Operational Anomalies as a Cause of Safety-Critical Requirements Evolution

Robyn R. Lutz *
Jet Propulsion Laboratory
and Iowa State University
rlutz@cs.iastate.edu

Ines Carmen Mikulski
Jet Propulsion Laboratory
Pasadena, CA 91109-8099
ines.c.mikulski@jpl.nasa.gov

Abstract

This paper reports the results of a small study of requirements changes to the onboard software of seven spacecraft subsequent to launch. Only those requirement changes that resulted from operational (i.e., post-launch) anomalies were of interest here, since the goal was to better understand the relationship between critical anomalies during operations and how safety-critical requirements evolve. The results of the study were surprising in that anomaly-driven requirements changes during operations were rarely due to previous requirements having been incorrect. Instead, changes involved new requirements either (1) for the software to handle rare but high-consequence events or (2) for the software itself to compensate for hardware failures or limitations. The prevalence of new requirements as a result of post-launch anomalies suggests a need for increased requirements-engineering support of maintenance activities in these systems. The results also confirm both the difficulty and the benefits of pursuing requirements completeness, especially in terms of fault tolerance, during development of critical systems.

Keywords: operational anomaly, requirements evolution, software safety, spacecraft, maintenance

1. Introduction

This paper reports the results of a study of safety-critical requirements changes, in response to anomalies during flight, to the software onboard seven spacecraft. We distinguish these

*To appear, *The Journal of Systems and Software*. An early version of this paper was presented at the Fifth IEEE Symposium on Requirements Engineering, Toronto, Canada, Aug. 27-31, 2001.

anomaly-driven requirements changes from requirement changes resulting from planned evolution or maintenance in an effort to understand and, perhaps, reduce their number and attendant risks.

In planned evolution or maintenance there are many requirement changes to the onboard software on a spacecraft after launch. The lifetime of a spacecraft is usually measured in years, and scheduled updates must maintain the software as the spacecraft proceeds through the phases of its mission. For example, new software tailored to the next phase will often be uplinked to a spacecraft's computers prior to each navigational maneuver, orbital insertion around a planet, sequence of scientific data-gathering, etc.

In this study, however, it was not these anticipated requirements changes due to scheduled maintenance that were of interest. Instead, the goal was to better understand the relationship between anomalies during operations and the evolution of safety-critical requirements. We thus focus on a very small but essential and high-risk (because urgent and unplanned) subset of the total set of requirements changes to the spacecraft software. Software requirements such as these that are essential to the accomplishment of the spacecraft's mission are defined as safety-critical in this domain. The objects of study were thus the unanticipated requirements changes prompted by critical, operational anomalies.

The rest of the paper is organized as follows. Section 2 describes the approach. Section 3 presents and discusses the results. Section 4 places these results in the context of related work in both requirements engineering and maintenance. Section 5 provides a summary and some concluding remarks.

2. Approach

The data for the analysis of critical, unanticipated requirements changes were drawn from an institutional database of anomaly reports. Data were analyzed from seven spacecraft: the Galileo mission to Jupiter, launched October, 1989; Mars Global Surveyor, a mapping mission launched in November, 1996; Cassini/Huygens, launched in October, 1997, to explore Saturn and Titan; Deep Space 1, a technology demonstration mission (of ion propulsion and remote agent technologies, among others) launched in October, 1998; Mars Climate Orbiter, launched December, 1998, and lost at Mars; Mars Polar Lander, launched January, 1999 to study the Martian surface and dig for water ice, lost during landing; and Stardust, a spacecraft that will return cometary material to Earth, launched in February, 1999.

The reporting mechanism for the operational anomaly data is an on-line form called an Incident/Surprise/Anomaly report (ISA). An ISA consists of three parts. The first part is filled in at the time of the occurrence by the operator. The second part is filled in by the analyst assigned to investigate the occurrence. The third part is filled in later with a description of the corrective action that was taken to close out the incident. Additional information regarding criticality, priority, time and date, subsystem, etc., can also be entered into the available fields.

It is worth noting that an ISA is not a defect report. An ISA is written whenever the behavior of the system differs from the expected (i.e., required) behavior in the eyes of the operator. Thus, the ISA provides valuable information to the requirements engineer because it tends to capture gaps between the requirements as specified and implemented and the, perhaps different, user's expectations.

The ISA also provides a means of documenting near-misses, i.e., failures that almost occurred but were prevented by some fortuitous circumstance (e.g., fault monitoring, contingency commands, a change of mode, etc.). In some cases the near-miss prompts a change to the flight software requirements. For example, in this study six ISAs described incidents in which an in-flight anomaly triggered a contingency (safe) mode or fault-protection response. In these cases a new software requirement resulted from analysis of the incident in order to preclude such an anomaly in the future.

3. Results and analysis

The data set analyzed consisted of 189 ISAs in the highest criticality level from the seven spacecraft listed above. The criticality level had been assigned by each project based on standard classifications (JPL, 1997). Since there were slight differences in the processes of the seven projects regarding which fields of the anomaly reports were used, we studied the anomaly reports that met one of the following three criteria in order to assure that we provided coverage of all critical ISAs: Red flag or Potential red flag = On (indicates high mission risk if the event were to recur; significant or catastrophic risk; and uncertain fix); Criticality = 1 (the highest category, indicating an unacceptable risk with no workaround); or Criticality = 2 and Priority = 1 and Failure Effect >= 2 (the high priority is assigned by the correcting agency indicating a "must-fix" situation; the failure effect of the anomaly is significant or catastrophic) Anomalies meeting one or more of these criteria were studied and are together included under the shorthand term "critical ISAs" in this paper.

The results reported here are part of a larger project to use analyses based on Orthogonal Defect Classification (ODC) (Chillarege et al., 1992) to characterize post-launch safety-critical software anomalies. The ODC-based approach has allowed detection of a surprising number of high-criticality anomalies resolved by changes to flight software requirements (the "target" in ODC terms) during operations.

Table 1 summarizes the results. 44 of the 189 critical ISAs had flight software as their target, i.e., the anomaly prompted a change to the flight software. (The other 145 ISAs produced changes to procedures, ground software, documentation, etc., outside the scope of this paper.) 15 of the 44 ISAs resulted in updates only to the code but not to design or requirements (e.g., bias or filter updates, adjustment of a timeout parameter, erroneous re-initialization to "on" rather than "off"). Nine of the remaining anomalies that had flight software as their target had fixes to design logic as the corrective action, but had no effect on requirements. Of the remaining ISAs, two were maintenance problems (incorrect software patches); one recorded an occasion on which an existing contingency software command, previously created just in case an overpressure emergency should ever occur, needed to be sent to the spacecraft to close a leaking valve; one fix was not implemented due to cost and schedule tradeoffs; and five anomalies were not classified due to currently incomplete

Change	*Number*
New requirement	**11**
Design logic fix	9
Code fix	15
Maintenance (previous patch fixed)	2
Contingency command	1
Fix not implemented	1
Unclassified	5
Total: Changes to Flight Software	44

Table 1: Summary of flight software changes due to safety-critical operational anomalies

information.

The discussion that follows focuses on the remaining eleven of the 44 high-criticality, flight software ISAs, since each of these involved new software requirements for the flight software.

3.1 New requirements for rare events

Post-launch critical anomalies were resolved by new requirements to handle rare or anomalous events in seven cases. In the first of these, an unusual code path (due to an unanticipated combination of circumstances) caused unexpected behavior. In another, an unforeseen scenario led to the use of obsolete data in a particular case. In another case, an inappropriate software request for data just as it became unavailable resulted in loss of timing and mode synchronization among software components. In three other critical anomalies, a rare scenario led to an overflow. For example, in one of these cases, a contingency situation (failure of both redundant units) caused an overflow of the message queue and a warm boot.

In another anomaly, safety-critical post-launch requirements changes were initiated due to a rare environmental event–namely the unexpected outflow of some debris that interfered with the spacecraft's ability to determine its position in space. The new software requirements were to make the spacecraft more fault-tolerant to that type of temporary "loss of vision" in the future. In each of these seven cases, the anomaly was considered to contribute risk to the mission, and a critical software change was made to add robustness against future occurrences.

These results confirm the importance of rare events in critical failures. As Hecht noted in his 1993 paper, "the inability to handle multiple rare conditions, such as response to hardware failures or exception conditions caused by the computer state, is a prominent cause of program failure in well-tested systems" (Hecht, 1993). Hecht further noted, "Rare events were clearly the leading cause of failures among the most severe failure categories." The results described here suggest a close, causal relationship betwen critical anomalies post-launch that are due to rare events and the evolution of the flight-software requirements to

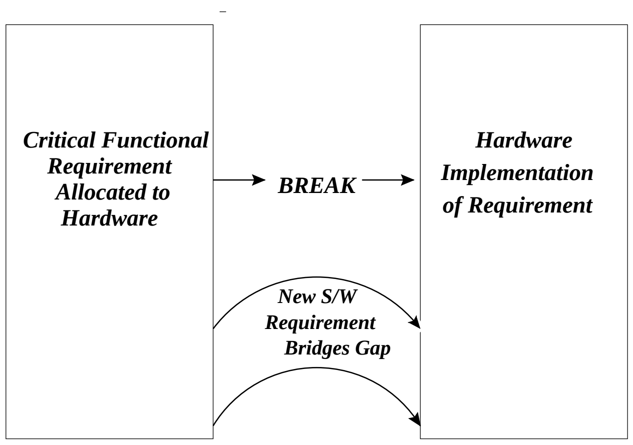
Critical Functional Requirement Allocated to Hardware
BREAK
Hardware Implementation of Requirement
New S/W Requirement Bridges Gap

Figure 1: Software requirements change to compensate for hardware failure

protect the systems against such occurrences in the future.

3.2 New requirements to compensate for hardware

Critical requirements changes were driven by changes to the hardware in four cases. In one case, for example, a hardware failure prompted on-board fault-protection software to turn off the hardware component. Subsequent analysis revealed the "what-if" scenario that the other two, redundant components might fail in worse condition (unlikely, but credible). In that case, the on-board software would need to turn on the "least-failed" component that had been previously turned off. A new software requirement to facilitate this switching was established in response to the failure scenario arising from the initial hardware failure.

In another case, a new capability was added to the flight software in response to a damaged solar array panel that could not deploy as planned. In a third case, a new software requirement resulted from the discovery of unexpected angular rates around an axis whenever a thruster was fired. Similarly, when proximity to an antenna caused noise in some transducers, resulting in inaccurate readings and unnecessary re-setting of components, the anomaly was resolved by a new flight software requirement to compensate for the noise.

One issue of interest in these cases is that the trigger for software change was hardware failure. This is contrary to the underlying assumption of some defect models that what breaks is what gets fixed. It is very typical, however, of complex, heavily embedded software on the spacecraft, in which, as hardware degrades, the software requirements evolve to close the gap (Fig. 1).

Perhaps the best-known example of this is the re-programming of one of the Galileo spacecraft's computers with clever, new compression algorithms to minimize scientific data loss when Galileo's large antenna failed to deploy.

More recently, when the spacecraft Deep Space 1 lost a critical sensor, the software on board was changed to compensate for the hardware failure. The failure of the Deep Space 1 star tracker in November, 1999, jeopardized the planned encounter of the spacecraft with a comet. The star tracker determines the spacecraft's orientation in space and, without it, the spacecraft is in some sense blind. In order to compensate for the hardware failure, software was radioed to re-program the on-board camera to serve as a replacement for the star tracker. The project manager called the updated software "very complex and innovative" and labeled the change a "rescue"(JPL News Release, 2000). Although none of the requirements changes in this study approached the scope of the Galileo or Deep Space software changes, the possibility of having to rebuild remotely a significant amount of the software emphasizes the need for requirements engineering support during the post-launch maintenance phase.

3.3 Consequences for the requirements process

The profile of critical anomalies found during *operations* on these three spacecraft was compared with earlier work by one of the authors on critical anomalies during *integration and system testing* of flight software. The previous work was on two different, but fairly similar spacecraft (Voyager and Galileo), roughly comparable in function and complexity to the spacecraft in this study. It was found in the earlier study that, during the testing phase, most of the critical anomalies involved requirements or interfaces (Lutz, 1993).

The small number of critical requirements-related anomalies found post-launch in the current study, and the fact that all the requirements-related anomalies yielded new requirements (rather than corrected requirements) suggest that the testing process is doing a good job of removing requirements-related defects. The extensive integration and system testing of troublesome components may also provide some explanation for a recent finding by Fenton and Ohlsson of what they call "strong evidence of a counter-intuitive relationship", i.e., that modules that are the most fault-prone pre-release are the least fault-prone post-release (2000). It may be that modules identified as fault-prone during spacecraft system testing–especially if the fault affects requirements–are (appropriately) subjected to more thorough testing.

An interesting question regarding the 145 critical ISAs that did not produce changes to flight software is whether a mechanism similar to the use of flight software to compensate for hardware problems occurs, whereby changes to ground (as opposed to flight) components are compensating for problems in flight software. That is, how many of the ISAs involved problems with the flight software that were remedied by changing the more readily modified components of the system such as ground software or procedures?

Investigation revealed that, in fact, only six of the ISAS met this criteria, and that only one of the ISAs involved a change to ground software requirements. Of the six ISAs that involved flight software problems but not flight software fixes, four of the six resulted in changes to prevent the recurrence of the problem. Of these, one involved modification to the ground software (to add a pause), one resulted in an update to documentation (regarding an unanticipated side effect of a software command), and two led to changes in operational procedures (to preclude recurrences of the scenarios).

The other two of the six ISAs described modifications to recover from future recurrences of the problem. These included updating the procedure to recover from radiation-induced bit errors and adding a procedure to automatically recover pending commands lost if the software crashed.

None of these six ISAs involved flight software requirements, in the sense that none of these scenarios would, if identified during requirements analysis, have changed the flight software requirements. Thus, it appears that changes to ground software, procedures, and documentation are not masking changes to flight software requirements.

As far as the long-term goal of the research in which this study is embedded, i.e., to further reduce the number of safety-critical anomalies post-launch, the results are somewhat negative. It is difficult to see how the requirements engineering process during development can be readily adjusted so as to preclude the post-launch requirements changes.

To the extent that improvement is possible, these results emphasize the benefit of thorough hazard analysis and fault-scenario explorations, and of extensive contingency planning during requirements analysis. Even when a possible requirement has not been implemented, documented contingency studies can facilitate accurate requirements evolution when it becomes necessary during operations. The fact that seven of the eleven critical post-launch requirements changes were in response to rare events indicates that the cost/benefit tradeoff of such hazard analyses makes them practical for such critical systems.

In summary,

- Bad things did happen due to incomplete requirements, i.e., incomplete requirements were not "good enough" for these critical systems. The benefit of working toward complete requirements was clear.

- The missing requirements were "hard", i.e., they involved subtle, rare, or unexpected circumstances or scenarios. The difficulty and cost of achieving the level of requirements understanding needed to forestall such anomalies were high.

- What broke was not always what got fixed, i.e., new software requirements compensated for hardware failures or evolving limitations.

4. Related work

Most work in requirements evolution focuses on the pre-implementation phases of a system. For example, Anton and Potts describe the use of goals and obstacle analysis to refine evolving requirements (1998). Zowghi, Ghose, and Pappas provide a logical framework for reasoning about requirements evolution, also within the requirements analysis phase of development (1997). An open issue worth exploring is to what extent these techniques are also useful for analyzing the consequences of requirements evolution during operations.

Requirements evolution post-deployment has been studied primarily from the viewpoint of how it can be managed. DeLemos provides a model of an operational system in which requirements evolution (in this case, automating the self-destruct feature of a rocket) can be structured so that the components remain unchanged while their interactions adapt to

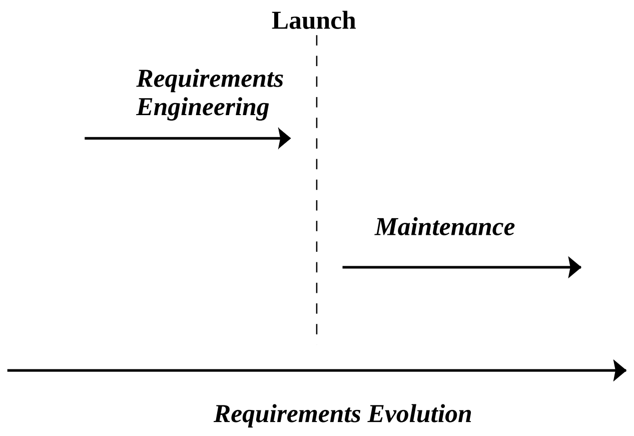

Figure 2: Continuous evolution of requirements vs. discontinuity in methodologies

the changed requirements (2000). In our study, the requirements changes were low-level and functional rather than architectural, so primarily involved the components themselves.

Lam and Loomes, with experience in product line evolution, discuss management of requirements evolution after installation with particular attention to the impact on stakeholder viewpoints (1998). The requirements changes that they describe are much more open to negotiation than are the safety-critical requirements changes we saw in this study. However, their emphasis on modeling evolution as a series of distinct changes, and on developing a "richer notion of traceability" fit well with the analytical process involved in making anomaly-driven requirements changes on the spacecraft.

Leveson (1995) emphasized the difficulty of producing and maintaining high-reliability operational software. She noted that even though the Space Shuttle had one of the most sophisticated software development processes in existence, with extensive resources devoted to maintenance and verification, software errors at the highest level of severity had been discovered in its released software.

Fickas and Feather provide a possible direction for actually reducing the unpredictability of some of the anomaly-induced requirements changes (1995). They describe requirements monitoring for dynamic environments. It may be necessary in such domains for the system to evolve, e.g., as assumptions underlying the requirements change. An open question is to what extent it might be possible, via monitoring, to anticipate some of the rare events or hardware failures that triggered the critical requirements changes on the spacecraft.

As distinct from requirements-engineering approaches, maintenance methodologies tend to focus on classifying and managing requirements changes, rather than on analyzing or anticipating the changes. Figure 2 summarizes the gap that appears to exist between requirements-engineering-based analysis of requirements evolution and maintenance-based studies of requirements evolution.

Harker, Eason, and Dobson (1992) classify evolving requirements as Mutable (in response to the environment), Emergent (in response to a fuller understanding of possible scenarios and their consequences), Consequential (post-delivery pressures for enhancements), Adaptive (allowing local customization), and Migration requirements (supporting gradual movement to the new system). At least in the spacecraft domain, the categories can sometimes overlap. Some anomaly-induced requirements changes can accurately be described as both Mutable (in response to changes in the environment or hardware) and Emergent (in response to a better understanding of the possible failure scenarios).

Bennett and Rajlich (2000) note in their recent roadmap paper that software evolution lacks a standard definition. They use the term "Maintenance" to refer to general post-delivery activities, and divide the Maintenance phase into five sequential stages: Initial development, Evolution, Servicing, Phase out, and Close down. The goal of the software evolution stage is "to adapt the application to the ever-changing user requirement and operating environment. The evolution stage also corrects the faults in the application and responds to both developer and user learning, where more accurate requirements are based on the past experience with the application."

The spacecraft post-launch requirement changes also here correspond to several phases. Clearly, the on-board software fits the software phase called "Evolution." It also, to some extent, fits the subsequent phase of software maturity, called "Servicing." The Servicing phase is characterized by the danger of loss of key personnel and information (typical of lengthy spacecraft missions), as well as by a planned commitment (also typical of spacecraft missions) to keep requirement changes small in scope.

Part of the difficulty in using the maintenance literature to understand the critical spacecraft requirements changes is that the domain of concern in the maintenance literature is often the business environment (e.g., handling the clamor of competing users) rather than safety or mission-critical physical environments. One exception is the recent work by Tai et al. to reduce the risk of maintenance in critical systems and support the evolvability of spaceborne computing systems post-launch (2000).

5. Conclusion

The results suggest that, for critical systems, effort spent on requirements analysis, especially of failure scenarios, rare events, and contingency planning for how software can compensate for hardware failures, is merited. Incomplete requirements did, in fact, cause anomalies to occur. The bad news was that these missing requirements were hard–that is, they involved subtle, rare, or unexpected circumstances or combinations of events. One of the lessons learned from the study of requirements changes during operations was that new software requirements were often needed to make the deployed software more robust against unanticipated scenarios. To a limited extent, requirements evolution in response to these causes may be able to be anticipated, and we have indicated some promising directions in current research toward this goal.

Another lesson learned was that requirements evolution post-launch was driven in part by a dependence on software to compensate for evolving hardware limitations. Contrary to common defect analysis assumptions, in these cases what broke (the hardware) was not what got fixed (the software). This mechanism appears to be more common than expected in systems where the hardware is difficult to replace. For example, implantable medical devices also use safety-critical software evolution to compensate for hardware anomalies. We saw, as well, that existing maintenance models do not incorporate the requirements-engineering techniques that might help in analyzing and anticipating possible requirements evolution.

Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, under a contract with the National Aeronautics and Space Administration. Partial funding was provided under NASA's Code Q Software Program Center Initiative UPN 323-08. The authors thank reviewers of an early version of the paper for the question posed in 3.3. The first author thanks Dr. Daniel Paulish and Siemens Corporate Research, Princeton, NJ, for their hospitality during the writing of part of this paper.

References

[1] Anton, A. I. and Potts, C., 1998. The use of goals to surface requirements for evolving systems. In: Proceedings of the 20th International Conference on Software Engineering. IEEE Computer Society, Los Alamitos, CA, pp. 157–166.

[2] Bennett, K. H. and Rajlich, V. T., 2000. Software maintenance and evolution: a roadmap. In: Finkelstein, A. (Ed.), The Future of Software Engineering, ACM Press, New York.

[3] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M., 1992. Orthogonal defect classification: a concept for in-process measurements. IEEE Transactions on Software Engineering 18 (11), 943–956.

[4] deLemos, R., 2000. Safety analysis of an evolving software architecture. In: Proceedings of the Fifth IEEE International Symposium on High Assurance Systems Engineering. IEEE Computer Society, Los Alamitos, CA, pp. 159–167.

[5] Fenton, N. E., and Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering 26 (8), 797–814.

[6] Fickas, S. and Feather, M., 1995. Requirements monitoring in dynamic environments. In: Proceedings of the Second International Symposium on Requirements Engineering. IEEE Computer Society, Los Alamitos, CA, pp. 140–147.

[7] Harker, S. D. P., Eason, K. D., and Dobson, J. E., 1992. The change and evolution of requirements as a challenge to the practice of software engineering. In: Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE Computer Society, Los Alamitos, CA, pp. 266–272.

[8] Hecht, H., 1993. Rare conditions–an important cause of failures. In: Proceedings of the Eighth Annual Conference on Computer Assurance. IEEE Computer Society, Los Alamitos, CA, pp. 81–85.

[9] Jet Propulsion Laboratory, 1997. ICAP Anomaly Process, Glossary. Safety and Mission Assurance Information Systems, Jet Propulsion Laboratory, Pasadena, CA.

[10] JPL News Release, July 27, 2000. Space rescue makes close encounter possible. Jet Propulsion Laboratory, Pasadena, CA.

[11] Lam, W. and Loomes, M., 1998. Requirements evolution in the midst of environmental change: a managed report. In: Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering. IEEE Computer Society, Los Alamitos, CA, pp. 121-127.

[12] Leveson, N., 1995. Safeware: System Safety and Computers. Addison-Wesley, Reading, MA.

[13] Lutz, R., 1993. Analyzing software requirements errors in safety-critical, embedded systems. In: Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp. 126–133.

[14] Tai, A. T., Tso, K. S., Alkalai, L., Chau, S. N., and Sanders, W. H., 2000. On low-cost error containment and recovery methods for guarded software upgrading. In: Proceedings of the 20th International Conference on Distributed Computing Systems. IEEE Computer Society, Los Alamitos, CA.

[15] Zowghi, D., Ghose, A. K., and Peppas, P., 1997. A framework for reasoning about requirements evolution. In: Proceedings of the Third International Symposium on Requirements Engineering. IEEE Computer Society, Los Alamitos, CA. pp. 247–257.