Using Defect Reports to Build Requirements Knowledge in Product Lines

Robyn Lutz
Jet Propulsion Lab/California Institute of Technology & Iowa State University
rlutz@cs.iastate.edu

Nicolas Rouquette
Jet Propulsion Lab/California Institute of Technology
nicolas.rouquette@jpl.nasa.gov

Abstract

In a recent study of a product line, we found that the defect reports both (1) captured new requirements information and (2) implicated undocumented, tacit requirements information in the occurrence of the defects. We report four types of requirements knowledge revealed by software defect reports from integration and system testing for two products in this high-dependability product line. We argue that store-and-retrieve-based requirements management is insufficient to avoid recurrence of these types of defects on upcoming members of the product line. We then propose the use of two mechanisms not traditionally associated with requirements management, one formal and one informal, to improve communication of these types of requirements knowledge to developers of future products in the product line. We show how the two proposed mechanisms, namely feature models extended with assumption specifications (formal) and structured anecdotes of paradigmatic product-line defects (informal), can together improve propagation of the requirements knowledge exposed by these defects to future products in the product line.

1. Introduction

Product line engineering has achieved a high measure of success based on its achievement of systematic reuse of product-line assets for a family of similar products as indicated, e.g., by the hall of fame for product lines [30]. The cost advantages of adopting a product-line approach have been widely reported [23, 31]. Claims of higher quality are harder to demonstrate but appear to have merit [17, 31]. As the product line matures, the developers' increasing familiarity with the set of product line artifacts and their growing understanding of the product-line domain assist in the construction of high-quality products. With careful management, requirements knowledge in a product line thus promises to be both incremental and cumulative. We are interested in how this requirements knowledge can be used to prevent requirements-related defects in product lines.

The development of a product line is typically divided into two phases, *Domain Engineering* and *Application Engineering*. In the first phase, Domain Engineering, the product-line assets are developed. The most important of these are the software requirements, the shared software architecture, and the reusable components and test suites. The requirements are defined by means of a Commonality and Variability Analysis that identifies the common requirements, called *commonalities*, that are shared by all members of the product line and the variable requirements, called *variabilities*, that distinguish among the members of the product line. Variabilities are optional or alternative features that some but not all products have. In the second phase, Application Engineering, the product line assets, such as a common architecture and shared requirements, are reused to build each new system in that product line.

In the work described here we are concerned with a product line of flight software for spacecraft, previously used on seven missions managed by Jet Propulsion Lab with two more missions in development. An example of a commonality from that domain is that all spacecraft have software fault protection to respond to power loss. An example of a variability is that the choice of reaction wheels used to control spacecraft attitude (i.e., position) differs among the products.

This paper describes an effort to use defect reports from previous product-line members to improve the quality of future product-line members by reducing their requirements-related defects. The motivation is that in order to achieve the promise of higher quality offered by a product line approach, we need to avoid repeating in future products the mistakes made in previous products or similar variations thereof.

However, we show that the requirements knowledge available in defect reports of previous product-line members goes far beyond simply avoiding similar mistakes in the future. Capture of tacit requirements knowledge implicated in such reports supports incremental improvement of the product line as described below. To this end, we hope to add defect reports as another key product-line asset.

We focus on requirements-related defect reports because those have been shown to be the most vexing in terms of being difficult to preclude and having a major impact on dependability [16]. This is primarily because requirements-related defect reports often involve missing or incorrect requirements mandated by new knowledge about the environment or about subtle interactions (often timing-related or resource dependencies) among the subsystems.

This paper reports results from an analysis of defects reported during integration and testing for two earlier members of a spacecraft flight software product line to uncover tacit requirements knowledge for the benefit of an upcoming member spacecraft. Our analysis suggests opportunities for the defect reports to serve as a richer source of information for mining tacit requirements knowledge for future product-line members. Although this data is available, its role in the product line is yet to be defined. Due to the project-based organization, the focus has been on avoiding recurrence of defects on the current project (i.e., product-line member) rather than on using this knowledge to improve future projects (i.e., future product-line members).

To the best of our knowledge, this work represents the first attempt at using defect analysis to incrementally improve the requirements knowledge for a product line. There have been multiple efforts to use defect reports to measure quality (e.g., bugs remaining) [7,21], improve the organization's development process [4,14,16,26], and predict future fault occurrences [2, 15], but these efforts have not been in a product line setting.

What makes the defect analysis of prior systems potentially even more useful in a product line setting is that the defect patterns evidenced in previous systems are more likely to be similar to those in future systems because of the high degree of commonality among the product-line members' requirements, the structured reuse of the product-line artifacts, and the shared domain. By capturing the missing or tacit requirements knowledge in early systems, we seek to reduce requirements-related defects on later systems. We use Kruchten, Lago and van Vliet's definition of tacit knowledge as knowledge that is essential but not documented [12]. Because historically it has been the requirements-related defects (e.g., missing requirements, incorrect requirements, or misunderstood requirements) that have caused the most problems in terms of time-consuming debugging and non-localized fixes, the future product-line projects have encouraged this investigation and helped out with additional domain expertise.

In pursuing this investigation, we face two challenges:

- *How to capture relevant product-line requirements knowledge from the defect reports*. This involves acquiring access to members' defect reports from previous product-line applications (non-trivial, as some of these belonged to other organizations), analyzing them and deciding how to filter out information that was not relevant to future products (e.g., one-time variations). We describe below how we identify relevant defect reports and how we use Orthogonal Defect Classification (ODC) [2] to identify requirements-related patterns in the defect data. We also describe an extension to the feature model to model tacit assumptions.
- *How to pro-actively communicate the new requirements-related information to developers of future product-line members*. Initially, we concentrated on determining how to specify and store the information so that it could be retrieved readily by future projects. However, after reviewing our preliminary results with a domain expert with many years of project experience [24], we realized that making the requirements knowledge available to future product-line developers took us only halfway towards our objective of propagating information onward to other product-line systems.

  It is insufficient to specify the findings so as to enable retrieval of the defects-learned requirements knowledge by future developers (a "pull" mechanism for querying stored information). In addition, it is necessary to provide a "push" mechanism so that the requirements knowledge that could have prevented the defects in previous product-line applications will be remembered in future product-line applications.

  We introduce below the concept of a Product-Line Analysis Defect Paradigm (PLA-

DP) (extending Petroski's design paradigm [22]) to propagate forward requirements knowledge gained through product-line analysis of recurring patterns of requirements-related defects. Essentially, we identify a small set of representative anecdotes where a serious defect involving a failure or near-miss occurred that could have been prevented by additional (but usually unavailable at the time) requirements knowledge, where that knowledge also will be needed by future product-line members.

This anecdotal approach to "remembering old lessons learned" (as the domain expert put it) is compatible with the culture of an organization such as ours that builds high-dependability systems. Moreover, the unfolding of requirements-related contributing causes in such anecdotes is often a gripping reminder to the developers of subtle interactions and dependencies of the software requirements on hardware idiosyncrasies and environmental rare events.

Addressing the first challenge involves a domain-engineering activity to incorporate defect reports from previous products into the set of product-line assets. Addressing the second challenge involves an application-engineering activity to make it easy to reuse the defect-report information for the next product in the product line. We hope that this combination of "pull" mechanisms (storing and querying defect reports as product-line assets) and "push" mechanisms (telling stories via PLA-DPs) will help improve the management of requirements knowledge derived from defect reports in product lines.

Analysis of the defect reports generated during integration and system testing for the earlier product-line members showed four types of new requirements knowledge that will be needed by future members of the product line. Each is described in more detail in Section 4.

1. *Newly discovered requirements.* These defect reports from testing described missing or incomplete requirements that will also be essential for some future systems in the product line.
2. *Unexpected requirements dependencies.* A closely related knowledge type was the uncovering during testing of unexpected dependencies among existing variability requirements. This information about requirements dependencies should be provided for future systems in the product line to avoid recurrence.
3. *Tacit requirements rationales.* Undocumented rationales to justify requirements' decisions contributed to problems during testing. Developers of future systems in the product line need this understanding, especially to avoid unintended impacts of changing requirements.
4. *Misunderstood requirements.* These defect reports described requirements-related information that had confused the developer or the tester, e.g., because it was ambiguous. Requirements-related information that confuses developers on one product in the product line, if not clarified, can confuse developers on subsequent products in the product line.

It is these four types of requirements information, revealed by the defect reports of application-engineered instances of the product line, that we seek to incorporate into the domain-engineered product-line assets. Enabling improved reuse of the new requirements knowledge gained during integration and system testing across the product line is the goal of the investigation.

The remainder of the paper is organized as follows. Section 2 briefly describes the spacecraft application. Section 3 describes the approach used to analyze the requirements-related testing defect reports. Section 4 presents two mechanisms to improve communication of these types of requirements knowledge to developers of future products in the product line. Section 5 puts the results in the context of recent, related work. Section 6 provides concluding remarks.

2. Application

We briefly describe three of the spacecraft using the flight software product line to provide needed background. The GRAIL (Gravity Recovery & Interior Laboratory) project will launch twin spacecraft in 2011 to perform gravity mapping of the moon (Figure 1). GRAIL has approximately 3,000 low-level software requirements textually documented in the DOORS requirements management toolset. Lockheed Martin will supply the flight software. The flight software on GRAIL is the most recent instance of a successful Lockheed Martin (LM) software product line. The product-line assets include the core flight software modules.

GRAIL plans significant reuse of the MRO (Mars Reconnaissance Orbiter) flight software, a previous instance of the LM product line. MRO launched in 2005 and is currently orbiting Mars on a five-year mission. We analyzed software defect reports from MRO and from an earlier spacecraft in the flight software product line, Mars Odyssey (ODY). ODY launched in 2001 and is currently operational on an extended science mission. It is also well known for its role in conveying transmissions from the two Martian rovers to Earth.

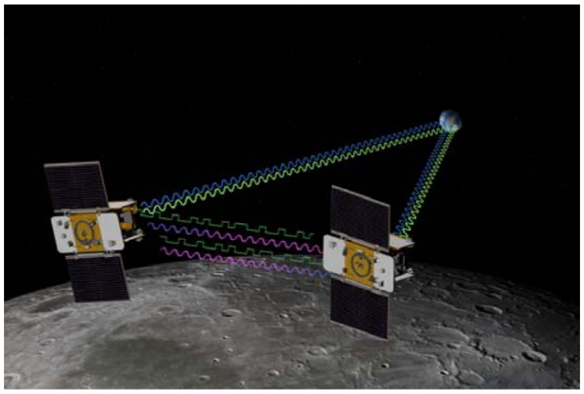
Figure 1. GRAIL (courtesy JPL)

3. Analysis

Our analysis approach is based on an on-going effort to learn from previous product-line anomalies how to improve each new product-line application (such as GRAIL). The intent is to feed-forward insights and concerns from systematic study of the MRO and ODY defect reports to the GRAIL project. The underlying intuition is that the bug-report database for previous members of the flight-software product line captures experience with the product-line code that we can build on to ensure that with software product-line reuse comes enhanced software quality and a reduction in defects during integration and system testing of subsequent product-line members.

Figure 2 shows an overview of this process. On the left we see a sequence of spacecraft in the product line going from top to bottom down the page. The curved arrows show that each inherits the shared (common) software base. In the middle, the horizontal arrows show how each of these projects produces a set of Problem/Failure Reports (PFRs) that is recorded in the problem-reporting database. The dotted line focuses in on MRO and GRAIL. The curved arrow on the right-hand side of the figure represents our goal: to build requirements knowledge from defect reports for previous product-line members (MRO and ODY) in order to reduce defects on the new product-line member (GRAIL).

The dataset for the analysis consisted of the 69 MRO and 24 ODY PFRs classified as flight software-related in the JPL problem-reporting database. The on-line 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.

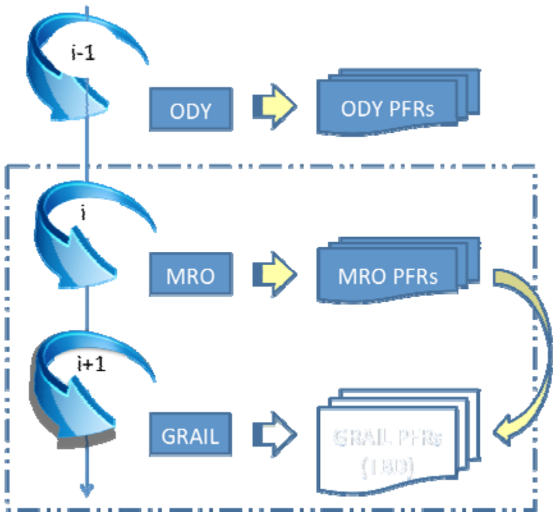

Figure 2. Product-line approach to building requirements knowledge from defect reports

We analyzed the MRO and ODY PFRs using a variation of ODC (Orthogonal Defect Classification) [2]. ODC is a technique that we have previously used to analyze both PFRs and post-launch anomalies on spacecraft [18]. It provides a way to "extract signatures from defects" [2] and to correlate the defects to attributes of the development process. ODC differs from causal analysis, another widely used defect-analysis technique, which instead does a manual, in-depth search for root cause, usually of a subset of defects.

We wrote a script to extract the relevant fields from the PFR database and to format the information for ODC analysis. The approach used four attributes to characterize each PFR: Cause, Target, Problem Type, and Subsystem. The Cause was extracted from the PFR cause field, e.g., “Software design” or “Support equipment (software)”. The Target described the entity that was fixed or changed to avoid the problem in the future. It was extracted from the PFR disposition field. The Problem Type characterized the fix. It was manually classified from the textual description of the problem/failure in the PFR form, e.g., “algorithm” or “timing”. We also extracted and recorded the MRO subsystem for which the fix/change was made, as well as some other information to help with follow-on analyses.

The tall bars at the back of Figure 3 show Algorithms as the most frequent PFR problem for flight software (totaling 34), followed by Configuration PFRs (11), Hardware Design PFRs (8), and Timing PFRs (7) in bars near the foreground of the figure. Requirements-related PFRs are common, but Requirements was rarely selected as the primary cause once code existed. Thus, although no PFRs on MRO and only 6 PFRs on ODY had Requirements listed as their Cause, we will see that many of the algorithm and timing problems did, in fact, have tacit requirements were implicated in the problems.

To build requirements knowledge across the product line, the requirements-related information learned during Application Engineering of individual systems, i.e., from the analysis of the PFRs from testing MRO and ODY, needs to be incorporated into the Domain Engineered products. To this end, we looked at all the software PFRs, even for features that would not be on GRAIL, since subsequent spacecraft in the product line might require those features. This is especially true because, as a Discovery-class mission, GRAIL has stricter budgetary limits, such as limited redundancy and fault protection, than other spacecraft that preceded and will follow it.

The investigation showed that requirements-related defects do recur in the product line. For example, ODY had problems related to managing the mode of operation for the spacecraft's attitude control system and its devices, including reaction wheels. In MRO, the problem of managing the mode of operation for the spacecraft's attitude control system became more difficult, e.g., due to additional redundancy in the power system for the reaction wheels. GRAIL will have common features with ODY and MRO (including attitude control via reaction wheels), but variation in the amount of redundancy and in the configuration of devices for attitude control. There is thus a need to assure that GRAIL will not have a repeat of problems regarding the management of mode of operation for the spacecraft's attitude control system.

The analysis found four types of requirements-related knowledge in the MRO and ODY software-related PFRs:

1. *Newly discovered requirements.* These defect reports describe missing or incomplete requirements where the knowledge needed to identify the requirement only surfaced during testing. Consistent with our earlier studies on some spacecraft not in a product line [16,18], these new requirements often involved complicated interface issues between software variabilities or between hardware and software variabilities. Several of the incomplete requirements involved fault protection, which is of special concern in high-dependability systems such as these. Many of these newly discovered requirements will also be needed in some future systems in the product line.

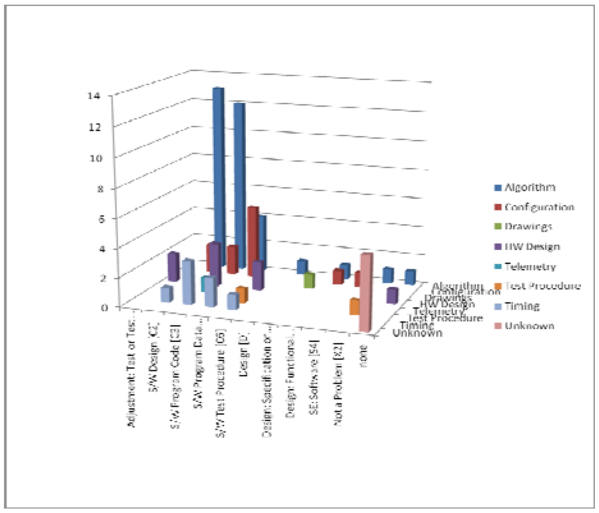

Figure 3 ODC analysis of MRO defect reports

2. *Unexpected requirements dependencies.* A closely related knowledge type was the uncovering during testing of unexpected dependencies among existing variability requirements. These usually involved new knowledge about coordination constraints and were often revealed as inconsistent states. In some cases the unexpected, latent requirements dependency involved an incorrect assumption. This new or corrected knowledge about requirements dependencies will be needed by future systems in the product line to avoid recurrence. A potential concern is whether some dependencies were resolved by one-time workarounds that may not carry over to the next product.

3. *Tacit requirements rationales.* These undocumented rationales, i.e., "justifications of decisions" [5], contributed to defect occurrences, either because the rationale was not sufficiently documented or because it was incorrect. This information was not available to the tester regarding why a requirement had to be the way it was. These rationales often involved the

hardware or environment. Sometimes the new knowledge involved an idiosyncrasy that was known but not explicitly documented. This kind of requirements knowledge is especially useful to future developers during evolution of the product line, as it captures potential, unintended impacts of changing requirements. For example, a report this year from a NASA investigation of risks associated with the growth in complexity of flight software issued sixteen recommendations, with the second one entitled "emphasize requirements rationale" [6].

4. *Misunderstood requirements.* Some defect reports were caused by requirements or requirements-related information that was in some sense documented, but in such a manner that it confused the developer or the tester. Often this occurred because the documentation was partial or ambiguous. Such gaps in requirements understanding often surface when the software behavior is accurate but surprises the testers, leading them to initiate a defect report. In previous work we found that in some cases where the software behaved correctly but unexpectedly, similar, subsequent requirements confusion by others (e.g., operators of the deployed system) could also occur [18]. It seems likely that, in a product line, a similar phenomenon will occur. Requirements-related information that confuses developers on one product in the product line, if not clarified, can confuse developers on subsequent products in the product line. This suggests that in a product line, improving the communication of requirements knowledge can help preclude the recurrence of the same confusion and avoid making the same mistake again on later products.

4. Results

The introductory section of the paper described two challenges to using the requirements knowledge gained from defect analysis of individual product line systems to improve the requirements of subsequent product-line members:

- How to capture relevant product-line requirements knowledge from the defect reports.
- How to pro-actively communicate the new requirements-related information to developers of future product-line members.

The first challenge has to do with *preserving* the new requirements-related knowledge so that it can be reused across the product line as part of the domain-engineered product-line assets. The second challenge has to do with *conveying* the new requirements knowledge to developers of a future project for use in its application engineering.

This section describes how we tried to address the two challenges described in the introduction through the use of two mechanisms not normally associated with requirements management, one formal and one informal, to improve communication of these types of requirements knowledge to developers of future products in the product line. (1) To formally preserve new requirements-related knowledge we *extend feature models with assumption specifications*. (2) To informally convey the new requirements knowledge we use *structured anecdotes of paradigmatic defects*. Together, these two mechanisms appear to help build and propagate the four types of requirements knowledge exposed by the PFRs for use in future products in the product line.

4.1 Extending the feature model to preserve new product-line requirements knowledge

Associating new requirements knowledge with the features in a product line provides a natural way to preserve this information for future product-line applications. A feature model describes the common and variable requirements of a product line by showing the structural relationships (aggregation and generalization) and dependencies (e.g., required, excluded) among the features [3,11]. To incorporate new requirements knowledge, we build on Lago and van Vliet's approach to modeling tacit assumptions in an architecture-based product-line feature model [13].

To document an assumption, they first identify the features directly influenced by the assumption and then define the dependencies between the assumption and the feature model. Their feature model specifies which features are potentially impacted by an assumption as well as on which assumptions a feature depends. Architectural modules and interfaces implement each feature.

Example: Extended Feature Model
The transponder is the spacecraft receiver/transmitter used for telecommunications. During system testing on MRO, a false assumption regarding the transponder was discovered, resulting in new requirements knowledge. The tacit assumption previously had been that the transponder state always reflected the state of the carrier, i.e., locked or unlocked. However, it was found in system testing that these values could be temporarily out of

synchronization when the carrier was transitioning between lock and unlock. The consequence was that the flight software requirement for fault-protection checking of the transponder telemetry had to be revised. New, timing-related software requirements knowledge arising from the asynchronous carrier lock had to be captured. Moreover, since the transponder was a product-line asset, the new knowledge needed to be preserved for other product-line members. Fig. 4 shows a simplified diagram of how the corrected assumption is preserved by incorporating it into a product-line feature model.

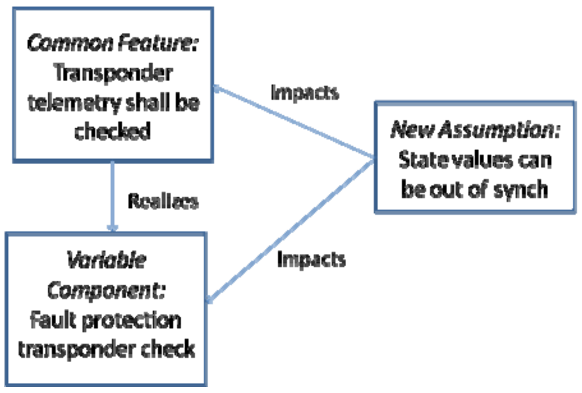
Common Feature: Transponder telemetry shall be checked
Impacts
New Assumption: State values can be out of synch
Realizes
Impacts
Variable Component: Fault protection transponder check

Figure 4 Feature model extended with assumption

To date we have avoided scalability concerns by only proposing to record those assumptions implicated in defect reports in the feature model. This decision is consistent with the scope of our current effort to use knowledge about defects in past systems to improve future systems in the product line. Because defect reports are systematically tracked to closure, requiring maintenance of these defect-related assumption links in the feature model as the product line evolves appears to be practical. However, it is an open question and beyond the scope of this effort whether it would be feasible, or even desirable, to record in a feature model all the many assumptions for complex systems such as the spacecraft. By instead focusing on the historically troublesome assumptions, the extended feature model remains readable and compelling.

4.2 Constructing PLA-DPs to propagate new product-line requirements knowledge

To capture tacit requirements knowledge from past defect reports for use in future products in the product line, we use Product-Line Analysis Defect Paradigm (PLA-DPs). PLA-DPs are stories of previous failures or near-misses on previous products in the product line. They are concrete instances of a pattern of defects found via the product-line analysis of the defect reports. These make the tacit requirements knowledge manifest and more actively involve the developer in scrutinizing previous product-line experience for requirements-related information relevant to their particular project. The remembrance of anecdotes helps ensure that lessons learned on early instances of the product line do not become lessons lost on later instances.

The PLA-DPs extend Petroski's design paradigms [22] to requirements. A design paradigm is a case study "capable of being presented as a fresh and memorable story" that embodies "a general principle of design error" that can also arise in new situations. It both improves understanding and alerts developers to common pitfalls [22]. For example, Petroski describes the collapse of the Tacoma Narrrows Bridge, subtitling it "a paradigm of the selective use of history" and describes the risk of design myopia that prolonged success can bring.

PLA-DPs thus form a small set of anecdotes that represent snapshots of requirements-related defect patterns where sharing the knowledge with subsequent projects is essential. The intent is to make the information not just available to someone looking for it but to actively propagate it to other product-line projects' teams. The PLA-DP is a good fit with product lines because product-line requirements have a high degree of commonality. (Although PLA-DPs are not restricted to requirements-related bugs, we focus here on those.)

Explicit and sustained attention to past anomalies characterizes spacecraft design projects, most especially when the spacecraft inherit components from each other or form part of a product line. For example, Bayer's paper [1] has described 14 MRO post-launch anomalies in detail, including the story of each problem as it unfolded, often from the perspective of the operator, together with recovery attempts, the causal chain and contributing factors, root causes, safety nets used (e.g., that prevented a near-miss from being a catastrophe), corrective actions, and the lessons learned. Presentations to subsequent projects and to lunchtime seminars help propagate the memory of those anomalies.

One recommendation arising from our study was to explicitly include PLA-DP accounts in the domain-engineered product-line repository. By associating them with the defect reports whose lessons they generalize, they can be automatically extracted and distributed (i.e., pushed) to each new project. Since the intent is to make the PLA-DPs not only accessible but also notorious, each textual account is best accompanied by a graphics-rich slide set that visually presents the story of the failure or near-miss and the insights for future systems in the product line. Dudley Herschbach, a Nobel Laureate in chemistry, has similarly described the use of compelling stories as a way to teach the habit of “actively scrutinizing evidence and puzzling out answers” and has urged emphasis on the “human adventure of intellectual exploration, replete with foibles and failures but ultimately achieving wondrous insights” [9]. The PLA-DPs are intended to convey a similar excitement and insight.

Example: PLA-DP

We next give a simplified version of an example PLA-DP from MRO experience, followed by an explanation of its consequence for subsequent spacecraft in the product line.

During integration testing of MRO, the electrical-power subsystem reported an unexpected software state. Analysis showed that this was due to a race condition that could occur if the reaction-wheel subsystem (responsible for aiming the spacecraft) was powered off and then back on in within five minutes. In this case an uplinked sequence of pre-programmed software commands had turned the reaction wheels off. A minute later the high-level fault protection software had commanded the reaction wheels back on. This caused the power software to be “confused” about what state the wheels were in. The fix entailed a new software requirement to disable a schedule-verification function whenever the switch was being commanded.

This anecdote identifies additional requirements-related information regarding time-related constraints on the allowable interactions between the software sequence and the fault-protection software in this product line. This is important information not only for the software developers on this project but also for the developers on the other spacecraft in this product lines. This anecdote also provides a rationale for why the flight software sometimes disables the schedule-verification function. More generally, this anecdote alerts the developer to identify which commands in this product line can be issued (thus, can compete) by different software systems (here, fault protection and sequences) to the same hardware component. This is especially important as timing defects are probably more expensive to fix than function defects, according to a recent study [26].

5. Related Work

The analysis presented here draws together work in requirements management of dependencies, rationales, and assumptions with work in defect analysis, and applies those to product lines.

Most discussion of knowledge management in software development has described architectural knowledge [12] rather than requirements knowledge. Where requirements have been considered, the description is usually of managing the elicitation of tacit requirements [8, 27] rather than requirements maintenance over the lifetime of a product line.

Savolainen and Sajaniemi described a structured feature model that provided a very detailed specification for each feature, including error behavior [25]. They described the need to capture complex feature interactions and to make feature behavior conditional on the presence and absence of other features. However, unlike our work, they did not consider information from defect reports.

Dutoit and Paech included the use of rationales in use-case and scenario-based modeling [5]. They defined rationale to be the justification of system or process decisions, including description of options Their work surveyed rationale management methods and described a process to elicit, document and maintain rationale using web-based tool support for a two-column table where requirements appear in the left column and rationales in the right column. However, unlike the work described here, they considered a single system rather than a product line and provided no discussion of defects.

As described above, Lago and van Vliet showed how assumptions about, e.g., the execution environment, can be incorporated into a graphical feature model by adding assumption nodes and links to the set of features influenced by the assumption [13]. They found that they had to add new features to the model (i.e., make implicit decisions explicit) in order to characterize the assumptions, that assumptions could cross-cut features, and that the dependencies among features and assumptions could be complicated. They recommended the explicit modeling of assumptions for added understanding and traceability and to explore the effect of changing assumptions.

Jirapanthong and Zisman advocated using traceability relations in product line engineering [10]. They described the automatic generation of traceability relations among feature-based documents. However, they did not include defect reports among the eight types of documents that they considered.

Defect analysis during testing has been used to evaluate the readiness of software for release and to estimate the reliability of the software [2]. Fenton and Ohlsson have described the problems in using such defect analysis results to measure the quality of deployed software [7]. Dalal, Hamada, Matthews, and Patton have used ODC to guide pre-release process improvement [4]. Ostrand and Weyuker compared pre and post-release faults in an investigation of module fault density and fault proneness components [21].

While many authors have described the use of defect mining to improve the quality of a single project or of the development process, there has been little work that uses defect analysis results to improve or manage the requirements knowledge needed for a product line. One exception is Maalej and Happel's suggestion of the value of providing a hierarchical schema for classifying errors in order to enable finding similar situations for which experience already exists [19]. Mohagheghi, Conradi, Killi and Schwarz studied reused components and found that they had lower defect density than non-reused ones but more defects with highest severity [20]. At the architectural level, Trew used root-cause analysis of 900 problem reports to identify and package rules to reduce integration errors in a product line [28].

Defect analysis has shown that misunderstanding of requirements and their underlying rationales frequently cause defects. Lauesen and Vinter, for example, looked at 200 of the 800 defect reports available a few months after a product's release. They found that about half of the defect reports involved requirements defects, with missing requirements being the most frequent cause [14]. Similarly, in an early study of testing defects in the spacecraft domain, we found that the most common causes of critical software defects were misunderstanding the software's interfaces with the system and discrepancies (e.g., omissions or inaccuracies) between documented requirements and actual requirements [16]. Van Lamsweerde and Letier's classification of common information-related obstacles to achieving requirements goals (e.g., Information unavailable, Information not in time, and Wrong belief) [29] were used in [18] to describe requirements defects.

6. Conclusion

As resources allow, we plan to expand our use of defect analysis to study the PFRs on other spacecraft in the flight software product line. This may involve PFRs from additional past spacecraft in the product line, such as Mars Odyssey or Phoenix, as well as of implications for additional future spacecraft, such as Juno (launching to Jupiter in 2010) in the product line. More generally, we plan to continue to investigate how requirements knowledge gathered from defect reports from previous product-line members can be preserved and conveyed most effectively to help reduce the risk of similar anomalies for future product-line members.

Acknowledgments. The authors thank Michel Ingham and Steve Larson, JPL, for useful feedback on preliminary work and Michael Stevens, Lockheed Martin, for additional explanations.

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology and funded by NASA's OSMA Software Assurance Research Program.

References

[1] T.J. Bayer, "Mars Reconnaissance Orbiter In-Flight Anomalies and Lessons Learned: An Update", *IEEE Aerospace Conference*, 2009.

[2] 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.

[3] H. Cho, K. Lee, and K. C. Kang, "Feature Relation and Dependency Management: An Aspect-Oriented Approach", *SPLC 2008*, pp. 3-11.

[4] S. Dalal, M. Hamada, P. Matthews, and G. Patton, "Using Defect Patterns to Uncover Opportunities for Improvement," *Proc. Int'l Conf Applications of Software Measurement*, 1999.

[5] A. Dutoit and B. Paech, "Rationale Management in Software Engineering", in *Handbook of Software Engineering and Knowledge Engineering*, Chang, S.K. (Ed.), World Scientific Publishing. 2000, pp. 787-816.

[6] D. Dvorak, ed., *NASA Study on Flight Software Complexity*,http://oceexternal.nasa.gov/OCE_LIB/pdf/1021608main_FSWC_Final_Report.pdf, 2009.

[7] N. E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans on Software Eng*, vol. 26, no. 8, Aug, 200, pp. 797-814.

[8] P. Grunbacher and R. O. Briggs, "Surfacing Tacit Knowledge in Requirements Negotiation: Experiences using EasyWinWin", *Proc. 34th HICSS*, 2001.

[9] D. Herschbach, "The Impossible Takes a Little Longer", in *Science Literacy for the 21st Century*, Stephanie P. Marshall, Judith A. Scheppler, & Michael J. Palmisano, Eds., Prometheus, 2003.

[10] W. Jirapanthong and A. Zisman, “Supporting Product Line Development through Traceabilility, *APSEC’05.*

[11] K. C. Kang, J. Lee, and P. Donohoe, “Feature-oriented Product Line Engineering”, *IEEE Software*, vol. 9, issue 4, July, 2002, pp. 58-65.

[12] P. Kruchten, P. Lago and H. van Vliet, “Building Up and Reasoning about Architectural Knowledge”, *QoSA* 2006, LNCS 4214, pp. 43-58.

[13] P. Lago and H. van Vliet, “Explicit Assumptions Enrich Architectural Models”, *ICSE’05*, pp. 206-214.

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

[15] M. Leszak, D.E. Perry and D. Stoll, “Classification and Evaluation of Defects in a Project Retrospective,” *The Journal of Systems and Software*, vol. 61, issue 3, 1 April, 2002, pp. 173-187.

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

[17] R. Lutz, “Enabling Verifiable Conformance for Product Lines”, *SPLC* 2008, pp. 35-44.

[18] R. Lutz, I. C. Mikulski, “Requirements Discovery during the Testing of Safety-Critical Software”, *ICSE 2003*, pp. 578-585.

[19] W. Maalej and H.-J. Happel, “A Lightweight Approach for Knowledge Sharing in Distributed Software Teams”, *Proc. 7th Conf on Practical Aspects of Knowledge Management*, LNCS 5345, pp. 14–25, 2008.

[20] P. Mohagheghi, R. Conradi, O.M. Killi, H. Schwarz, “An Empirical Study of Software Reuse vs. Defect-Density and Stability”, *ICSE 2004*, 282-292.

[21] T. J. Ostrand and E. J. Weyuker, “The Distribution of Faults in a Large Industrial Software System,” *Proc Int’l Symp on Software Testing and Analysis*, in *Software Engineering Notes*, July, 2002, pp. 55-64

[22] H. Petroski, *Design Paradigms: Case Histories of Error and Judgment in Engineering*, Cambridge University Press, 1994.

[23] K., Pohl, G. Bockle and F. van der Linden, *Software product line engineering: foundations, principles and techiques*. 2005, Springer, DE.

[24] R. Rasmussen, “Thinking Outside the Box to Reduce Complexity in NASA Flight Software”, App. H in D. Dvorak, ed., *NASA Study on Flight Software Complexity,* http://oceexternal.nasa.gov/OCE_LIB/pdf/1021608main_FSWC_Final_Report.pdf, 2009.

[25] P. Savolainen and J. Sajaniemi, “Improving Knowledge Sharing in Embedded Software Production Line”, *1st Intl Workshop on Managing Requirements Knowledge (MARK'08)*, 2008.

[26] F. Shull, V. Basili, B. Boehm, A. Winsor Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, “What We Have Learned About Fighting Defects”, *Proc. 8th IEEE Symp on Software Metrics*, 2002.

[27] A. Stone and P. Sawyer, “Identifying tacit knowledge-based requirements”, *IEE Proc.-Softw.*, vol. 153 (6), Dec., 2006.

[28] T. Trew, “Enabling the Smooth Integration of Core Assets: Defining and Packaging Architectural Rules for a Family of Embedded Products”, *SPLC 2005,* pp. 137-149.

[29] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-Oriented Requirements Engineering,” *IEEE Trans on Software Eng*, vol. 26, no. 10, Oct. 2000, pp. 978-1005.

[30] Software Engineering Institute, “Product Line Hall of Fame,” http://www.sei.cmu.edu/productlines/plp_hof.html

[31] Weiss, D.M. and Lai, C. T. R., *Software Product Line Engineering: A Family-Based Software Development Process*. Boston: Addison-Wesley, 1999.