Ongoing Requirements Discovery in High-Integrity Systems

Robyn R. Lutz, *Jet Propulsion Laboratory, California Institute of Technology, and Iowa State University*

Inés Carmen Mikulski, *Jet Propulsion Laboratory, California Institute of Technology*

Too often, we act as if we can confine requirements discovery to the requirements phase of development. For example, using the common phrase "requirements elicitation" often implies that we can, with a good process, know all software requirements at an early development phase. However, for many complicated, highly interactive, or embedded systems, especially in critical domains such as spacecraft, we continue to discover requirements knowledge into deployment and beyond.

In addition, difficulties with requirements are a well-known source of both testing and post-release defects.[1,2] Missing or erroneous requirements are also a frequent cause of accidents in deployed systems.[3] We can improve our systems' quality by means of a better understanding of the mechanisms by which we discover requirements and manage them in testing and operations.

We analyzed anomaly reports from testing and operations for eight spacecraft projects at the California Institute of Technology's Jet Propulsion Laboratory, showing that many of the anomalies during these phases involve software requirements discovery. As a result, several patterns of requirements discovery emerged. In turn, identifying these patterns leads to six guidelines for managing the ongoing requirements discovery.

Discovering new requirements and requirements knowledge continues throughout the lifetime of many high-integrity embedded systems. Understanding the mechanisms for how we discover and resolve requirements identifies guidelines to help prevent anomalies found during testing from recurring during operations.

Analyzing anomalies

An institutional, Web-based database contains the anomaly reports for the eight JPL spacecraft projects. Separate online forms exist for describing testing anomalies and operational anomalies, but they're similar. Both forms contain an anomaly's description, a subsequent analysis of the anomaly, and a description of the corrective action taken to close out the anomaly report.

The data set of testing anomalies used for the twin Mars Exploration Rover spacecraft contained 463 filled-in forms written during integration and system testing. These twin spacecraft launched in June 2003 and reached Mars in January 2004. Their two robotic rovers (see Figure 1) are currently exploring Mars to search for, among other things, evidence of past water.

The data set of operational anomalies consisted of nearly 200 anomaly reports ranked critical on seven already-launched spacecraft. Table 1 lists the spacecraft, their missions, and their launch dates. Although we analyzed all available testing anomaly reports, we analyzed only the critical anomaly reports from the deployed systems. This is because at the time of analysis the criticality rankings weren't available for all the testing anomalies. The same types of requirements discovery evident in testing caused critical anomalies in operations, which motivates continuing work in this area.

We analyzed the anomalies using an adaptation of the *Orthogonal Defect Classification*, a defect-analysis technique that Ram Chillarege and his colleagues at IBM developed.[4] ODC provides a way to extract signatures from defects and correlate the defects to attributes of the development process. Our adaptation of ODC to the spacecraft domain used four attributes to characterize each anomaly reported. The first attribute is the *activity*, which describes when the anomaly occurred. The *trigger* indicates the environment or condition that had to exist for the anomaly to surface. For example, the trigger could be a hardware-software interaction. The *target* characterizes the high-level entity that was fixed in response to the anomaly's occurrence (for example, "flight software"). Finally, the *type* describes the actual fix that was made (for example, "function/algorithm").

Anomaly reports document defects as well as any behavior that the testing or operational personnel don't expect. The anomaly reports are thus a rich source of latent requirements (where the software doesn't behave correctly in some situation due to incomplete requirements) and requirements confusion (where the software behaves correctly but unexpectedly).

The anomaly reports showed two basic kinds of requirements discovery:

- New, previously unrecognized requirements or requirements knowledge (such as constraints)
- Misunderstandings by the testers or users regarding existing requirements

Table 2 describes how the ODC target and type identified the various ways to handle the requirements discovery:

- *Software change*. Implement a new requirement in software.
- *Procedural change*. Enforce a new requirement with a new operational rule.
- *Document change*. Solve a requirements misunderstanding by improving the documentation.

Figure 1. A Mars Exploration Rover. (photo courtesy of the Jet Propulsion Laboratory and the California Institute of Technology)

Table 1

Flight software systems

Project	Launch	Mission
Galileo	1989	Jupiter
Mars Global Surveyor	1996	Mars
Cassini/Huygens	1997	Saturn/Titan
Mars Climate Orbiter	1998	Mars
Deep Space 1	1998	Flight-test new technologies
Mars Polar Lander	1999	Mars
Stardust	1999	Comet Wild 2

Table 2

Orthogonal Defect Classification signatures investigated

Category	ODC target	ODC type
Incomplete requirements and software fix	Flight software	Function/algorithm (primarily)
Unexpected requirement interactions and procedural fix	Information development	Missing procedures or procedures not followed
Requirements confusion and documentation fix	Information development	Documentation or procedure
Requirements confusion and no fix	None/unknown	Nothing fixed

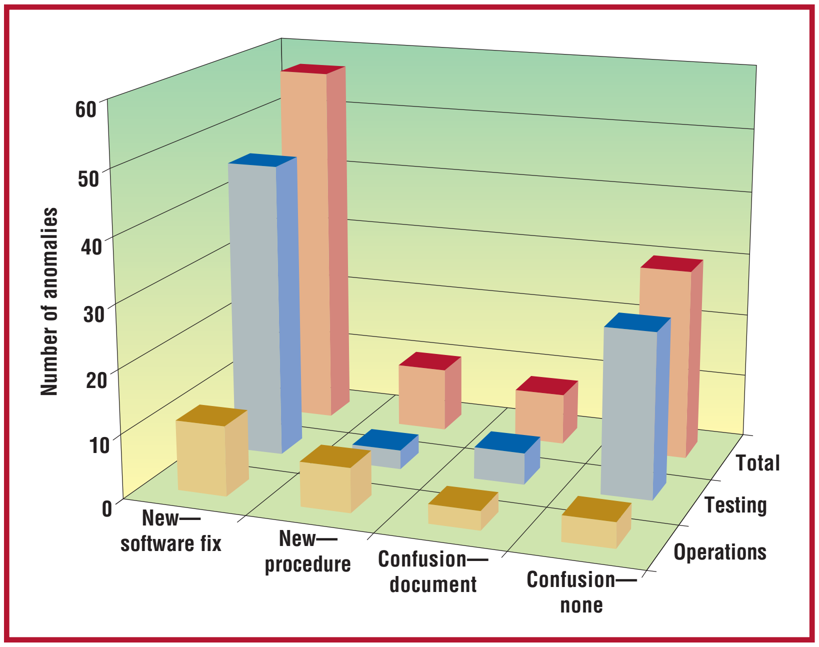

Figure 2. Requirements discovery classification.

- *No change*. Make no change because the software worked correctly and the behavior just confused the user.

Figure 2 provides an overview of these four mechanisms in the anomaly reports. The x axis shows the four mechanisms in the order we described earlier. The y axis displays the number of anomaly reports characterized by each mechanism. The z axis distinguishes the operational phase (blue) from the testing phase (red) and provides a total for each mechanism (yellow). For example, Figure 2 shows that in both testing and operations implementing new software requirements is the most frequent of the four mechanisms.

Discovering new requirements

We found two mechanisms for handling requirements discovery of a previously unidentified requirement or of previously unknown interactions among requirements. The projects resolve such anomalies either by changing the software or by changing the procedures to implement the new requirement.

Discovery resolved by changing software

In this mechanism, the first that we found, the anomaly report describes the discovery of new requirements knowledge. The projects corrected the anomaly by implementing the new requirement in the onboard flight software. For example, in one case, analyzing a testing anomaly revealed a previously unidentified requirement to reenable a reset driver during a reboot.

Projects implemented newly discovered requirements during operations by uploading a patch, or new software, to the spacecraft. In this article, we don't consider requirements involved in the planned evolution or scheduled system maintenance.. This is because software is regularly uploaded to the spacecraft before a new mission phase to control that phase's activities. For example, as the mission passes from cruise to planetary encounter, new software requirements are implemented in the flight software. However, these planned updates don't routinely reflect the discovery of new requirements. This contrasts with the unplanned changes to requirements prompted by critical anomalies during operations that we studied here.

In testing, 218 anomaly reports had the ODC target "flight software." As Figure 2 shows, 46 of these involved incomplete or missing requirements resolved by software changes. These missing requirements were either unidentified or new requirements. Many describe timing or initialization issues arising from the interaction among software components or between software and hardware. For example, in one such anomaly, a new requirement became evident during testing. The new requirement was for the initial state for a component to wait for a motor's initial move to complete. In another case, testing identified an off-nominal scenario in which certain interfaces had to be temporarily disabled to achieve correct behavior.

In operations, 44 anomaly reports had the "flight software" target. Figure 2 shows that 11 of these involved missing or incomplete requirements. In four cases, a new software requirement, implemented by uploading a change to the flight software, compensated for a hardware component's failure or degradation. For example, when a damaged solar array couldn't deploy correctly, a new flight software requirement added needed functionality in response. In another anomaly, noisy transducers caused excessive resetting of hardware components, a risk to the system. In response, a new flight software requirement compensated for the noise.

In seven other cases, the anomaly was resolved with a new software requirement to handle an unusual event or scenario. In these cases, the requirements discovery involved unforeseen system behavior that was resolved by requiring additional fault tolerance for similar, future incidents. In one anomaly, for example, an unexpected outflow of debris temporarily blinded the spacecraft, making it difficult to determine its position in space. Consequently, new software requirements made the spacecraft robust against similar future events.

In testing and operations, requirements discovery was often resolved by changing the software in the systems studied. In testing, new requirements emerged most often from subtle dependencies among software components or between the software and hardware. In operations, rare scenarios or hardware degradations caused critical anomalies resolved by urgent, unplanned software requirements changes.

Discovery resolved by changing procedures

The second mechanism also involves new requirements, but in these cases, the projects implemented the new requirement via an operational procedure. Newly discovered requirements or interactions are dealt with by changing the process external to the software so that the software doesn't reach the anomalous state again. Such anomalies usually involve unexpected requirements interactions detected during testing or operations. Analyzing the anomaly sometimes results in a new requirement that certain activities be performed in a specific order (for example, to prevent a race condition) or in a specific timing relationship. For example, in one case, a software fault monitor issued redundant off-commands from a particular state reached during testing. The developers decided to prevent these redundant commands procedurally by selecting limits that would avoid that state in the future.

In testing, 30 anomaly reports had an "information development" target and a "missing or incomplete procedure" type. Of these, three anomaly reports displayed this mechanism. In post-launch operations, seven critical anomalies had the same target and type. In each case, a procedure implemented the new requirements knowledge.

For example, one anomaly identified the need to recover the commands remaining to be executed after an abnormal termination occurred. This requirement was resolved by creating a new operational procedure to respond to similar situations in the future. In another case, a problem occurred when two streams of data were sent simultaneously. The anomaly revealed a latent requirement that hadn't been previously recognized to ensure that only one stream of data at a time be transferred. Again, this was handled procedurally. In a third anomaly, the software behavior was incorrect in that a maneuver was erroneously performed twice rather than once. This occurred when the software was loaded to memory too soon—that is, to an area of memory that was currently active. The fix was to add a procedure to enforce a new requirement preventing the configuration problem's recurrence.

Handling a new requirement via a procedural change avoids the cost and risk of updating complex, safety-critical flight software. However, allocating requirements to procedures carries the risk that the procedure won't execute correctly on each occasion when the situation requires it. Resolving critical anomalies such as these via changes to procedures places high dependence on the requirements knowledge and the operational personnel's motivation. The fact that several operational anomalies have the "procedures not followed" type indicates that such dependence can be a risky strategy.

The anticipated length of a system's lifetime might be another factor in deciding whether to handle a requirement procedurally. For a relatively short-lived system (such as a mission to

In testing, new requirements emerged most often from subtle dependencies among software components or between the software and hardware.

Operational anomalies due to requirements confusion occasionally resulted in improved documentation of a procedure.

Mars, measured in months), a change to operational procedures might make sense. For a relatively long-lived system (such as a seven-year trip to Saturn followed by a multiyear scientific mission), there will inevitably be personnel turnover and ensuing loss of requirements knowledge, potentially adding risk.

Discovering requirements confusion

The third and fourth mechanisms differ from the first two in that they involve the discovery of a requirements confusion rather than a missing or incomplete requirement. In these cases, the software works correctly, but the testing or operational personnel are surprised by its behavior.

Confusion resolved by changing documentation

In this mechanism, projects resolved an anomaly by fixing the documentation. This might mean updating the design document to better explain the rationale for the requirements, adding a flight rule to describe the required preconditions or constraints on issuing a particular command, or documenting a hardware unit's unexpected effect on software behavior in the idiosyncrasies list. In each case, the documentation changes to better explain the required behavior and the requirements rationale. The goal is to prevent this anomaly's recurrence.

For example, one testing anomaly was caused by testing personnel's incorrect assumption that some heaters would remain off as the software transitioned between two specific modes. The anomaly was resolved by correcting the design documentation to describe the software requirement implemented by another component to turn the heaters off when this transition occurred.

Analyzing problem reports from testing showed 28 anomalies with an "information development" target and "documentation" type. Five of these involved the testers making incorrect assumptions about the requirements. These misunderstandings emerged during testing when correct software behavior didn't match the testers' expectation. Such testing reports were handled by correcting the source of the misunderstanding via improved documentation.

Requirements confusion also caused critical operational anomalies in the systems. Analysis identified three such anomalies with an "information development" target and "documentation" type. Of these, one anomaly involved requirements confusion. In that case, the anomaly reported a drop in battery power resulting from a requirements misunderstanding of the behavior initiated by using a command. The corrective action was to document the required behavior and associated command in an operational flight rule.

Operational anomalies due to requirements confusion occasionally resulted in improved documentation of a procedure. In these cases, communication of a known constraint improved. For example, in one anomaly, testers didn't understand a required precondition for a calibration (that the instrument be in an inertial mode). They avoided the problem in future calibrations by documenting this requirement in the systems checklist.

Confusion resolved without any change

In this case, anomaly reports are false positives, reporting a problem when the software actually behaved correctly and according to the specified requirements. In each of these anomalies, the projects subsequently determined that no change was necessary. For example, in some cases, no change was made because the situation couldn't recur in the rest of the mission.

There were 64 testing anomalies with a "none/unknown" target and a "nothing fixed" type. In most cases, resolving the anomaly report without a fix was appropriate. For example, several anomalies referred to problems that were no longer relevant (for example, the current build removed the issue). However, analyzing testing-problem reports shows that in 26 anomalies with this ODC signature, the same requirements confusion might recur in operations. These merit additional attention.

For example, in one anomaly, the tester assumed that a telemetry (data) channel provided the current value of a counter, whereas the channel instead provided the counter's high-water mark (the highest value yet recorded for the counter). So, even when the counter was reset, the telemetry value remained constant. The requirements rationale was sound—that the fault-protection software needs information regarding the worst case over a time interval. However, the tester's misunderstanding was reasonable and indicated that a similar erroneous assumption might be possible later.

On the basis of our experience with this type of anomaly,[5] we recommend that when both the situation in the anomaly report and the requirements confusion can recur, developers should attempt to prevent future occurrences after deployment. Required behavior that surprises the testers should not also surprise the users.

In this type of situation, testing can be a crystal ball into operations. System behaviors that confuse the test team might also confuse operators down the road. For example, in one testing incident, the tester expected that commands issued to a hardware unit when it was turned off would be rejected. Instead, the commands unexpectedly executed when the component was rebooted. This behavior was necessary in the system's context, and it matched the specifications. However, tardy execution of the commands was understandably not the behavior that the tester expected. If this confusion recurred in operations, it could cause a serious problem. In this case, the test incident acted as a crystal ball into future possible operational problems. Calling attention to the mismatch between software behavior and operator assumptions helped reduce the possibility of this confusion recurring.

Also, four cases of requirements confusion in operations warranted clarifying the documentation to avoid future anomalies due to the same requirements confusion. Interestingly, in all four anomalies, the projects decided that they didn't need to take corrective action to remedy the requirements misunderstanding for the system in question. However, in all four cases, the anomaly report noted that the misunderstanding could also occur on future spacecraft. That is, the operations teams perceived the requirements misunderstanding as a recurrent risk on other systems. This focus on the next-generation systems by operational personnel suggests the need for defect analysis to broaden the perspective from considering a single system to considering a set, or family, of similar systems (in this case, interplanetary spacecraft). These results suggest that better reuse of knowledge regarding past requirements confusions might forestall similar requirements confusions on other systems in the same product family.

Guidelines for ongoing requirements discovery

The experiences we report here show that requirements discovery caused anomalies during both testing and operations in the systems we studied. Furthermore, similar mechanisms for requirements discovery and resolution were at work in testing and operations. Given that requirements discovery continues throughout a system's lifetime, several guidelines for mining anomaly reports are evident when managing this evolution:

- *Plan for continuing requirements engineering activity*. Our experience confirms the value of continued requirements engineering activities throughout a system's lifetime: maintenance of requirements rationales; explicit traceability from requirements to procedures, as well as to software; and analysis of requirements interactions. (Eric Dubois and Klaus Pohl have described this problem as "continuous requirements management."[6])
- *Mine anomaly reports*. Bug reports from testing and operations are a rich, under-used source of information about requirements.
- *Use reports of near misses and false positives to prevent problems*. Software behavior that surprises the testing teams might also surprise the users. We too often ignore these mismatches between actual and expected behavior when the software behaves correctly. This throws away a chance to avoid future requirements confusions by improving documentation or training now.
- *Implement newly discovered requirements by updating software rather than procedures*. Asking users to avoid a certain scenario is overly optimistic if the consequences of error are severe.
- *Flag patterns of requirements confusion for extra attention*. Analyzing anomalies across the eight spacecraft systems reveals certain recurring patterns of misunderstandings. Possible responses include adding these patterns to inspection checklists, test cases, and assertion checking.
- *Take a product line perspective*. Many requirements discoveries that occur during testing and operations were described in the anomaly reports as also applying to other, similar systems in the same product line. Cross-correlating requirements discoveries among these systems can reduce anomalies across multiple systems.

Required behavior that surprises the testers should not also surprise the users.

About the Authors

Robyn R. Lutz is a senior engineer at the Jet Propulsion Laboratory and an associate professor of computer science at Iowa State University. Her research interests are in safety-critical product lines, defect analysis, and the specification and verification of requirements, especially for fault monitoring and recovery. Contact her at 226 Atanasoff Hall, ISU, Ames, IA 50011; rlutz@cs.iastate.edu.

Inés Carmen Mikulski is a senior engineer at the Jet Propulsion Laboratory. Her research interests center on developing a project- and institution-level metrics program as part of JPL and NASA's software quality improvement initiative. She received her MS in mathematics from XXXXXXXXXXXXXXXXX. Contact her at JPL, MS 125-233, 4800 Oak Grove Dr., Pasadena, CA 91109; ines.c.mikulski@jpl.nasa.gov.

We are working to apply these guidelines to future NASA projects both on spacecraft and ground-based systems. One such collaborative application will be to the software controllers for the network of antennas used to communicate with Earth-orbiting and deep-space missions.

Acknowledgments

We thank Daniel Erickson and the Mars Exploration Rover engineers and test teams for their assistance and feedback, and the reviewers for several useful suggestions. An early version of this article was presented at the 11th IEEE Conference on Requirements Engineering in September 2003. NASA's Office of Safety and Mission Assurance Center Initiative UPN 323-08 funded this research, which we performed at the Jet Propulsion Laboratory under a contract with NASA. National Science Foundation grants CCR-0204139 and CCR-0205588 partly support Robyn Lutz's research.

References

1. R. Lutz and I.C. Mikulski, "Operational Anomalies as a Cause of Safety-Critical Requirements Evolution," *J. Systems and Software*, vol. 65, no. 2, Feb. 2003, pp. 155–161.
2. S. Lauesen and O. Vinter, "Preventing Requirements Defects: An Experiment in Process Improvement," *Requirements Eng. J.*, vol. 6, no. 1, Feb. 2001, pp. 37–50.
3. N. Leveson, *Safeware*, Addison-Wesley, 1995.
4. R. Chillarege et al., "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Software Eng.*, vol. 18, no. 11, Nov. 1992, pp. 943–956.
5. R. Lutz and I.C. Mikulski, "Requirements Discovery during the Testing of Safety-Critical Software," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 578–583.
6. E. Dubois and K. Pohl, "RE 02: A Major Step toward a Mature Requirements Engineering Community," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 2003, pp. 14–15.

Fill?