Requirements Discovery During the Testing of Safety-Critical Software

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

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

Abstract

This paper describes the role of requirements discovery during the testing of a safety-critical software system. Analysis of problem reports generated by the integration and system testing of an embedded, safety-critical software system identified four common mechanisms for requirements discovery and resolution during testing: (1) Incomplete requirements, resolved by changes to the software, (2) Unexpected requirements interactions, resolved by changes to the operational procedures, (3) Requirements confusion by the testers, resolved by changes to the documentation, and (4) Requirements confusion by the testers, resolved by a determination that no change was needed. The experience reported here confirms that requirements discovery during testing is frequently due to communication difficulties and subtle interface issues. The results also suggest that "false positive" problem reports from testing (in which the software behaves correctly but unexpectedly) provide a rich source of requirements information that can be used to reduce operational anomalies in critical systems.

1. Introduction

This paper describes the role of requirements discovery during the testing of a safety-critical software system. Difficulties with requirements have been repeatedly implicated as a source of both testing defects [2, 7] and accidents in deployed systems [3, 10]. In an effort to improve our understanding of how requirements discovery occurs during testing, and how such discoveries are resolved (or are not resolved) prior to deployment, we investigated the requirements-related problems reported during testing of a safety-critical system currently under development. Analysis of the problem reports generated during integration and system testing of the software distinguished four common mechanisms for requirements discovery and resolution:

(1) *Incomplete requirements, resolved by changes to the software.* As often occurs, testing caused several previously unidentified requirements to surface. These new requirements usually involved complicated interface issues between software components or between hardware and software. Several of the incomplete requirements involved fault protection, of special concern in safety-critical systems.

(2) *Unexpected requirements interactions, resolved by changes to the operational procedures.* A closely related mechanism for requirements discovery was the identification during testing of unexpected interactions among the existing requirements. Typically, these interactions resulted in new required sequencing of activities when the interleaved processes unexpectedly caused incorrect behavior or did not achieve the required precondition for correct execution of the software.

(3*) Requirements confusion by the testers, resolved by changes to the documentation.* Testing revealed some significant misunderstandings on the part of the testers regarding what the requirements actually were. In these cases the software worked as required and the requirements were correct, but the software's behavior was unexpected. The corrective action was not to fix the software, but to enhance the documentation in order to better communicate the required software behavior or requirements rationale.

(4) *Requirements confusion by the testers, resolved by a determination that no change was needed.* In this mechanism testing also revealed a gap in requirements understanding. However, the problem report was judged to be a "false positive," i.e., indicating failure where the software in fact behaved correctly. We found that in some cases where the software behaved correctly but unexpectedly, an opportunity was missed to prevent similar, subsequent requirements confusion by the operators of the deployed system. We propose some guidelines for distinguishing and responding to such situations.

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. It was funded by NASA's Code Q Software Program Center Initiative, UPN 323-08. The first author's research is supported in part by National Science Foundation Grants CCR-0204139 and CCR-0205588.

The experience reported here suggests that problem reports generated during testing are an underused source of information about potential requirement-related anomalies that may occur after the software is deployed. Test defect reports provide a unique source of insights into future users' gaps in domain knowledge, misidentification of requirement rationales, and erroneous assumptions regarding required sequences of activities. In this limited sense, testing problem reports may provide a preview of some possible operational problems. The main contributions of the paper are (1) to identify the common mechanisms by which requirements discovery and resolution occurred during testing, and (2) to report the lessons learned regarding how such discoveries can be better used to reduce future requirements anomalies in the deployed system.

The rest of the paper is divided into sections as follows. Section 2 describes the approach used to investigate requirements discovery during testing. Section 3 discusses and evaluates the results in the context of some illustrative examples. Section 4 briefly compares the experience described here to others' findings. Section 5 summarizes the lessons learned.

2. Approach

The data for this analysis consisted of the 171 completed Problem/Failure Reports (PFRs) written by project test teams during integration and system testing of the Mars Exploration Rovers (MER). MER, to be launched in 2003, will explore Mars with two robotic rovers equipped to search for evidence of previous water. The size of MER's flight software is roughly 300K Lines of Code, implementing approximately 400 software requirements of varying degrees of granularity. Although the software was delivered in a series of builds, we do not distinguish here among the builds due to the relatively small number of PFRs.

The on-line Problem/Failure Reports (PFRs) filled out by the project consist of three parts. The first part describes the problem and is filled out by the tester when the problem occurs. The second part is filled out by the analyst assigned to investigate the problem. The third part is filled in later with a description of the corrective action that was taken to close out the problem.

The approach we selected for the analysis of the PFRs was an adaptation of Orthogonal Defect Classification (ODC) [1]. ODC provides a way to "extract signatures from defects" and to correlate the defects to attributes of the development process (Fig. 1). Our ODC-based approach uses four attributes to characterize each PFR: Activity, Trigger, Target, and Type. The Activity describes where the defect surfaced, e.g., Integration Test or System Test. The Trigger describes the environment or condition that had to exist for the defect to appear. In the testing environment, the trigger was usually the testing of

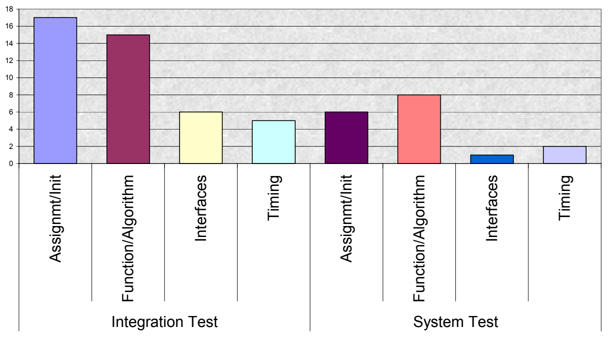

Figure 1. Types of Corrections for Testing Reports

a single command or of a capability sequence (i.e., a software requirement scenario). The Target describes the high-level entity that was fixed in response to the problem report, e.g., Flight software, Ground software, etc. The Type describes the actual correction that was made, i.e., "the meaning of the fix" [1].

The two authors classified the PFRs using the adapted ODC. Both of us have experience on flight projects at JPL but neither are directly involved with the testing of the MER software. MER engineers generously assisted us with answers to our process and domain questions.

Following the ODC approach, we defined each classification attribute and the possible values it could take in a document that was reviewed by MER project personnel. Adaptation of the standard ODC categories to the spacecraft domain was driven by the need to capture core properties of the anomalies seen during testing. In order to improve repeatability and reduce bias, the process of classification involved three steps in which (1) each analyst separately classified the set of anomalies, (2) inconsistent classifications were highlighted and each analyst had an opportunity to correct any clear errors in her own classifications (e.g., missing fields), and (3) they analysts jointly reviewed the remaining inconsistencies and resolved them through discussion. A detailed description of the classification process and of efforts to remove bias is provided in [9].

The work reported here is part of a multi-year pilot study to reduce the number of safety-critical software anomalies that occur post-launch. This paper reports the first experience using the adapted ODC technique on a spacecraft currently under development. The motivation was to mine the testing problem reports for insights into how requirements discovery during testing can be used to forestall or mitigate some critical software anomalies during operations.

3. Results and analysis

We here describe each of the four mechanisms for requirements discovery and resolution identified during analysis of the Problem/Failure Reports (PFRs) generated in integration and system testing of the spacecraft software. A subsection describes each mechanism in terms of the ODC classification values that characterize it, provides a more in-depth causal analysis of some typical examples, and evaluates the adequacy of the corrective action taken to resolve the requirements discovery.

3.1 Incomplete requirements, resolved by changes to the software

Sixty-five of the completed 171 integration and system testing PFRs were resolved by a change to the flight software (Fig. 2). In ODC terms, the Target for these sixty-five PFRs was "Flight Software." Twenty-three of the Flight Software PFRs had an ODC Type of "Assignment/Initialization." These PFRs were resolved by changes to parameters in the light of new system knowledge. They entailed discovery of new requirements knowledge, but not of new functional requirements. Two typical examples of these PFRs are, in one case, a change to the value of the variable "max" to avoid unintended triggering of fault protection and, in another case, a change to require that a component come up disabled rather than enabled after a reboot.

Another twenty-three of the sixty-five Flight Software testing PFRs had an ODC Type of "Function/Algorithm." Some of these changes involved design or implementation issues such as testing of functions not yet delivered in the current build. However, the PFRs of interest to us from a requirements perspective are the ten that entailed more substantial changes to the flight software as the result of knowledge gained during testing.

Each of these ten PFRs was resolved by requiring a new software function. Many of the corrective actions taken to close these PFRs involved additional reasonableness checks on preconditions and post-conditions. Several involved startup/restart scenarios, or the correct triggering of recovery software. New requirements included an additional health check, a parameter validation check, an inhibit to checks of disabled software, distinguishing unavailability from non-response of a unit, turning off encoding in some cases, ignoring false out-of-order messages, providing a new capability to copy a rate to a register, an additional check so a warning does not occur in a shutdown mode, and a new capability to command a hardware unit.

An additional seven of the Flight Software PFRs had an ODC Type of "Timing," and seven more had an ODC Type of "Interfaces." In these types, as well, the role of testing in the discovery of new requirements was evident. Due to space constraints, we only mention briefly that several resulted in new requirements to insert delays in the software to compensate for interface delays. It is

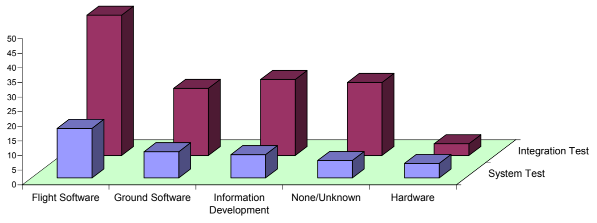

Figure 2 . Fixing Testing Problem Reports

worth noting that no PFRs documented extra requirements (where the flight software did more than it should).

3.2 Unexpected requirements interactions, resolved by changes to the operational procedures

The previous subsection described new requirements that were discovered during testing and resolved by changes to the flight software. In this subsection we describe unexpected requirement interactions that were discovered during testing and fixed, not by changes to the software, but instead by changes to the procedures that will constrain operational activities.

This mechanism for requirements discovery tended to involve emerging requirements, not discovered until testing, on the sequencing or timing of activities in interfacing software components or software/hardware interfaces. The ODC Target for these thirteen PFRs was "Information Development" and the ODC Type for these PFRs was "Missing or Incomplete Procedures." This second mechanism is a special case of the incomplete requirements described above, involving new knowledge and requirements that must be enforced on interactions. However, this mechanism differs from the first mechanism described above in that achievement of the new requirement is here allocated to procedures rather than to software.

Most of these PFRs dealt only with testing procedures and were not relevant to operations or maintenance. However, three of them involved discovery during testing of unexpected requirements interactions. In each of these three cases, responsibility for the requirement was allocated to operations. For example, in one PFR testing revealed that unless a spacecraft component was re-calibrated before use, it triggered fault-protection software. The discovery of this requirement for sequential activities (first calibrate, then use) was allocated to an operational procedure.

In another, a tester observed that, contrary to expectations, an off command was issued redundantly by a software fault monitor. Analysis showed that this behavior was correct, but idiosyncratic. The corrective action was to avoid these redundant commands during operations by carefully selecting the high and low limits to preclude the state observed in testing. It is easy to see how, even with a documented procedure in place, this situation might recur in operations.

This third mechanism for requirements discovery is of interest in preventing operational anomalies because corrections made to procedures still depend on the correct implementation of the procedure by the operator of the deployed system each time the relevant scenario arises. We were thus interested in whether some of the new requirements for constraining interactions, levied on the operational procedures, might be better handled in software. Given the small number of PFRs in the study, no conclusion was appropriate. However, the examples suggest that in long-lived systems, the tradeoff between easy but operator-dependent procedural fixes and more costly but operator-independent software fixes should be considered.

3.3 Requirements confusion by the testers, resolved by changes to the documentation

The previous two subsections both described requirements discovery mechanisms in which the testers' expectations were consistent with the required software behavior. Testing revealed missing requirements that had to be added in order to achieve the correct, and expected, behavior. The requirements discovery mechanism described in this section is different in that the testers' expectations regarding the required software behavior were incorrect. The resolution was to try to remove the source of the testers' confusion by improving the documentation of the existing requirements and their rationale.

Fourteen of the 171 testing PFRs were resolved by changes to the documentation. The ODC Target for these PFRs was "Information Development" and the ODC Type was "Documentation." (Only PFRs that changed just documentation but not software or procedures are labeled this way).

Four of the PFRs of type "Documentation" revealed erroneous requirements assumptions by the testers. For example, in one case, the tester incorrectly assumed that certain heaters remain on during the transition from one mode to another, as the spacecraft transitions from the pre-separation mode of the Mars lander to the entry/descent mode (as the lander enters the Martian atmosphere). The tester's assumption was reasonable but incorrect. In fact, there is a software requirement on another component to turn the heaters off when this transition occurs. Documentation of this fact was added to the Functional Design Document and the procedure writers were notified of the update in order to correct the misunderstanding prior to launch.

In these PFRs it was requirements confusion, rather than new requirements that were discovered during testing. The perceived inconsistency between the test results and the required behavior was inaccurate. The corrective action was not to fix the software but the source of confusion. This resulted in improved communication of the rationale for the existing behavior in the existing project documentation

3.4 Requirements confusion by the testers, resolved by a determination that no change was needed.

The final mechanism for requirements discovery is similar to the previous one except that no fix is made, even to documentation. Thirty of the 171 testing PFRs have an ODC Target of "None/Unknown" and an ODC Type of "Nothing Fixed." The reason that nothing was fixed is that these PFRs were "false positives," raising an alarm when nothing was broken. Our interest in investigating this mechanism was to see if any of these PFRs described requirements confusion or requirements interactions that could potentially recur in flight operations. If so, it might be that some change to documentation or procedure was indicated.

As expected, for most of the PFRs there was, in fact, nothing to fix. For example, thirteen of the thirty PFRs referred to problems that were no longer relevant (e.g., the current build removed the issue); two were clearly one-time operator errors (e.g., misreading the test results); and three were relevant only to the test environment but not to flight. However, eight of the thirty raised possible flight concerns, although in each case the software worked as required. We describe several of these more fully here, since they support our claim that false positives encountered during testing often provide a useful window into latent requirements misunderstandings during operations.

For example, in one case the PFR reported as an error that commands issued when a remote unit was off were not rejected as expected, but instead were completed when the unit rebooted. Although the software operated correctly, the PFR revealed a gap in understanding of the rationale for the software's required behavior (a gap, by the way, that was shared by the analysts). Since this requirements confusion could apparently reappear in a post-launch operational scenario, it may merit additional documentation to preclude a similar mistake by an operator.

Another PFR of Type "Nothing Fixed" describes a situation in which one component, attempting to communicate with another component, received warning messages indicating that an invalid response had occurred. In fact, the communication attempt happened to occur during a few-millisecond timeout that takes place in some particular scenarios. This behavior is, in fact, correct and required, and subsequent communication attempts will be normal. However, the effect of the timeout is rather subtle.

In a third example, the tester incorrectly assumed that a telemetry (data download) channel output the value of a counter when the channel instead provided the value of the counter's high-water mark (the highest value yet recorded for the counter). Thus, even when the counter was reset, the telemetry value remained constant. The requirements rationale is sound -- that the fault-protection software needs information regarding the worst case over a time interval, not just the current snapshot of a frequently reset counter. However, the requirements misunderstanding by the tester is reasonable and suggests that a similar erroneous assumption might be possible later.

Testing PFRs often provide detailed descriptions of sequences of input, states, error messages, and even partial dumps in order that the test scenario can later be duplicated. This level of detail is extraordinarily useful in allowing an analyst to pinpoint not only whether an error has occurred but also the source of any confusion regarding the required behavior. Incorrect assumptions (e.g., about the effect of specific commands on the state of the system) and gaps in domain knowledge (e.g., of hardware idiosyncrasies or transients) can often be identified from the details in the problem reports.

3.5 Implications for testing

Given limited project resources (in terms of schedule and budget), should these "false-positive" testing reports be documented further? Based on the problem reports seen here and on past experience with operational anomalies [8, 9], we suggest the following guideline: *if the situation described in the problem report could recur in operations, and if the requirements confusion or misunderstanding of required interactions could also recur in operations, then the problem report may merit additional attention.* Using this guideline, each of the three examples above would have involved additional corrective actions.

For example, one such false-positive PFR recorded a perceived discrepancy between two time tags that should be identical. In fact, the software worked as required. The two time tags were two different representations of the same time (cumulative number of seconds since a standard base time and the translation of that value to the current UTC, the Universal Time). This misunderstanding by the tester is one that could be repeated by an operator or maintenance programmer with conceivably hazardous effect, so may merit additional documentation.

Experience with the MER testing PFRs also suggests that PFRs related to certain critical activities always merit additional attention even if the PFR merely records requirements confusion. Thus, if the testing PFR involves fault protection software, critical control software, critical maneuvers or activities (e.g., engine burns), or critical mission phases (e.g., insertion of the spacecraft into a planetary orbit), then the problem report should take into account measures to prevent the required behavior that surprised the testers from later surprising the operators.

3.6 Implications for operations

False-positive problem reports from testing (when the software behavior was correct but unexpected, so nothing was fixed) have significant value in a development organization if the requirements confusion or emerging domain knowledge that led to them can be identified and remedied. Especially in a long-lived spacecraft system where turnover of operational personnel is to be expected, loss of knowledge regarding requirement rationale can be substantial. It appears that testers' requirements confusion may provide some small degree of "crystal ball" insight into possible future post-release misunderstandings and, thus, the opportunity to mitigate those gaps, whether by documentation, training, or changes to software or procedures. Techniques to trace the requirements misunderstandings encountered during testing into operations are at this time an open problem.

Some results from a recent study by the authors confirm that the requirements discovery mechanisms found in testing can affect safety-critical operations. This ODC-based study profiled 199 safety-critical software anomalies recorded post-launch on seven spacecraft [9]. One of the surprises to emerge from that study was that some procedures needed for post-launch operations were not in place, and that these omissions contributed to 21% of the safety-critical anomalies. Another finding related to requirements discovery was that in most of the anomalies of Type "Nothing Fixed" (14% of the total), what was originally reported as a safety-critical anomaly was in fact the required behavior of the spacecraft, i.e., requirements confusion. Better understanding of the various requirements-discovery mechanisms in testing has as its primary goal to prevent slippage of requirements-related testing problems into operations.

4. Related Work

Most work on the analysis of testing defects has focused on measuring the quality or readiness of the software for release (see, e.g., [2]). In our study, the focus was instead on how to use the requirements discoveries made during testing (either of incomplete software or of incorrect human assumptions) to reduce critical defects during operations.

The results reported here tend to confirm the central role that Hanks, Knight, and Strunk have found for problems communicating domain knowledge [3]. Weiss, Leveson, Lundqvist, Farid, and Stringfellow specifically implicate requirements misunderstanding in several recent disasters, stating, "software-related accidents almost always are due to misunderstanding about what the software should do" [10]. In this regard, the instances of requirements confusion found here are somewhat similar to the examples of mode confusion by pilots and other operators that Leveson and others have described.

Previous work by one of the authors found that safety-related testing defects on two earlier spacecraft arose most commonly from (1) misunderstanding of the software's interfaces with the rest of the system and (2) discrepancies between the documented requirements and the requirements needed for correct functioning of the system [7]. A recent study by Lauesen and Vinter found similar results for non-critical systems, with slightly more than half the defect reports being requirements defects and the major source being missing requirements [5]. Several defect classification methods (see, e.g., [6, 1]) include communication failures as root causes or as defect triggers. However, these approaches tend not to distinguish requirements confusion in which the reported software behavior is actually correct from other kinds of communication failures, as we found helpful here. These studies also focus on ways to prevent requirements defects from reaching testing, whereas we were more interested in how to use testing problem reports to prevent defects from reaching operations.

Harold recently suggested the use of "test artifacts" for software engineering tasks in describing future directions for work, but added that "this research is in its infancy" [4]. The experience described here suggests that testing problem reports may be useful test artifacts that can be more effectively mined for requirements insights to reduce post-deployment anomalies.

5. Conclusion

The results reported here distinguish four common mechanisms for requirements discovery and resolution during the integration and system testing of a safety-critical software system. One of the lessons learned was that requirements discovery during testing is frequently due to communication difficulties and subtle interface issues. Requirements discovery in testing thus drove changes not only to the software but also to the operational procedures and to the documentation of requirements rationale. Another lesson learned was that false-positive problem reports from testing (where the software behaves correctly but unexpectedly) provide a rich source of insights into potential requirements-related anomalies during operations. This information may be able to be used to reduce operational anomalies in critical systems.

Acknowledgments. The authors thank Daniel Erickson and the Mars Exploration Rover engineers and test teams for their assistance and feedback.

References

[1] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurements, *IEEE Trans on SW Eng*, Nov. 1992, pp. 943-956.

[2] S. Gardiner, ed. *Testing Safety-Critical Software*, Springer-Verlag, London, 1999.

[3] K. S. Hanks, J. C. Knight, and E. A. Strunk, "Erroneous Requirements: A Linguistic Basis for Their Occurrence and an Approach to Their Reduction," *Proc. 26th NASA Goddard SW Eng Workshop*, IEEE, Greenbelt, MD, Nov., 2001.

[4] M. J. Harold, "Testing: A Roadmap" in *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, New York, 2000.

[5] S. Lauesen and O. Vinter, "Preventing Requirements Defects: An Experiment in Process Improvement," *Requirements Engineering Journal*, 2001, pp. 37-50.

[6] M. Leszak, D. E. Perry, and D. Stoll, "A Case Study in Root Cause Defect Analysis," *Proc 22nd Intl Conf SW Eng (ICSE'00)*, IEEE CS Press, Los Alamitos, CA, 2002, pp. 428-437.

[7] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc IEEE Intl Symp Req Eng*, IEEE CS Press, 1993, pp. 126-133.

[8] R. Lutz and I. C. Mikulski, "Operational Anomalies as a Cause of Safety-Critical Requirements Evolution," *The Journal of Systems and Software*, to appear.

[9] R. Lutz and I. C. Mikulski, "Empirical Analysis of Safety-Critical Anomalies During Operations," submitted to *IEEE Trans on SW Eng.*

[10] K. A. Weiss, N. Leveson, K. Lundqvist, N. Farid, and M. Stringfellow, "An Analysis of Causation in Aerospace Accidents," *Space, 2001*, Aug., 2001.