimization of landing and exploration sites. This software will become increasing difficult to effectively debug and test and will be prone to new risks.

The first step of the Software Engineering Institute's (SEI) risk management process is risk identification [1]. The purpose of this step is to anticipate risks before they materialize as actual problems. For example, an unclear requirement is a risk, but the incorrect software implementation due to this lack of clarity is a problem. Uncertainties and issues can be transformed into a list of risks that specify the cause of the concern and its potential impact. For a risk to be useful once it is identified, it must be clear, concise and informational within a particular context. This context describes the circumstances, contributing factors and related background information of the identified risk.

Risks that are not identified are not available during the risk management process to analyze and mitigate. The identification of risks is therefore the most important step in this process. Risks identified late in the software development process may require additional, expensive software rework and testing. Risks that are not identified can have catastrophic consequences.

Certain software risks can be identified by analyzing and applying lessons learned from previous missions and by proactively anticipating new risks from evolving technologies. Historically, risks have often been identified in a reactive mode - in response to a failure, change, or problem. These reactive risk identification methods may detect historical problems, but will not adequately detect latent risks from new technologies. The detection of these types of risks in interplanetary probes is essential to a successful mission.

2. UNIQUE ASPECTS OF SPACE PROBE SOFTWARE

Many software risks found in interplanetary probes are common to all software applications. These risks can be avoided through a careful and thorough application of best software engineering practices. However, certain factors make space, and more specifically probe, software different from other software.

Early space software had few commands placed in small memories, and the development process was poorly documented and idiosyncratic. Software size and complexity has greatly increased since then.

Interplanetary probes now contain and are a part of complex systems that have embedded computers that must interact with sensors, ground systems, and other on-board computers. The interactions and coupling of the software systems have critical time-dependent processes. Probe software is also becoming more autonomous and can compensate for mission impacting problems and limitations through self-diagnosis error detection and correction routines.

These new capabilities are increasing the probe's software complexity, making it impossible for a single person to understand all the software components and their interactions. Many of the interactions among the software systems and onboard hardware are essential to mission success. An error in any one of a number of flight parameter values, lines of code, or software interfaces could translate into a total mission loss.

Interplanetary probe software is only used once operationally. Since no two missions are exactly the same, field data from longer-term usage of the software is unavailable to help identify risks. Therefore, complete and accurate models, simulations and testing are important for a successful mission.

The software factors previously noted are common to all space software. The software unique to probes is the entry, descent and landing software that controls the most perilous phases of the mission. This software, for example, may need to make important autonomous realtime decisions to select a landing site that could include altitude ranges, surface slopes, number and size of rocks, temperature, and winds. The software will need to balance scientific objectives with safety objectives to optimize the site location.

3. MISSED OPPORTUNITIES

Risk identification processes are often believed to be adequate until a problem occurs. Accident investigations have shown that the reasons for many software problems are systemic issues that fall outside of the traditional risk identification processes [2]. The root causes of many space accidents are surprisingly simple and seemingly avoidable.

The first U.S. interplanetary mission was Mariner 1in 1962. This was intended to be the first planetary flyby and was destined for Venus. However, the spacecraft was destroyed shortly after liftoff after it had gone off-course. The Post Flight Review Board later attributed the error to an omission of a hyphen in the code, which allowed incorrect guidance signals. It is interesting to note that the Soviet Union attempted four planetary missions prior to Mariner 1, which were all failures [3]. Both of these early space programs had more failures than successes.

The following three missions illustrate missed opportunities to find critical problems prior to launch. Although not discussed here, secondary and less probable causes of accidents also need to be understood and their lessons applied to future missions. These lessons should become part of the feedback loop to the risk management process to inform the space community of new risks. Of course, populating a risk database following an accident is not the preferable method of identifying risks.

3.1 Mars Climate Orbiter (MCO)

The Mars Climate Orbiter (MCO) was lost in 1998 when it entered the thin Martian atmosphere at a lower elevation than expected during an aerobraking sequence. According to the investigation board, the root cause of this accident was the failure to use metric units for the thruster performance data as described in the software interface specification. This resulted in a navigation error that placed the spacecraft at about a 57 km periapsis, instead of the intended 140-150 km. This altitude was too low for the spacecraft to survive [4].

3.2 The Mars Polar Lander (MPL)

This probe was launched in 1999 and failed in the entry, deployment, and landing (EDL) sequence on Mars. The legs of the probe had Hall Effect sensors that were programmed to shut down the descent engines within 50 milliseconds after touchdown on the planet. It was determined that the most probable cause of the accident was that the software interpreted the leg deployments as a touchdown, shutting off the engines. The vehicle then likely dropped about 40 meters and was destroyed. Although the Hall effect is well

understood, the software requirements were not flowed down properly to the software engineers.[5].

3.3 Solar Heliospheric Observatory (SOHO)

SOHO was launched in 1998 to study the solar atmosphere, corona and wind. Although not an interplanetary probe, it is relevant as another example of the risks common to all space software. Two months after completing a successful two-year primary mission in 1998, contact with SOHO was lost. The NASA/ESA Investigation Board concluded that there were no anomalies on-board the vehicle, but instead the problems had been caused by a number of ground errors that led to a significant loss of spacecraft attitude. The loss was caused by operational errors, the failure to adequately monitor the vehicle's status, and a decision to disable part of the on-board autonomous failure detection system. These multiple, avoidable factors created the circumstances that directly caused the errors [6].

.

4 ROOT AND AUXILIARY CAUSES

Many software failures are the result of the systemic factors listed below. Systemic factors look beyond the particular technical causes for an accident and describe or identify the underlying reason or root cause for the problem. Although the risks associated with some of these factors are difficult to quantify, they can still be incorporated into a risk management plan to help ensure that a similar problem is not repeated. Various systemic software risk factors have been listed in the past [2, 7]. These factors have been divided into technical and non-technical categories. The non-technical category includes communication, management and organizational factors. The technical category includes causes related to software engineering.

4.1 Non-Technical Factors

4.1.1 Complacency

One factor that results in increased risk is the complacency due to past successes and the feeling that risk decreases over time because the software has become more "mature." This complacency can cause safety requirements to be relaxed and increased risks to be inadvertently accepted. One example is the risk resulting from a reduction in the monitoring of the software development and testing process. Typically reductions in safety, quality assurance, operations and training can lead to such failures. Warning signs often occur before accidents, but are often unreported or not acted upon. The failed MCO, MPL and SOHO missions were all affected by this culture of complacency.

4.1.2 The Diffusion of Responsibility

A diffusion of responsibility may result in an incomplete coverage of issues by an organization. A program is best served by leadership that demonstrates a broad understanding of all systems and interfaces, and by qualified individuals that take responsibility for each system, subsystem and interface. Reviews and technical discussions should have representatives from all affected disciplines. Software engineers need to interact with other disciplines to understand the source of their software requirements and to identify gaps that may not have been properly communicated to them. However, under principles of good management, only one individual can ultimately be responsible for each and every decision and that individual needs to feel accountable for his or her decisions.

4.1.3 Information Transfer

A clear communication path is needed to transfer important information among project team members. However, a filter is also needed to avoid information overload. Excessive information can confuse or bury more important information. This problem has been exacerbated by email and inexpensive data storage. The transfer of critical information should be handled in a formalized manner to ensure that the message is clear, that the impact and significance is understood by the recipient, and that the issue is properly managed. The use of voice mail and email may provide an illusion that a problem is being actively managed, but the casual nature and volatility of both these communication methods may interfere with an effective communication process.

4.1.4 Employee Turnover

The loss of key technical and management personnel contributes to risk as project continuity is lost and the experience base shrinks. This problem has become worse as new programs replace legacy programs, the aging aerospace workforce moves toward retirement, cost cutting measures reduce the number of employees on a program, and aerospace employees are lost to

higher-paying non-industry companies and projects. It is an enormous challenge in the industry to find key individuals with the right experience and training for the new positions created by these trends.

4.1.5 Legacy Software, Reuse and COTS

The use of Commercial off-the-shelf (COTS), reused, or legacy software is generally believed to reduce risk and the cost of software development because of its previous usage and testing. However, a hidden risk can be that the internal architecture and code in the software is not an exact match for the new probe application. For example, the software may have been developed for a different environment and worked correctly with the original, but not current, set of requirements. The behavior of such software may not be linear or stable in the new environment. COTS or third party software is especially a concern because the source code may be proprietary. The developer may not have access or understand the internals of the software except from limited vendor information. Software engineers must understand the assumptions and limitations of any COTS or legacy code before adoption. An independent verification and validation (IV&V) program is especially important to reduce risk for this type of software.

4.1.6 Requirements Creep and Unneeded Code

Requirements creep occurs when features are added to the software that are not true requirements. Instead of converting these features to actual requirements, or removing them, the inadequately documented features are left in the software. Often requirements from a previous mission are retained under the assumption that removing the unnecessary software code would create a higher risk than leaving the code unchanged. However, this unneeded code increases complexity during development and increases the possibility of untested states.

4.2 Technical Factors

4.2.1 Inadequate Engineering

Software activities are sometimes inadequate due to a lack of resources or critical processes that are flawed. One such problematic process is the incorrect flow-down of system requirements. A variety of other inadequacies and engineering problems include: interfaces that are defined after the software coding has already begun, programmers who do not have the systems experience or science background to understand the problem they are solving, inconsistent or incomplete error handling, and testing that does not cover all of the possible exceptions. The example of the MPL failure discussed earlier illustrates inadequate engineering; i.e. there was no reasonableness check for the altitude when the descent engines were turned off.

4.2.2 Inadequate Reviews

Every review needs to have a clear goal and outcome. The responsibilities of all review participants should be clearly understood. Quality Assurance (QA), for example, should not approve documents or processes by merely checking signatures without understanding the real quality of the documents. QA also needs to ensure that peer reviews take place and that a "second set of eyes" reviews processes, coding, and parameters at appropriate points in the development process. It is often the case that a second review by a qualified person will be sufficient to greatly increase quality. However, by itself, QA cannot ensure that all critical software values, analyses and processes are correct. An independent verification and validation (IV&V) organization is necessary to double-check all software engineering work performed. A contributory factor to the failure of the SOHO mission was that important reviews were bypassed because of tight schedules and compressed timelines.

4.2.3 Inadequate Specifications

Many software-related problems are caused by flawed requirements and their flow-down, rather than by actual coding problems. Requirements are sometimes incomplete or unclear, assumptions may be incorrect or unstated, and system-states may not be understood and controlled. Good specifications that follow an effective process and have traceable requirements are important to minimize risks. The MPL accident report suggested that the software engineers on the project would have prevented the accident if they had understood the rationale behind the design.

4.2.4 Inadequate Education

Software engineers are rarely taught safe design principles, such as eliminating unused functionality, designing appropriate error detection and correction, and conducting reasonability checks. In a complex system such as interplanetary space probes, small errors can have catastrophic consequences. Education

should be on-going so that software engineers continue to remain current on important advances in their discipline. A typical undergraduate education does not typically provide the level of skills needed for architecting software for complex space systems.

4.2.5 Inadequate Testing

Although the space industry attempts to "test like you fly, fly like you test," in reality, simulations and test environments have limitations and may not be able to meet that goal. The understanding of these limitations and testing assumptions are important to reduce risks. Testing is only as good as the test plan, test equipment, and the test cases chosen. In addition, testing may not catch every problem, especially when COTS or legacy software is being used. The internals of the software need to be well understood so that assumptions are known and appropriate test cases can be chosen. Software is too complex for test cases to be comprehensive, and instead need to include representative non-nominal and stressed conditions. Inadequate testing could be considered a contributing factor in almost every probe accident.

4.2.6 Inability to Understand Software Risks

Some engineers do not realize the complexity of the software and believe testing will reveal any latent problems. The software may be assumed to be correct unless testing shows otherwise. Practical testing, however, is limited to a subset of possible states. This inability to understand software risks also allows the inclusion of unnecessary software functionality. The over-reliance on testing to uncover problems and unused functionality increases risk. The removal or modification of software fault protection for reasons like performance also increases risk and have led to mission failures.

4.2.7 Use of Hardware Techniques for Software

Software is pure design, and its failure modes are different then physical devices. Software doesn't have safety margins like hardware - a single bit flip may cause a mission loss in a poorly designed system. Most software problems are flaws in the requirements specification and not in coding errors. Failure modes are often not clearly defined for software. A statement like, "Flight software doesn't execute properly" is too vague to be useful in identifying risks. Redundancy in software is no help in mitigating these risks if the problem is in the hardware. In fact, such software redundancy may give a false sense of security if the solution is merely duplicating a design error, and the additional complexity may actually increase the risk.

5.0 ANTICIPATING FUTURE RISKS

Lessons learned and accident reports are useful in understanding and avoiding past errors, but latent software risks are always a concern. Increased software complexity and new technologies increase these risks. A compilation of potential risks of new software technologies used in a probe can be used to address and mitigate these risks. New modeling and analysis tools also will be needed as these technologies are introduced in the future.

6.0 THE MODIFIED TAXONOMY-BASED RISK IDENTIFICATION METHODOLOGY

There are several commercial and government software tools that are advertised to help identify risks - none of them do so effectively. They instead allow risks already identified to be classified according to areas like severity and probability as part of an overall risk management process.

6.1 Risk Identification Methods

There are many ways to identify risks - all of them can be useful, but each is limited in the number and types of risks that they can effectively identify. Here are six approaches to identify risks:

1. **Identification Based on Experience** – The engineers, scientists and others who have the most experience and understand the details of the probe are the best suited to identify its risks. A process can be developed to allow these experts to report risks.
2. **Identification Based on Historical Data** – After each mission, the data is analyzed by the various subsystems and risks are identified either directly or by analogy. Applicable risks are identified and tracked for each mission. NASA, for example, maintains a Lessons Learned/Best Practices database that may be useful in identifying risks. Data from as many sources as possible should be compiled for a comprehensive list of possible risks.
3. **Brainstorming** – All stakeholders may add value to the risk identification process. The

synergy of a diverse group may help expose a new risk.

4. **Voluntary Risk Reporting** – A method of reporting risks, anonymously if desired, should be in place. Potential risks that are submitted are analyzed for applicable missions. This provides an avenue for risks to be identified that may be politically unpopular or without another outlet to be raised.
5. **Reviews** – Requirements, code and other reviews and working groups are common methods of identifying risks.
6. **Testing** – Testing can expose flaws in both processes and assumptions and is a primary method of risk identification, although it is preferable for risks to be exposed earlier in the software development cycle.

6.2 The SEI Taxonomy-Based Risk Identification Method

A systematic method to ask the right risk-related questions is needed that can be tailored for a particular interplanetary probe. A process needs to be in place that goes beyond assigning blame, and instead concentrates on helping understand the sources of risks. One method is based on the Software Engineering Institute's (SEI) Taxonomy-Based Risk Identification method (SEI/CMU Technical Report CMU/SEI-93-TR-6 ESC-TR-93-183).

The SEI Taxonomy-Based Risk Identification method is a process to identify risks associated with the development of a software-dependent project in a systematic and repeatable way. It has been tested in both government and civilian projects and shown to be useful, usable, and efficient. In general, the method consists of a series of questions that helps surface risks by using an interviewing process with subsets of the project team.

Some engineers and scientists may express resistance to this method because many of the questions are open-ended and some risks cannot be easily quantified. Technical people prefer to have a method that can assign a numerical value to the severity and probability of risks, but the quantification of some of the identified risks may be misleading. For example, bringing a new technical lead into a project certainly introduces risk, but it is difficult to place a numerical value on a person's education and experience as compared to the previous technical lead - to do so would be meaningless. However systems have been proposed that quantify every risk in such a way. A list of questions similar to the SEI risk identification taxonomy process, for example, has been proposed by Karolak [8]. He quantifies each risk identification question with a number between 0 (none) and 1 (all).

The Taxonomy-Based Risk Identification method consists of 194 questions divided into three main divisions (Product Engineering, Program Engineering, and Program Constraints) and 13 further sub-divisions. The sub-divisions are listed below with a sample question from each one to illustrate the type and level of sample questions. The numbers in brackets before the sample questions are the numbers assigned by SEI.

A. Product Engineering

1. Requirements - [6] Are the external interfaces completely defined?

2. Design - [16] How do you determine the feasibility of algorithms and designs?

3. Code and Unit Test - [42] Is the development computer the same as the target computer?

4. Integration and Test - [51] Are the external interfaces defined, documented, and baselined?

5. Engineering Specialties - [66] Are safety requirements allocated to the software?

B. Development Environment

1. Development Process - [85] Is there a requirements traceability mechanism that tracks requirements from the source specification through test cases?

2. Development System - [94] Does the development system support all aspects of the program?

3. Management Process - [106] Are there contingency plans for known risks?

4. Management Methods - [126] Does program management involve appropriate program members in meetings with the customer? (including Technical Leaders, Developers, Analysts)

5. Work Environment - [139.b] Are members of the program able to raise risks without having a solution in hand?

C. Program Constraints

1. Resources - [147] Are there any areas in which the

required technical skills are lacking?

2. Contract - [165] Are there dependencies on external products or services that may affect the product?

3. Program Interfaces - [175] Are the external interfaces changing without adequate notification, coordination, or formal change procedures?

6.3 Questionnaire Creation Process

Specific software risks are often known by project personnel but are not always communicated. Effective risk identification requires a process and culture of open communication to encourage all stakeholders to use their knowledge of the project to help with its success. The basic risk identification methodology for a particular probe should be tailored from the generic SEI Risk Taxonomy Method questionnaire to include risks peculiar to that probe.

The tailoring of the generic SEI software questionnaire should follow a systematic process to create a probe-specific questionnaire:

1. Review each generic risk taxonomy question and tailor it to the particular space probe domain.
2. Review the literature for other potential risks from new technologies used in the probe. Add or tailor the generic risk taxonomy questions for these additional risks.
3. Review the literature on lessons learned from other similar missions. Add or tailor the generic risk taxonomy questions for these additional risks. Note that these lessons learned do not need to be from similar probes or even from aerospace risks, but may include any software lesson that can be applied to the particular probe.
4. Have the Probe's Software Team and other experts review the list for further modifications.

These steps will result in a baseline risk questionnaire, which can be modified on an on-going basis as risks are identified from reviews and other sources. This dynamic baseline can also be modified and used as a starting point for other similar missions.

7.0 PROCESS FOR APPLYING THE METHODOLOGY

The reviewers must prepare before the interviews are conducted. If possible, they should understand the processes that are being used by the organization and initially assess whether the processes are complete, accurate, and being followed. This will help facilitate the interview process.

The following steps describe a process to apply the proposed methodology after the modified baseline questions have been completed:

1. Set up Interview Groups

Interview groups should be determined and sorted by functional area or subsystem. For example, the interview groups might be divided into command and control software, mission data software, sequencing software, guidance and navigation software, flight parameters, scientific experiment, testing, and IV&V. There is no single approach or perfect grouping, but it may prove more efficient to keep personnel together that work on similar tasks.

Ideally, there would be no reporting relationships in a particular interview group. Management commitment is needed for the interview process, but management should not be present so the interviewees feel they can speak freely. If possible, groups would be limited to 5 participants [1], although this may be difficult to ensure in practice.

2. Divide the Questions

Certain questions are more appropriate for some groups than others. The questions for each interview group should be a subset of the total risk identification questions. A list of questions tailored to each group should be provided to the group members before the interviews.

3. Conduct the Interviews

The risk identification session begins with briefing the participants with the methodology and its purpose. During the interview, the tailored questions can lead to

other issues, concerns and risks. Items of risk noted during the interviews are used to update the risk database for the mission and the questionnaire template for the mission. For practical reasons, the interviews should not normally last more than 2 hours.

4. Provide Feedback

A summary of the potential issues is provided back to the participants at the end of the interview, and a hard copy is subsequently provided to the project manager .

5. Repeat Process

Depending on the size and complexity of the project, this process may need to be repeated several times to different groups during the software development life cycle. If the methodology is only performed once near the beginning of the project, many risks will still be unknown, but if it is only performed near the end of the project, then the risks may be expensive and difficult to correct.

8.0 TESTING THE METHODOLOGY

The best test of the usefulness of a method is whether it actually works. The lessons learned from the failed missions in Section 3 were used to tailor some of the baseline questions to test the methodology. Obviously these examples are somewhat contrived since the problems have already occurred, but they are still useful to illustrate how the process works and ensures that these type of systemic problems are addressed and are less likely to re-occur on future missions.

Risk 1: Complacency because of past mission successes

Question: Do you feel that your past successes have reduced your risk of failure?

Risk 2: The constants definition process not well defined

Question: Are there formal, controlled plans for all development activities? Are developers familiar with the plans?

Risk 3: There was a lack of anyone in charge of the entire process

Question: Is there a qualified person responsible for every process and the overall processes?

Risk 4: No communication channel for relaying the problem when discovered

Question: Are the realtime interfaces among the different organizations sufficient for status and anomaly resolution?

Risk 5: No formal anomaly reporting and tracking system

Is there a formal anomaly reporting and tracking system for all phases of the development and deployment process?

9.0 CONCLUSION

Early software risk identification is important for the success of interplanetary probes. Potential risks must be considered not only from other probe missions, but also from other sources within the aerospace industry as well as from other industries. There are currently no commercial or government products that are completely adequate for probe software risk identification. Most available risk management applications only provide a shell for previously identified risks.

The SEI risk identification method can be used as a starting point to create a tailored true risk identification methodology for interplanetary space probes. The general software risk questions are tailored through a series of steps that ultimately provides a mission with a unique questionnaire for risk identification. A process to apply the methodology transforms it from a theoretical suggestion to a practical process for identifying risks for a particular probe. The software lessons learned from other missions are incorporated into the questions to ensure the past problems are not repeated.

This proposed methodology could replace ad-hoc, undocumented, incomplete or reactive methods that are typically used. This methodology will enable a more consistent identification of risks in interplanetary probe software systems with an ultimate goal of more successful missions.

10. REFERENCES

[1] Carr, Marvin J., et al. *Taxonomy-Based Risk Identification*, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993.

[2] Leveson, Nancy G., *Systemic Factors in Software-Related Spacecraft Accidents*, American Institute of Aeronautics and Astronautics, 2001.

[3] Williams, David, *Planetary Sciences at the National Space Science Data Center,* 19 May 2004, NASA Goddard Space Flight Center, 17 August 2004, <http://nssdc.gsfc.nasa.gov/planetary/chronology.html >.

[4] Stephenson, Arthur G., *Mars Climate Orbiter: Mishap Investigation Board Phase I Report,* NASA, November 10, 1999.

[5] JPL Special Review Report, *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions,* Jet Propulsion Laboratory, March 22, 2000.

[6] Joint NASA/ESA Investigation Board, *SOHO Mission Interruption,* NASA/ESA, 31 August 1998.

[7] Leveson, Nancy G., *The Role of Software in Spacecraft Accidents*, American Institute of Aeronautics and Astronautics, 2001.

[8] Karolak, Dale W, *Software Engineering Risk Management,* IEEE Computer Society Press, Los Alamitos, California, 1996.
