Evolution of Safety-Critical Requirements Post-Launch

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 three spacecraft subsequent to launch. Only those requirement changes that resulted from post-launch anomalies (i.e., during operations) 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, post-launch requirements changes were rarely due to previous requirements having been incorrect. Instead, changes involved new requirements (1) for the software to handle rare events or (2) for the software 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.

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 three spacecraft. We distinguish these 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 computer prior to each navigational maneuver, orbital insertion around a new 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, post-launch 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 three spacecraft: Mars Global Surveyor, a mapping mission launched in November, 1996; Cassini, a mission to Saturn launched in October, 1997; and

Deep Space 1, a technology demonstration mission (of ion propulsion and remote agent technologies, among others) launched in October, 1998.

The reporting mechanism is an on-line form (called an Incident/Surprise/Anomaly, or ISA, report) that 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 later filled in 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 what NASA calls "dive-and-catch" defects, 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 two ISAs described incidents in which an inflight anomaly triggered a contingency (safe) mode or fault-protection response. In both 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

A sample of 86 ISAs in the highest criticality level from three spacecraft was analyzed. The criticality level is assigned by the project based on standard classifications [8]. Because of slight differences in the processes of the three projects regarding which fields of the anomaly reports were used, we studied all anomaly reports that met one of the following three criteria 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); or Priority = 1 (the high priority is assigned by the correcting agency indicating a "must-fix" situation) Anomalies meeting one or more of these criteria were studied and are together included under the shorthand term "critical ISAs" in this paper.

Seventeen of the 86 critical ISAs had flight software as their target, i.e., the anomaly prompted a change to the flight software. (The other 69 ISAs produced changes to procedures, ground software, documentation, etc., outside the scope of this paper.) Eight of the seventeen ISAs resulted in updates only to the code but not to requirements (e.g., bias or filter updates, adjustment of a timeout parameter, erroneous re-initialization to "on" rather than "off"). The ninth of the seventeen flight software ISAs involved a maintenance problem (an incorrect software patch). The tenth of the seventeen flight software ISAs 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. The discussion that follows focuses on the remaining seven of the seventeen 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 four 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 two other critical anomalies, a rare scenario led to an overflow. In each of these 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" [7]. Hecht further noted, "Rare events were clearly the leading cause of failures among the most severe failure categories."

3.2 New requirements to compensate for hardware and environment

Critical requirements changes were driven by changes to the hardware or environment in three cases. In one

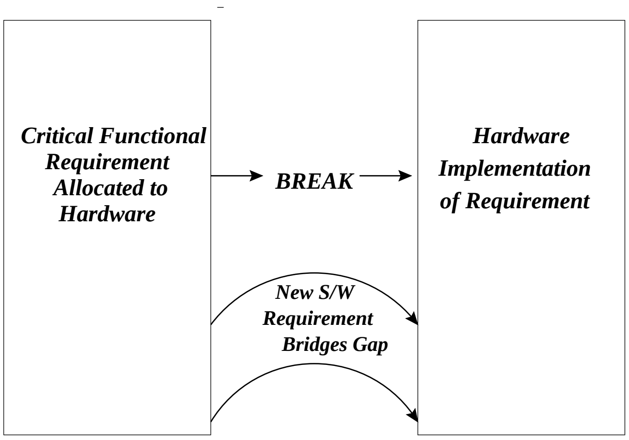

Figure 1: Software requirements change to compensate for hardware failure

case, 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 another case, 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 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 a third case, a new capability was added to the flight software in response to a damaged solar array panel that could not deploy as planned.

One issue of interest in the last two cases above 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 reprogramming 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" [11]. 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 [10].

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 [4]. 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 was posed by reviewers regarding the 69 critical ISAs that did not produce changes to flight software. Was it possible that a mechanism similar to the use of flight software to compensate for hardware problems occurred, whereby changes to ground (as opposed to flight) components were compensating for problems in flight software? The reviewers asked how many of the 69 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 69 ISAS met this criteria, and that only one of the 69 ISAs involved a change to ground software requirements. Of these six ISAs that involved flight software problems but not flight software fixes, four 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). Two 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. That is, 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 where a possible requirement has not been implemented, documented contingency studies can facilitate accurate requirements evolution when it becomes necessary during operations. The fact that four of the seven critical post-launch requirements changes were in response to rare events or planning for rare events indicates that the cost/benefit tradeoff of such hazard analyses makes them worthwhile in practice 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 is 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 [1]. Zowghi, Ghose, and Pappas provide a logical framework for reasoning about requirements evolution, also within the requirements analysis phase of development [13]. An open issue worth exploring is to what extent these techniques are also useful for analyzing the consequences of requirements evolution post-launch.

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 his case, automating the self-destruct feature of a rocket) can be structured so that the components remain unchanged while their interactions adapt to the changed requirements [3]. In our study, the requirements changes were low-level 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

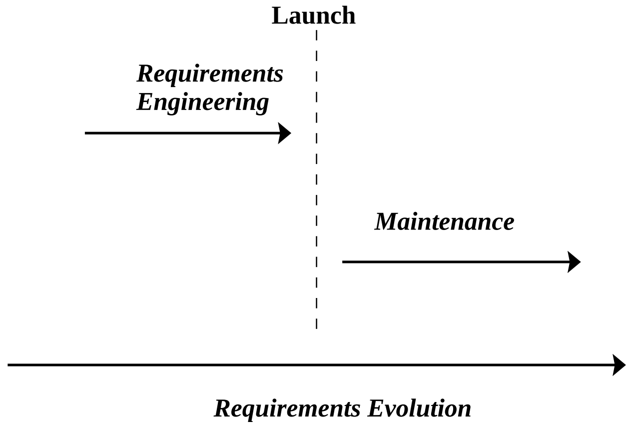

Figure 2: Continuous evolution of requirements vs. discontinuity in methodologies

the impact on stakeholder viewpoints [9]. 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.

Fickas and Feather provide a possible direction for actually reducing the unpredictability of the anomaly-induced requirements changes [5]. 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 RE-based analysis of requirements evolution and maintenance-based studies of requirements evolution.

Harker, Eason, and Dobson 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) [6]. 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 note in their recent roadmap paper that software evolution lacks a standard definition [2]. 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."

As with the previous classification scheme, the spacecraft post-launch requirement changes fit several phases. Certainly the on-board software fits the software Evolution phase. However, it also, to some extent, fits the subsequent phase, Servicing or software maturity, with its danger of loss of key personnel and information due to the length of the spacecraft mission and the planned commitment 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 [12].

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. 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.

One of the lessons learned from the study of requirements changes post-launch was that new software requirements were often needed to make the deployed software more robust against unanticipated scenarios. 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). We saw, as well, that existing maintenance models do not incorporate 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 the anonymous reviewers 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] A. I., Anton and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems," *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 19–25, 1998, pp. 157–166.

[2] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: a Roadmap," in *Foundations of Software Engineering, ICSE '00*, ed. A. Finkelstein, ACM Press, 2000.

[3] R. deLemos, "Safety Analysis of an Evolving Software Architecture," *Proceedings of the Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE '00)*, IEEE Computer Society, Los Alamitos, CA, 2000, pp. 159–167.

[4] N. E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Transactions on Software Engineering*, 26, 8, August, 2000, pp. 797–814.

[5] S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," *Proceedings of the Second International Symposium on Requirements Engineering*, York, IEEE, 1995.

[6] S. D. P. Harker, K. D. Eason, and J. E. Dobson, "The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering," *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 266–272.

[7] H. Hecht, "Rare Conditions–An Important Cause of Failures," *Proceedings of the Eighth Annual Conference on Computer Assurance*, IEEE, June, 1993, pp. 81–85.

[8] "ICAP Anomaly Process, Glossary," Safety and Mission Assurance Information Systems, Jet Propulsion Laboratory, January 17, 1997.

[9] W. Lam and M. Loomes, "Requirements Evolution in the Midst of Environmental Change: A Managed Report," *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, Florence, Italy, March 8–11, 1998, pp. 121-127.

[10] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, 1993, pp. 126–133.

[11] "Space Rescue Makes Close Encounter Possible," *JPL News Release*, July 27, 2000.

[12] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "On Low-Cost Error Containment and Recovery Methods for Guarded Software Upgrading," *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS '00)*, Taipei, Taiwan, April, 2000.

[13] D. Zowghi, A. K. Ghose, and P. Peppas, "A Framework for Reasoning about Requirements Evolution," *Proceedings of the Third International Symposium on Requirements Engineering (RE '97)*, Annapolis, MD, January 6–10, 1997, pp. 247–257.