West Virginia University
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2010

Identifying Common Patterns and Unusual Dependencies in Faults, Failures and Fixes for Large-scale Safety-critical Software

Margaret L. Hamill
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
Hamill, Margaret L., "Identifying Common Patterns and Unusual Dependencies in Faults, Failures and Fixes for Large-scale Safety-critical Software" (2010). *Graduate Theses, Dissertations, and Problem Reports*. 4601.
https://researchrepository.wvu.edu/etd/4601

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Identifying Common Patterns and Unusual Dependencies in Faults, Failures and Fixes for Large-scale Safety-critical Software

by

Margaret L. Hamill

Disseration submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Katerina Goseva-Popstojanova, Ph.D., Chair
HanyAmmar, Ph.D.
Bojan Cukic, Ph.D.
Robyn Lutz, Ph.D.
Arun A. Ross, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2010

Keywords: faults, failures, fixes, quality assurance

Copyright 2010 Margaret L. Hamill

Abstract

Identifying Common Patterns and Unusual Dependencies in Faults, Failures and Fixes for Large-scale Safety-critical Software

Margaret L. Hamill

As software evolves, becoming a more integral part of complex systems, modern society becomes more reliant on the proper functioning of such systems. However, the field of software quality assurance lacks detailed empirical studies from which best practices can be determined. The fundamental factors that contribute to software quality are faults, failures and fixes, and although some studies have considered specific aspects of each, comprehensive studies have been quite rare. Thus, the fact that we establish the cause-effect relationship between the fault(s) that caused individual failures, as well as the link to the fixes made to prevent the failures from (re)occurring appears to be a unique characteristic of our work. In particular, we analyze fault types, verification activities, severity levels, investigation effort, artifacts fixed, components fixed, and the effort required to implement fixes for a large industrial case study. The analysis includes descriptive statistics, statistical inference through formal hypothesis testing, and data mining. Some of the most interesting empirical results include (1) Contrary to popular belief, later life-cycle faults dominate as causes of failures. Furthermore, over 50% of high priority failures (e.g., post-release failures and safety-critical failures) were caused by coding faults. (2) 15% of failures led to fixes spread across multiple components and the spread was largely affected by the software architecture. (3) The amount of effort spent fixing faults associated with each failure was not uniformly distributed across failures; fixes with a greater spread across components and artifacts, required more effort. Overall, the work indicates that fault prevention and elimination efforts focused on later life cycle faults is essential as coding faults were the dominating cause of safety-critical failures and post-release failures. Further, statistical correlation and/or traditional data mining techniques show potential for assessment and prediction of the locations of fixes and the associated effort. By providing quantitative results and including statistical hypothesis testing, which is not yet a standard practice in software engineering, our work enriches the empirical knowledge needed to improve the state-of-the-art and practice in software quality assurance.

Acknowledgements

First and foremost, I would like to thank Dr. Katerina Goseva-Popstajanova for her support and guidance in helping me pursue this research. In addition, I would like to thank my committee members Dr. Hany Ammar, Dr. Bojan Cukic, Dr. Robyn Lutz, and Dr. Arun Ross for sharing their knowledge and time with me. I would also like to thank the entire faculty and staff from the WVU CSEE Department for their continuous support and encouragement and I would like to thank my office mates who have each contributed to the progression of my research.

The vast majority of this work was funded in part by a grant from the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) managed through the NASA IV&V Facility, Fairmont, West Virginia and then independently by the IV&V facility. I cannot thank the NASA personnel enough for their invaluable constant support: Jill Broadwater, Pete Cerna, Randolph Copeland, Susan Creasy, James Dalton, Bryan Fritch, Nick Guerra, John Hinkle, Lynda Kelsoe, Thomas Macaulay, Debbie Miele, Lisa Montgomery, James Moon, Don Ohi, Chad Pokryzwa, David Pruett, Timothy Plew, Scott Radabaugh, Dan Solomon, and Sarma Susarla. It should be noted that any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NASA personnel.

Finally, I would like to express my sincere gratitude to my parents, siblings and friends. Although, they may not realize it I could not have completed this degree without them as they provided me with constant support on a daily basis. Additionally, I would like to thank my new employer, KeyLogic Systems, for their flexibility as I finished and providing with an opportunity to continue working with NASA.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

From the convenience of finding the local coffee shop to necessity of supporting human life people rely on software daily. The consequences of software that fails to perform as expected vary across instances ranging from a slight inconvenience to the user to loss of human life. As software continues to evolve, becoming a more integral part of complex systems, the likelihood that it can contribute to catastrophic accidents increases. Thus, ensuring the proper functioning of software is of greater concern than ever before.

Quality assurance refers to a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. It includes set of activities designed to evaluate the process by which products are developed or manufactured [39]. Many organization have a separate department devoted solely to quality assurance, but with respect to software, quality assurance is an emerging field and thus it lacks empirical studies from which best practices can be determined.

The fundamental factors contributing to software quality are faults and failures. Based on the fact that various definitions exists, which have been used inconsistently in the past we begin by providing definitions, which were adapted from [39].

- A *fault* is an accidental condition or event, which if encountered, may cause the system or system component to fail to perform as required. Faults are the results of an incorrect step, process, or data definition in a computer program that can be introduced at any phase of the software life cycle.

- A *failure* is the inability of a system or component to perform its required functions within specified performance requirements.

Failures are caused by faults. However, not every fault will lead to a failure since the conditions under which a fault would result in a failure may never be met. In fact, the relationship between faults and failure is very complex and mainly unexplored. Our previous work [29], [30] which investigated the adequacy, accuracy, scalability, and uncertainty of architecture-based software reliability models using two large scale open source case studies, found that certain assumptions pertaining to the relationship between faults and failures (e.g., the assumption that each failure can be traced to a single component) do not appear to hold true and some heuristics used in related work (e.g., associating an individual failure with one unique fault) could not be justified. In fact, we found evidence that a single failure may be caused by multiple concurrent faults spread across components and a single fault may cause different failures depending on circumstances. Thus, to avoid making assumptions or using heuristics we define an additional term, fix.

- A *fix* refers collectively to all changes made to correct the fault(s) that caused an individual failure. Thus, a fix encompasses all faults associated with a single failure.

With respect to these definitions there are a few points to be noted. First, faults can be tied to any software artifact (e.g., requirements, design, source code, etc). Second, we consider both observed and potential failures, that is, failures that occurred during operation as well as failures that were prevented from happening by detecting and fixing faults during development and testing. Lastly, in a few cases, the changes made to implement a fix may be implementing a work-around that prevents a failure from reoccurring, rather than actually addressing the root cause fault(s) of the failure (i.e., less than 2%). Throughout this work we use the term failure to refer to both potential and observed failures, and use the terms fixes and changes interchangeably since we only consider changes made to fix faults and/or prevent failures from (re)occurring (i.e., we do not consider changes made for planned updates or requested enhancements).

Of course, as eluded to earlier not all failures are equal. Clearly, failures that can cause a loss of human life or a loss of mission critical functionality are of greater concern than, for example, failures that result in somewhat degraded system performance or a simple inconvenience. Additionally, operational failures are also of great concern as they can be more costly to fix, may lead to critical consequences (e.g., loss of mission) and/or can have a significant impact on an organization's reputation. Even when some failures do not significantly impact the system operation or contribute the critical consequences, if similar failures occur frequently, they too may need to be prevented.

The idea of implementing well planned quality assurance practices is just now becoming standard practice. It is well known that failures can be prevented by either avoiding the insertion of faults or by detecting and correcting faults during development and testing. However, the best practices and effort required for doing so can differ from system to system and throughout the development life-cycle. For example, as software matures, software reliability growth usually occurs and thus fault and failure characteristics change. Although, fault and failure analysis has shown to be quite beneficial in the past (e.g., an estimated seven million dollars in rework was saved by implementing guidelines and checklists developed based on a empirical study of software faults in [68]) few comprehensive studies have been published from which best practices can be drawn. Perhaps, best stated in [40]:

> "In traditional engineering disciplines, the value of learning from failure is well understood, and one could argue that without this feedback loop, software engineering cannot properly claim to be an engineering discipline at all. Of course, many companies track failures in their own software, but there is little attention paid by the field as a whole to historic failures and what can be learned from them".

As in any scientific field it is necessary to use empirical data to investigate theories, determine if and when observed phenomena occur, and update the theories accordingly. In fact, our earlier work showed that in cases when the assumptions made by architecture based software reliability models hold true the models make reasonable reliability estimates, but when the assumptions do not hold true, reliability estimates tend to be over optimistic [31]. Thus, not considering the validity of assumptions and heuristics can create a false confidence.

We conducted a systematic comprehensive study of faults, failures and fixes for a large safety–critical NASA mission. The mission was implemented through Computer Software Configuration Items (CSCIs), which span a wide range of applications from command and control; power generation, distribution, storage, and management of supporting utilities; and failure detection, isolation, and recovery; to human-computer interfaces and scientific research support. The CSCIs represent large scale software components, and thus are referred throughout the work as components. To the best of our knowledge, this dissertation represents the first effort to complete the link from the faults that cause failure (potential or observed) to the fixes implemented to prevent the failure from (re)occurring. Although, our work encompasses some of the work conducted by others, we analyzed a much larger set of fault, failure and fix features than any other study and maintained the link from faults to failure to fixes throughout the work. By qualitatively and quantitatively analyzing different features of faults, failures and fixes we improve the empirical knowledge by identifying common patterns and unusual dependencies that can be used to guide fault prevention, detection and removal efforts.

This work proceeds by introducing the research approach, goals and specific research questions investigated in Chapter 2. In Chapter 3 a detailed review of related work is presented and our unique contributions are explicitly stated in Chapter 4. The details of the case study as well as the data used are given in Chapter 5. Chapters 6 through 9 contain the detailed assessment analysis. Chapter 6 includes the exploration of individual fault and failure features; chapter 7 covers the analysis of pairwise correlation between pairs of fault and failure features; chapter 8 presents the analysis of the features across releases (i.e., n to n+1) and within (i.e., pre to post) releases; and chapter 9 concludes the assessment part with the analysis of fixes made to correct the faults that caused failures. Finally, in Chapter 10 the possibility of predicting features of fixes based on fault and failure features is discussed. Lastly, we address the threats to the validity of our work in Chapter 11 and then provide some concluding remarks and directions for future research in Chapter 12.

Chapter 2

Research Approach and Goals

Although it is clear that detailed empirical studies of faults and failures can benefit the software engineering community, such studies are rare. Perhaps best stated by [54], the lack of studies is in part due to the facts that:

1. Locating and gaining access to empirical fault and failure data is difficult, especially for large, real-world, complex systems.

2. Collecting and analyzing the necessary data is very time consuming, and therefore can be quite expensive.

3. Finding qualified personnel with the appropriate skills to perform the studies is often difficult.

Additionally, we point out that the process of linking faults to failures is often a challenging, time-consuming task. Furthermore, many organizations are reluctant to make fault and failure data available for research and publication. However, the ultimate goal of quality assurance is to improve overall software quality, thus, it is essential that we as a community overcome these challenges and learn from empirical fault and failure data. Whenever possible theories should be tested on empirical data to determine if and when observations hold true, so that the theories can be updated accordingly. By exploiting what is learned we can improve the software development processes as well as the products they produce.

Clearly, the best way to learn from faults and failures is to study real world instances of each, including the relationships the fixes implemented to correct the faults and prevent failures. As suggested in [46], "bug reports from testing and operations are a rich, under-used source of information about requirements". We believe the applicability of bug reporting databases (and other change tracking systems) is much broader as these systems provide valuable sources of information that hold high potential for conducting empirical fault and failure studies, especially when faults and failures are linked to fixes. Thus, we conduct a case study based on the change tracking system used by a large NASA mission (which is implemented through multiple large scale components) to explore different features of faults, failures and fixes.

The change tracking system provided a rich data set of faults, failures and fixes captured in software change requests (SCRs) and changes notices (CNs). Specifically, non-conformance SCRs, which were filed when non-conformance to a requirement was observed (e.g., through analysis activities, testing, in operation, etc.), represent failures (i.e., observed as well as potential failures) and describe the associated faults. CNs were used to track the changes made to address the non-conformance reported, that is, fix the fault(s) that caused the failure (or those that could potentially lead to a failure). Hence, considering these two types of change tracking documents together allowed us to link failures to the faults that caused them and to the changes made to implement fixes.

Overall, the goal was to characterize common patters and unusual dependencies in the data in order to improve the efficiency and effectiveness of quality assurance practices. Towards this end, we systematically investigated and characterized faults, failures and fixes based on detailed analysis of several features extracted from the change tracking database for multiple artifact releases of the NASA mission. Specifically, we investigated in the following fault, failure, and fix features:

1. Fault Type - the type of fault that caused the failure (e.g., incorrect requirements, coding faults, procedural non-compliance etc.).

2. Verification Activity - the verification activity taking place when the fault was detected or the failure was exposed (e.g., inspections, audits, testing, etc.). The verification activity also includes "on-orbit", which represents the cases where failures actually occurred on-orbit because the verification activities in place did not expose the associated faults pre-release.

3. Severity - the potential or actual impact of the failure on the system (e.g. safety-critical or non-critical).

4. Failed Component - the component (i.e., CSCI) the non-conformance was reported against.

5. Release - the software release of the failed component. It should be noted the components followed an iterative development process.

6. Investigation Effort - the hours spent investigating the observed non-conformance and filing the SCR.

7. Fixed Artifacts - the artifacts affected by the changes made to fix faults and prevent failures from (re)occurring (e.g., requirements documents, design documents, code, etc.).

8. Fixed Components - the components (i.e., CSCIs) affected by the changes made to fix faults and prevent failures from (re)occurring. It should be noted that, in general, the failed component (i.e., the component that exhibited non-conformance) was fixed, but in very few cases it may not have been fixed at all.

9. Fix Effort - the total effort in hours spent making the changes need implement the fix per individual failure.

10. Status - the current status of the SCR, that is, open, closed or no action required.

Analyzing these features allowed us to explore the existence of high priority classes of failures, that is, failures that are of utmost importance to prevent. Specifically we are interested in (1) safety–critical failures (i.e. failures that present a risk to human life [51]), (2) operational failures (i.e., post–release failures, which are typically more expensive to fix and in the case of the NASA mission can cause the loss of the mission), (3) failures whose faults required a relatively high effort to fix and (4) failures caused by dominating fault types. By considering the relationships amongst features of faults, failures and fixes we can help prevent the high priority failures. For example, if we can identify the types of faults that often lead to post–release failures (or safety-crtical failures), we can direct developers to focus on preventing the insertion on these types of faults and/or focus activities on detecting them pre-release.

By conducting and publishing this analysis we are taking a step toward a better understand of the complex relationships between faults, failures and fixes. Further, by exploiting the relationships between features we aim to decrease the time and effort required for future fixes by helping to identify the artifacts that need to be fixed, components that need be fixed. Thus, the study was conducted in two parts: assessment and prediction. First, we carefully assessed all available data to defined the fault, failure and fix features discussed above, and then qualitatively and quantitatively analyzed each. It should be noted that the detailed assessment was conducted by our independent research group, but was continually reviewed by and updated based on comments from NASA personnel involved in the development, and verification and validation activities associated with the mission. Once the assessment was completed, we explored the ability of making predictions to improve the efficiency and effectiveness of quality assurance efforts.

The specific research questions explored in this dissertation are introduced next. Research Questions RQ1 through RQ4 are relative to the assessment phase of our study; the results are presented in Chapters 6, 7, 8, and 9, respectively. The set of research questions RQ5, which relate to the prediction phase of the study, are presented in Chapter 10.

2.1 Assessment

We began our analysis by focusing on each fault and failure feature individually in order to reveal common trends. Specially, we explored the following sets of research questions:

RQ1: How are failures distributed across the values of each feature (i.e., fault type, verification activity, and severity level)?

A. Are some fault types more common than others?

B. Do some verification activities detect more faults than others?

C. Are some severity levels more common than others?

D. Is the effort spent reporting failures uniform per component?

As mentioned earlier, we investigated fault types because identifying the most common types of faults would help developers determine which types of fault they should focus on preventing and/or eliminating. Looking at the distribution of failures across verification activities allowed us to distinguish post–release failures from failures observed during development and testing and also consider which types of activities most commonly revealed faults and/or failures. Studying the severity allowed us to distinguish safety-critical failure from non-critical failures. Thus, the fault type, verification activity and severity features allowed us to focus on the high priority failure classes mentioned earlier. Exploring the effort spent reporting the failures can be useful with respect to seeing how effort is spread across the components.

Once the individual features were analyzed, we looked at the relationship(s) amongst different features to identify trends that could be used to update verification practices in order to improve effectiveness and efficiency. In particular, we explored pairwise relationships to answer the following research questions:

RQ2: Does pairwise correlation exist between fault and failure features?

A. Are certain verification activities more likely to detect certain types of faults? Further, are failures exposed during post–release activities (i.e. operational failures) caused by the same types of faults as failures exposed pre-release?

B. Are certain types of faults more likely to result in safety–critical failures?

C. Which verification activities reveal safety–critical failures?

Clearly, the identification of verification activities that are likely to detect common fault types would be beneficial for planning verification activities and allocating resources. Further, quality assurance practices could also be tailored to focus on activities that are most likely to reveal safety–critical failures or operational failures and/or prevent fault types that may be more heavily associated with such failures.

Based on the fact that as software evolves, software reliability growth typically occurs and hence fault and failure behavior changes we investigated trends across releases as quality assurance practices may need to be reassessed and updated throughout the life-cycle. This is especially interesting for the NASA mission which used an iterative development process, meaning functionality was often added in each release. Thus, the following set of research questions is focused on exploring the distribution of the number of failures and the trends in fault types, activities, and severity levels within individual releasees (i.e., from development and testing to on–orbit) and across multiple releases (i.e., from one release to the next).

RQ3: Do trends differ within releases or from one release to the next?

A. What is the distribution of the number of failures across releases? Further, is there a relationship between the number failures reported across releases, that is, between release n and release $n + 1$?

B. Does the contribution of dominating fault types change as the software matures across releases? Specifically, do dominating faults types change from one release to the next?

C. Are certain releases more likely to exhibit on–orbit failures? Further, is there a relationship between the number of failures reported during development and testing and the number of failures reported on–orbit?

D. Does the severity level of failures change as the software matures both within and across releases? Specifically, are safety–critical failures more likely to occur in earlier or later releases?

Knowing how failures distribute across releases can be helpful to project managers when scheduling software engineers' tasks assignments and time allotments. Additionally, if possible, identifying which parts of the system are likely to be responsible for failures in operation or even for failures in the next release would allow resources to be more efficiently focused.

Once the faults and failure were characterized we explored the changes made to fix the faults and prevent failures from (re)occurring. First, characterized the fixes in terms of the fixed artifacts, the fixed components and the effort required to implement the fixes. Then, we explored how fault and failure features where related features of fixes. Specifically, we investigated the following research questions to complete the assessment phase:

RQ4: What are the common characteristics of fixes? And, how do features of fault and failures relate to features of fixes?

A. Which components are affected most often by fixes? Further, are any groups of components commonly fixed together?

B. Which types of software artifacts were fixed most often?

C. Is there any relation between the type of fault that causes a failure and the types of artifacts that need to be fixed?

D. Is there any relation between the verification activity that revealed the fault(s) and the types of artifacts fixed?

E. In terms of the artifacts fixed, do safety–critical failures differ from non-critical failures?

F. Is the effort spent implementing fixes uniform per SCR? per component? If not, what are the common characteristics of fixes that required the most effort?

2.2 Prediction

Once the fault, failure and fix features were characterized we explored the possibility of predicting features of fixes based on the features describing failures and the associated faults. Specifically, we explored the following research questions:

RQ5: Can traditional machine learning algorithms be used to predict features of fixes based on fault and failure features entered in a non-conformance SCR?

A. Can we predict the types of artifacts that will need to be fixed?

B. Can we predict the components that will need to be fixed?

C. Can we predict the amount of effort required to implement the fix?

D. Can high priority failures classes be characterized by common features?

If possible, predictions with respect to the components and types of artifacts to be fixed would clearly be useful in increasing the efficiency of addressing future SCRs by pointing out what needs to be fixed and where. Additionally, the ability to predict the amount of effort require would be useful for planning and adjusting resources, which would be help for developers better estimate expected effort and allocate resources for the continued sustained engineering of the mission or perhaps for a similar missions. Finally, characterizing high priority failures classes would help identify common patterns and unusual dependencies in the fault and failure features associated with safety-critical failures, on-orbit failures, failures caused by common fault types, and failures that were associated with a high fix effort which could be useful in designing against such failures in the future.

Chapter 3

Literature Review

In this chapter we review the related work. For clarity, the discussion is split into multiple sections. In Section 3.1 we discuss works that characterize faults and failures (e.g., by fault type, life-cycle phase the failure occurred in or the fault was detected in, etc.). In Section 3.2 we discuss works that consider how fault and failure data changes across releases, most of which focus on predicting fault proneness. In section 3.3 we review works that considered architectural prosperities of the software for prediction purposes as our results show that fixes were related to the software architecture. In Section 3.4 we discuss works which use data from change tracking repositories to make predictions aimed at improving the process of fixing faults. Finally, in section 3.5 we discuss works that focused on predicting the effort associated with fixes. Each section provides and explanation of the relevance of the works discussed to our work; our unique contributions are discussed in Chapter 4.

3.1 Classification of faults and failures

A number of studies by Christmansson and Chillarege [19], Duraes and Madeira [22], Leszak et al. [43], Lutz and Mikulski [45], and Yu [68] are related to our work because they focus on characterizing and classifying fault and/or failure data. These studies differ in terms of their main goals, the systems studied, and variables explored. None of these works considered the evolution of software across multiple releases.

The goal of the study conducted by Christmansson and Chillarege in [19], as well as the study by Duraes and Madeira in [22] was to discover representative fault types to be used for fault injection at the source code level. To do so, each study used the Orthogonal Defect

Classification (ODC) to classify the fault types; Christmansson and Chillarege studied 408 defect reports from an IBM operating system and Duraes and Madeira studied 668 faults from twelve open source in [22]. Despite the vast differences in the systems studied in each work the distribution of faults across ODC defect types was very similar. It should be noted that both studies limited the analysis to faults directly related to the code and assumed the specification and requirements were correct. Neither [19] nor [22] explored how faults were discovered or the consequences of the corresponding failures. In [22] the ODC schema was extended to further classify faults based on whether the fault represented something missing, wrong or extraneous.

Leszak et al. analyzed a random sample of defect (i.e., fault and failure) modification requests (MRs) (i.e., 427 MRs total) from a network element of an optical transmission in [43]. The MRs were analyzed based on fault types, fault locations, root causes, triggers, the phase of the life cycle in which the fault was introduced, and the phase in which the fault was detected. The percentage of modification requests to fall into each classification category and the effort associated with each category were studied. The authors found that the same types of implementation faults dominated the entire sample, as well as the post–release sub–sample. Additionally, the majority of faults were detected during system integration. Furthermore, it was shown that addressing the implementation faults (which likely included coding faults and integration faults) consumed 75% of the total effort. However, contrary to the popular belief the data shows that faults injected during requirements phases on average do not require much more effort to fix than others.

In a study conducted by Yu, analysis of the phase when faults were introduced into a switching telecommunication software showed that nearly half of the 600 faults were related to coding faults and the majority of them could have been prevented [68]. A lower-level examination of the coding faults revealed that over 50% of them were related to logic, maintainability and interfaces. Based on additional, detailed root cause analysis the team identified and then implemented several countermeasures by developing specific coding fault prevention and inspection guidelines.

Both Leszak [43] and Yu [68] clearly showed that later life cycle faults must be addressed as they account for a significant percentage of faults and require a significant amount of effort to correct. The usefulness of such analysis can be seen in [68] as the release following the institution of the guidelines was one of the best released by the organization. Specifically, the number of faults delivered to the customer decreased by 35%, lowering the impact of the major actionable root causes by 60%.

The study conducted by Lutz and Mikulski in [45] was focused on safety–critical (i.e. high severity) post–launch anomalies of seven unmanned NASA spacecrafts. An ODC based technique was used to classify 199 safety–critical anomalies and the following attributes were considered: defect (i.e., fault) type, verification activity that was taking place when the anomaly was observed, trigger (i.e., the condition(s) that had to exist for the anomaly to surface), and target (i.e., the entity that was fixed). Descriptive statistics were used to explore the association between defect (i.e., fault) type and trigger, and also between target and trigger. The only formally tested hypotheses were with respect to the distribution of anomalies across categories for each of the ODC attributes studied. As expected, results showed that anomalies were not uniformly distributed across the categories of any attribute. Although correlation between some attributes was suggested by graphs and discussed in the paper, it was not quantified and the statistical significance was not explored.

3.2 Analysis within and across releases

Another group of papers related to our study, although with different goals, explored fault and/or failure characteristics across at least two releases by Fenton and Ohlsson in [26], Andersson and Runeson in [1], Ostrand, Weyuker and Bell in [54], [55], [56], and [9], Biyani and Santhanam in [11], and Pighin and Marzona in [58].

We start the discussion with a well known study conduced by Fenton and Ohlsson, which focused specifically on quantitative analysis of fault and failure data [26] for a large Ericsson telecommunications application. The authors empirically investigated several basic software engineering hypotheses related to: the Pareto distribution of faults, the use of size metrics for fault prediction, and the use of fault data to make predictions both across test phases and from pre– to post–release. The authors claimed the most surprising result found was the strong evidence that most fault prone modules pre–release are among the least fault prone post–release. No evidence was found to support the use of popular complexity metrics to predict fault density.

A literal replication of Fenton and Ohlsson study [26] was recently conducted by Andersson and Runeson in [1]. The same hypotheses were explored on three separate software development projects from a large telecommunications company, two of which were application projects developing consumer artifacts and one was an internal platform project. While the majority of the results were fairly consistent, the result related to the relationship between pre– and post–release faults was not. Thus, unlike Fenton and Ohlsson [26], Andersson and Runeson [1] found that a high incidence of pre–release faults implied a high incidence of post–release faults. Even more, the claim was supported with formal statistical analysis based on Pearson correlation coefficient, which showed moderate positive correlation. It should be noted Anderson and Runeson did not discuss the choice of Pearson correlation coefficient as a measure of correlation or its underlying assumptions.

The set of closely related studies conducted by Ostrand, Weyuker and Bell in [54], [55], [56], and [9] analyzed software faults from multiple large industrial systems. In [54], Ostrand and Weyuker explored ways to identify fault prone files by studying the distribution of faults, the relation between size metrics and fault density, and the persistence of faults both pre– to post–release and from one release to the next for twelve releases of an inventory control system. Similarly to Fenton and Ohlsson[26], the results suggested that the files that contained the most pre–release faults were not the most likely place to find post–release faults. Severity was considered in [54], but only to a limited extent, that is, evidence of the Pareto principle was found when analyzing the system by release and severity. Although the distribution of faults across the life–cycle phases in which faults were detected (which closely relates to verification activity) was studied, the relationship with severity was not explored.

A few additional releases of the system studied originally by Ostrand and Weyuker in [54] were considered in [55] to develop a negative binomial regression model. The model used selected variables from the current release of code (e.g., file size) and previous releases of the file change history (e.g., number of faults in previous releases, whether the file had been changed) to predict which files were likely to contain the most faults in the next release. The authors stated that although they had initially planned to use the severity rating in the model, it was determined that the ratings were highly subjective and inaccurate.

Later works [56], [9] by the members of the same group tested the accuracy and applicability of the model by applying it to even more releases of the original case study (an inventory system totaling 17 releases) and two new systems (i.e., 9 releases of a service provisioning system, and an automated voice response system which followed a continuous release pattern, without scheduled releases). In each case, the model proved to be quite accurate. that is, the top 20% of the files identified in each case contained between 73-83% of the faults. Although Ostrand, Weyuker and Bell (i.e., [55], [56], and [9]) did not specifically explore the relationship between the number of faults in two successive releases, the successful prediction results indicated an existence of some relationship.

Biyani and Santhanam explored the relationship between the number of faults in consecutive releases, as well as the relationship between pre– and post–release fault proneness for four releases of a commercial application in [11]. The results showed that to predict the number of faults in the current release it was sufficient to consider only the immediately previous release. With respect to the relationship between pre– and post–release faults the authors concluded that modules found to be faulty in development were likely to have a high number of faults remaining in the field, which is consistent with the result found Andersson and Runeson in [1]. Clearly, the issue of whether there is a positive or negative correlation between pre– and post–release faults and/or failures is still an open research problem.

Pighin and Marzona investigated fault persistence through software releases using data from 23 releases of a management application and 15 releases of a medical application in [58]. Unlike Biyani and Santhanam's work in [11], which dealt with number of faults, this work used the average fault density at each release to track the fault proneness across releases. The core result showed that files with an above average fault density in the first release tend to have higher than average fault densities in later releases. Specifically, 44% of the faulty files in the first release of the first project and 50% of the faulty files in the first release of the second project have a fault density 33% higher than the average fault density in later releases.

Overall for the the works discussed in this section some specifically explored the fault proneness within individual releases (i.e., pre– and post–release)([11] and [58]), while others specifically focused on fault-proneness across consecutive releases ([1], [5], [26], [54], [55],[56], and [9]. Additionally, it should be noted that these studies were conducted at different levels, that is, [1], [11], and [26] worked at the module level, while [5], [9], [54], [55],[56], and [58] worked at the file level. Further, none of these works considered fault types or activities (on a finer granularity than pre– and post–release) and only [54] considered the severity.

3.3 Analysis considering architectural properties

The papers discussed in this section are not tightly related to our work, but have been included because they incorporated analyses with respect to software architecture and our assessment of the characteristics of fixes showed the spread of fixes across components was related to the architecture. Thus, although the goals were quite different some of observations are quite relevant.

In [61] Shereshvsky et al. discussed the use of architectural level metrics to represent quality attributes based on the idea that architectural-level decisions may have a profound impact on finished software products. The explored metrics were based on coupling and cohesion of information, data and control flows in software systems; however due to the infancy of the work no empirical validation was provided. Abdelmoez et al. [2] derived an analytical formula to estimate change propagation probability at a system level based the flow of control and the flow of data between components. By comparing the estimated change propagation matrix to the empirically derived change propagation probability for a JAVA application it was shown that the formula could be used to accurately compare potential architectures.

The ability to identify software components that are likely to be fault (or failure) prone early on is clearly beneficial to managers in terms of project planning and resource allocation, as accommodations in the budget, schedule, methods used and resources available could be accurately scheduled as early as possible. Thus, relationships between software design metrics and fault and/or failure proneness have been explored by numerous researchers.

One of the first studies focused on design metrics was conducted by Zage and Zage throughout [69], [70], and [71]. The metrics explored included internal design metrics (i.e., stress points within functions) and external design metrics (e.g., control flow-in and -out of functions) extracted from the code. The results showed such metrics could be used to correctly identify functions that were likely to have faults.

Ohlsson and Alberg [53] conducted a design metrics study where the metrics were extracted directly from design documents. Based on the Formal Description Language (FDL) graphs used in the design documentation they were able to build a tool that analyzed the design documents automatically and thus easily extracted metrics before code was available. The FDL graphs represented functions and subroutines within modules. The metrics used were based on direct and indirect measures of the graphs and included derivations of McCabe's cyclomatic complexity. By correlating these metrics with the number of failure reports it was shown that it was indeed possible to build a useful fault-prediction model before the code was available.

Chiamber and Kremerer [18] developed a set of design metrics specific to object oriented development methodologies. The six proposed metrics incorporated the experiences of professional software developers and were constructed on a firm basis of theoretical concepts in measurement and the ontology of objects. The metrics were defined at the class level and were based on coupling and cohesion. Using two case studies, one from a software vendor that used object oriented design to development C++ libraries and the other from a semiconductor manufacturer that used Smalltalk for developing flexible machine control and manufacturing, the metrics were shown to be useful in terms of representing design properties, evaluating design decision and focusing resources most effectively.

Since their introduction the metrics defined in [18] have received a vast amount of attention and exploration. For example, Basili et al. [6] explored the validity of the metrics on eight different student development efforts of single information management system. The authors stated that several of the metrics from [18] appeared to be useful for predicting fault-proneness at the class level during the early phases of the life-cycle. Xu et al. [67] explored the use of Chidamber and Kremerers [18] metrics to predict the number of defects expected in a module for a NASA Public Data Set. The results strongly suggested that although some of the metrics from [18] are reliable defect predictors they do not demonstrate as powerful of an impact as a simple SLOC measure. Xu et al. [67] included a comparison of the related work on the metrics defined in [18] and showed that although in each case the explored metrics were found useful results differed across projects.

More recently, architectural based metrics have been used to specifically predict failure proneness (e.g., in [10], [74], and [75]). Zimmermann and Nagappan studied failure proneness of Windows Server 2003 based on dependencies between parts of the code and the complexity of dependency graphs [74], and [75]. The goal was to help managers choose parts of the system for the purpose of allocation of quality assurance resources. The results showed that models accounting for architectural dependencies did better at predicting post-release failures than models based on the typical complexity metrics. Bird et al. [10] focused on predicting failure proneness post-release based on the dependency structure between components and tasks assignments (i.e., who worked on which component how much). The prediction models were evaluated on Windows operating system Vista and multiple versions of the integrated development environment Eclipse. The results showed that these models performed as well as, if not better than models that did not consider the architectural dependency structure and task assignments.

In [41] and [73] models based on design metrics were compared with those based on code metrics as well as those based on combinations of code and design metrics. Jiang et. al [41] compared design and code metrics for 13 NASA datasets and found that the model that combine all metrics (design and code) did slightly better than models based on code metrics only, which did slightly better than models based on design metrics only. Similarly, Zhoa et al. [73] (which only considered one date set) found that both design and code metrics were correlated with the number of faults, but using the metrics together led to improvement in predictions.

Similarly to works discussed in this section we incorporated metrics representing the architecture into our prediction models. However, unlike these works, we consider architectural properties in addition to features of faults and failures.

3.4 Predictions using change request data

It will always remain true that no matter how successful the community becomes at removing faults some faults will indeed remain, which is why we focused on using data from change requests to help handle similar requests more efficiently and effectively in the future. Thus, in this section we discuss works which use data reported through bug reports, modification requests or change requests for predictions, including works conducted in [16] and [17] by Canfora and Cerulo, in [47] by Malin, and in [62] by Sheriff et al.

In [16] and [17], Canfora and Cerulo used historical change data stored in open source software repositories to help determine which files needed to be changed for new change requests and which developer was best equipped to make the changes. The hypothesis was that data stored in software repositories served as good descriptors of how past change requests have been resolved. The approach was based largely on textual similarity and used the textual descriptions of change requests to index developers and source files. An information retrieval method was applied to retrieve candidate developers and source files based on the textual description for a new change request. The accuracy of predicting which files needed to be changed as well as who was best equipped to make those changes varied greatly for different systems. The Firefox Mozilla [50] the web browse and three desktop applications from the *kde* open-source project [44] were used as case studies. With respect to files to be changed the first file identified to require a change was indeed changed 36%, 78%, 45%, 39% of the time depending on the system. Additionally, the first developer predicted as the best developer to handle the job was actually the developer that handled the change request between 30-50% of the time for *kde* studies system, and only 10-20% for Mozilla Firefox.

In [47] Malin explored the use of text mining and tagging of NASA software change requests to help find “more reports similar to this one” during in-flight anomalies. The authors used hierarchical aerospace ontologies of concepts and nomenclature to identify problem type and/or equipment type tags rather than using traditional keyword searches or data mining techniques, which often fail on natural-language text fields. Due to its infancy, limited results are publicly available.

In [62], Sheriff et al. proposed a methodology for determining the effects of changes made to correct failures and prioritize regression test cases needed to ensure fixes were correct and complete. Data from change requests was complied into a matrix that represented historically-based usage relationships between files. Clusters of files that historically changed together were generated by using singular value decomposition on the matrix. By combining the clusters with test case information the authors were able determine whether additional regression tests were necessary. It was found that additional regression test were necessary 50% of the time, and test predicted to have the highest priority (in terms of re-running tests) found a fault 60% of the time.

In summary, it is clear that data stored on change request repositories can be useful for making predictions. Unlike, [16], [17], [47] and [62] we used change request data to predict the artifacts fixed, the components fixed and the effort associated with implementing the fixes, features that none of these works considered.

3.5 Predicting the effort associated with fixes

In this section we discuss works that use features of faults, failures and/or change requests to predict the effort associated with fixes, including works by Evanco [24], Zeng and Rine [72], Weib et al. [65], Song [63], and Mockus [48]. It should be noted that these works specifically considered the predictions with respect to correction effort. For more typical effort estimation, that is, with respect to project development, schedule an/or budget the reader is referred to [42]. Additionally, with respect to the numerous research efforts focussed on fault and failure proneness prediction, we note that a good survey was recently published by Arisholm et al., and review [5] in this section as it was useful for interpretation of prediction results.

The study conducted by Arisholm et al. in [5] reviewed multiple fault-proneness studies and clearly showed that no single modeling technique constantly performs well across all projects. Further, the authors compared several ways of assessing the performance of the models and found that the best performing models differed based on evaluation criteria. Arisholm et al. proposed a new evaluation criteria, cost effectiveness (CE), which considers the cost of inspecting a file based on fault-proneness predictions. However, similarly to the more traditionally used methods for comparisons between models (e.g., receiver operator curves (ROC)), the CE measure is useful for binary-class problems, but for multi-class problems it would be considerably more difficult to conceptualize and interpret the results.

Evanco analyzed the fault correction effort for 509 faults detected during unit testing and system/acceptance testing for three Ada projects in [24]. Effort was classified in four categories: less than or equal to one hour, greater than one hour but less than or equal to one (8 hour) work day, greater than one day but less than or equal to 3 days, and greater than 3 days. The faults were classified based on the number of software components involved, the complexity of the components, and the testing phase in which the faults occurred. The analysis clearly showed that fault correction often involves examining and fixing more than one component. Using an ordered response model the effort was expressed in terms of a log-linear functional form and fault correction effort were ranked according to expected efforts. Some interesting results included (1) the indication that the greater the spread of the faults across components the more effort was required to correct the faults, and (2) faults were more difficult to isolate during system/acceptance testing than during unit testing.

In [72], Zeng and Rine analyzed fix effort from the NASA IV&V Facility Metrics Data Program KC1 dataset. The severity of the defect, the mode the system was operating in, the type of defect (e.g. configuration, design, not a bug etc.), and the number of source lines of code changed or added were used as input variables to estimate defect fix effort using dissimilarity matrices and self organizing neural networks. Although the experimental results indicated good performance for similar software development projects, poorer performance results were found for project with different development environments.

Weib et al. analyzed 567 issue reports from JBoss [65]. A distance function based on text similarity was defined using the title and description fields to measure similarity between issues. Statistical models were used to prediction correction effort for new issues using a Nearest Neighbor Approach (both with and without thresholds). The most interesting result showed that the accuracy of prediction varied greatly for different issue types (e.g., bugs, feature requests, tasks) and the nearest neighbor approach using thresholds was most accurate for predicting correction effort for ”bugs”.

In [63], Song et al. applied association rule mining to NASA's Software Engineering Laboratory's defect data consisting of more than 200 projects over 15 years. Defect associations with respect defect isolation effort and defect correction effort were explored. The attributes considered were defect type and three flags indicating whether the defect was due to a typographical error, whether code was left out (i.e., an omission error), and/or whether it was result of an incorrect executable statement (i.e., a commission error). Similarly to [24] the effort data was classified into 4 categories: less than or equal to one hour, greater than one hour but less than or equal to one (8 hour) work day, greater than one day but less than or equal to 3 days, and greater than 3 days. The accuracy of the association rule mining method was 93.80% predicting defect isolation effort and 94.69% for predicting defect correction effort, which proved to be significantly better than three other well-known machine learning methods (i.e., PART, C4.5, and Naive Bayes).

The ability to predict the amount of effort that remained to be spent on a project based on modification request data was investigated by Mockus et al in [48]. The goal was to predict the amount and distribution over time of the maintenance effort and repair effort associated with new features. The authors chose to represent effort in terms of the time from when the MR was opened to when it was closed due to the difficulty in obtaining actual hours spent making changes. Although, this study only considers repair effort to a limited extent (i.e., with respect to new features) the authors claimed the results of applying the model confirm a fundamental relationship between the new feature and defect repair changes.

Similarly to our work each of these works, our work specifically focused on predicting effort associated with fixing faults. However, the works in this section did not specifically considered predictions with respect to the components fixed or artifacts fixed.

Chapter 4

Contributions

Although fault and failure studies are becoming more and more common as both research and practitioner communities recognize the benefits, overall the software engineering community lacks detailed published empirical studies of large complex real world systems. The lack of studies is likely due to the facts that locating and gaining access to empirical fault and failure data can be quite difficult (as many organizations are reluctant to make fault and failure data available for research), and collecting and analyzing the data is very time consuming and often requires domain expertise. Even more, associating failures with the fault(s) that caused them, as well as the fixes made to prevent them from re-occurring is not straight forward. In fact, some related studies could not differentiate changes made to fix faults from changes made for enhancements, and those that did relied on assumptions or heuristics that were not completely justified. To best of our knowledge, this work is unique in that we analyzed the complete link from the faults that cause failure (potential or observed) to the fixes implemented to prevent them from (re)occurring.

We present a systematic approach for using information stored in software change requests to qualify and quantify features of faults, failures, and fixes as well as the relationships between them. Below we outline our major contributions in relation to the related works discussed in the previous chapter.

- We analyzed a large sample of over 2,500 software change requests (SCRs) entered for the purpose of fixing faults throughout the life of 21 components from flight software of a large, complex, safety–critical NASA mission containing millions of lines of code in over 8,000 files. In particular, we analyzed the types of faults that caused failures, the verification activities taking place when the failures were reported, the severity of failures, and the investigation effort associated with reporting failures. Additionally, we explored the software artifacts fixed, the components fixed to correct faults and the effort required to implement fixes. We measured the statistical association between each pair of attributes. Since fault type, verification activity, fixed artifacts and fixed components were based on a nominal scale, and severity was based on an ordinal scale, we used the χ^2 test for the distribution and the non–parametrical contingency coefficient C as a measure of the correlation between any two attributes. Based on a stratified random sample, we tested the statistical significance of the results.

- Some of the related works treated only particular attributes or limited the analysis. Specifically, [19] and [22] were mainly focused on the types of faults appropriate for fault injection at source code level only, while we explored fault types through the software life cycle (e.g., requirements, design, coding), as well as the activities taking place when the failures occurred, the severity of failures, the artifacts fixed and component(s) fixed. The work conducted in [43] was the only classification study that explored investigation effort and fix effort, but neither was discussed in relation to the artifacts fixed or components fixed.

- In general, severity of software failures has rarely been considered in published literature. In cases when it was considered, it was only to a limited extend. Thus, in [54], it was shown that the majority of faults that led to high severity failures were contained in a small portion of the files. In [19] and [45] severity levels were used as a way to limit the sample. That is, only faults that had an effect on users were considered in [19], while only post–launch, safety–critical anomalies were studied in [45]. Our analysis, in part was focused on safety–critical post–release failures. However, we considered also the bigger picture, that is, faults and failures that occurred throughout the entire life cycle, with all severity levels. Analyzing another safety–critical software system based on data from a broader portion of the life cycle will certainly enhance the empirical knowledge in area.

- Although some of the related work [68], [43], and [45] analyzed combinations of attributes, they neither quantified the extent of associations between attributes nor formally tested the statistical significance of the associations. The only formal statistical test was given in [45] with a goal to test whether the number of anomalies was uniformly distributed across categories for any attribute.

- We explored the trends of fault type, verification activity, and severity within individual releases (i.e., pre to post-release) and across multiple releases for a subset of eleven components, which had sufficient number of non-conformance SCRs and at least two releases. In addition to studying the trends of fault types, verification activity, and severity we also explored the raw number of failure within and across releases. For the number of failures in releases n and $n+1$ (i.e., across releases) and the number of pre–release failures and post–release failures (i.e., within individual releases) we used the Spearman correlation coefficient to quantify the correlation and test for the statistical significance of the results.

  - Related work in this area [54], [55], [56], [9], [58], [26], [1] looked at the relationships between the number of faults or fault density within individual releases (i.e., early testing vs. late testing and pre–release vs. post–release) and across multiple releases (typically from one release to the next). It follows that these papers considered only a limited view on verification activity , in the context of pre– and post–release. As described in the related work in section 3.1, severity was considered in a limited context only in [54]. Fault types (overall or their distribution within and across releases) were not considered in any of these papers.
  - Here as well, related works lacked formal statistical methods. The only paper that did quantify the results statistically was [1]. In particular, the correlation between pre– and post–release faults and between testing phases was computed using the Pearson correlation coefficient, without explicitly stating whether the assumptions required by the Pearson correlation coefficient were valid on the data.

- We also explored the possibility of applying traditional data mining techniques using fault type, verification activity, severity, investigation effort and architectural metrics to predict characteristics of fixes, that is, the artifacts to be fixed, the components to be fixed and the effort required to implement fixes.

  - The idea of using change request data to help improve future fixes is relatively new. Both [16] and [17] as well as the initial work in [47] focused on text-mining of English language fields describing the requested changes, which varied greatly from one entry to the next. We focus on using multiple features, some of which were selected from pre-defined lists and others that were classified from free text fields (with assistance from the project personnel). In [16] and [17] the ability to predict files that need to be changed to address a new change request was explored; the accuracy of results varied greatly across systems. In [62], change request data was used for prediction purpose, but their goal was very different than ours, that is, they focused on prioritizing regression test cases that should be executed to verify changes. None of these works explicitly considered fault types, verification activities, severity levels, investigation effort, or architectural properties; nor did they predict effort required to implement fixes.

– The ability to predict the effort associated with fixes was explored in [24], [72], [65], and [48], but the features used as input differed. In [24], [48], and [72] features available after the fix was identified were used as input (i.e., the number of components involved in [24] and the delta with respect to the number of lines of code changed in [72] and [48]). In our work, we focus on using only the features available when the change request was submitted. In [65] predictions were based on text similarity of titles and descriptions, thus fault type, verification activity and severity were not considered as input features.

Some of the research questions addressed in this dissertation, to the best of our knowledge, have not been addressed in the past. These include (1) exploring which verification activities are most likely to reveal high priority failures (and/or the faults associated with high priority failures), (2) studying distribution of the fault types, activities, and severity across multiple releases, (3) predicting features of fixes (i.e., artifact fixed, components fixed, and fix effort) based only information available when a fault was detected or failure was reported. In addition, we also addressed research questions that have been addressed in the past (e.g., with respect to the number of failures within and across releases). Thus, throughout our work whenever possible we compare our results with recent related studies in order to explore the external validity of the study.

In addition to the systematic approach we consider the quantification of the associations and formal tests of hypotheses, including tests for the statistical significance of the results, to be an important contribution because using formal statistical tests is not yet a standard practice in software engineering in general, and in software quality assurance in particular. The rigorous analysis presented enriches the knowledge related to faults, failures and fixes and we believe it has both scientific and practical software engineering value.

Chapter 5

Case Study Description

In this chapter we introduce the details of the NASA mission used as a case study, including the data analyzed. The mission studied was implemented through Computer Software Configuration Items (CSCIs), which span a wide range of applications from command and control; power generation, distribution, storage, and management of supporting utilities; and failure detection, isolation, and recovery; to human-computer interfaces and scientific research support. The mission is still active and requires sustained engineering. The CSCIs follow an iterative development process, which means new functionality is often added from one releases to the next. As mentioned earlier, we refer to CSCIs as components.

The analysis is based on a snap shot of data from the change tracking system provided to us by the project when the research began. Throughout the analysis, project personnel provided domain expertise which helped us to select the data sample analyzed, understand the software in the context of the larger system, and clarify the meaning, usage and relationships of data fields and values. Their input was invaluable and we are very grateful.

The change tracking system stores Software Change Requests (SCRs) entered by analysts for one of three reasons (1) non-conformance to a requirement was observed (2) the implementation of pre-planned updates is being requested or (3) additional functionality is requested. SCRs are filed at the component level per component release[1]. Since we were interested in exploring and characterizing the relationships between faults, failures and fixes we focused on SCRs entered when non-conformance to a requirement was observed. By definition non-conformance SCRs represent failures and characterize associated faults. It should be that some failures have been observed (for example during testing or operation), while

[1] It should be noted the iterative development of each component is on its own release schedule.

others were only potential failures that have been prevented from happening by detecting faults through verification activities (such as analysis, inspection, or testing) and then fixing them pre–release. Throughout the text we use the terms 'non-conformance SCRs' and 'failures' interchangeably.

The changes made to correct faults were tracked in the change tracking system through Change Notices(CNs). Similarly to SCRs, CNs are filed at the component level per component release. CNs are consider to be 'child' documents to their 'parent' SCRs. Due to the interdependence of components, each SCR can have multiple children CN documents, thus, the system allows for the tracking of changes to multiple components in relation single instances of non-conformance and also to multiple faults within the same component.

To allow better understanding of the data used in this study, we briefly describe the process followed by the NASA mission for creating and addressing software change requests.

Upon creation of a non-conformance SCR the originator records the component and release number for which the non-conformance was observed. Additionally, the originator records how the need for the change was discovered (e.g., inspection, analysis, regression testing, integration testing, on-orbit, etc.), a textual description of non-conformance, and the effort spent reporting the problem (which includes any analysis conducted post non-conformance observance as well as time spent filling out the SCR and in some cases time spent complying any addition information needed to provide evidence of the non-conformance), as well as some other mostly clerical information.

Each non-conformance SCR is reviewed by a board to determine whether or not the observed non-conformance needs to be addressed. If it is deemed the problem needs to be fixed, the SCR is assigned to an analyst.The analysts also records data in SCR, including as the source of the failure (selected from a pre-defined list), and the severity of the failure. Once a solution is identified by the analyst (or in some cases additional software developers) it must be approved by the board before the fix can be implemented. Upon implementing fixes analysts record the SCR associated with fix being implemented, a textual descriptions of the changes being made, the software artifacts changed and the effort spent implementing the changes. The board must then verifies the fix which lead to the closure of the SCR and all associated CNs.

Based on the availability of the data (i.e., whether data was available for all consecutive releases) and some input from project personal we selected 21 components for the analysis, referred to as components 1 through 21. The 21 components contain millions of lines of code in over 8,000 files. The components are arranged in 3-tier hierarchical structure. Component 20 is the single top level component in the hierarchical architecture and is directly connected to six components, 1, 14, 17, 18, 19, and 21, each of which resides within the second tier of the architecture. Each of these components connects a group of additional components to component 20. The components within these groups span the second and third levels of the architecture. It should be noted that components in one group can only interact with components in another group through the hierarchical structure, that is, through component 20. The number of release differed per components, ranging from one release to seven releases (see Table 9.1). Based on an initial review of the data available in the SCRs we removed SCRs with clearly erroneous data (e.g., missing multiple mandatory fields), SCRs that were withdrawn by the user, SCRs tagged as duplicates and SCRs tagged as operator errors. This process resulted in a sample of just over 2,500 SCRs which were entered throughout the life cycle (e.g., development, testing, and operation) of 21 components over a period of almost 10 years.

Details for each component, including the number of releases, the number of files, source lines of code, the cumulative number of non–conformance SCRs over all releases, the number of non-conformance SCRs per file, and the cumulative number of CNs over all releases are shown in Table 9.1. The distribution of the cumulative number of SCRs for all releases across components is shown in Figure 5.1. The components are grouped by the number of releases each has undergone.

As it can be seen from Table 9.1 and Figure 5.1, the size of components and number of SCRs written against each component differ greatly between components. Component size ranged from about 27,000 LOC to almost 740,000 LOC; the number of non-conformance SCRs ranged from 9 to 861; and the number of CNs ranged from 5 to 431. As expected, the single top-level component (i.e., component 20) was the largest component and had the most SCRs and the most CNs filed against. It should be noted that component 20, the component which has a much larger number of failures than any other component (see Figure 8.4), is significantly larger in terms of number of files than any other component but actually has less non-conformance SCRs per file than component 1, 15, and 16 (see Table 9.1, column 5). Additionally, component 20 is the oldest component, and it is the only component at the top level in the system hierarchy. Therefore, component 20 is a central point of interaction and communication between all components and hence it makes sense that a larger number of failures would be associated with component 20.

Component	**# of releases**	**SLOC**	**# of files**	**# of non -conf SCRs**	**# of non-conf SCRs per file**	**# of CNs**
1	1	48,910	207	350	1.69	418
2	2	60,386	200	22	0.11	44
3	2	78,854	287	8	0.03	43
4	2	46,657	228	15	0.07	30
5	2	71,953	269	13	0.05	52
6	2	92,978	321	21	0.07	54
7	2	34,938	289	73	0.25	121
8	2	43,012	270	103	0.38	233
9	2	21,266	125	19	0.15	116
10	3	83,134	356	27	0.08	62
11	3	103,145	444	40	0.09	72
12	3	55,475	277	12	0.04	44
13	3	57,800	599	104	0.17	158
14	3	27,940	280	75	0.27	149
15	3	38,882	84	81	1.04	130
16	3	47,541	169	129	0.76	231
17	4	147,520	587	202	0.27	202
18	5	174,614	552	239	0.41	252
19	7	164,419	747	183	0.33	313
20	7	737,504	1368	735	0.54	1430
21	7	74,618	415	104	0.25	120
Total	-	2,211,546	8,071	2,558	0.32	3,764

Table 5.1: Details of the 21 components

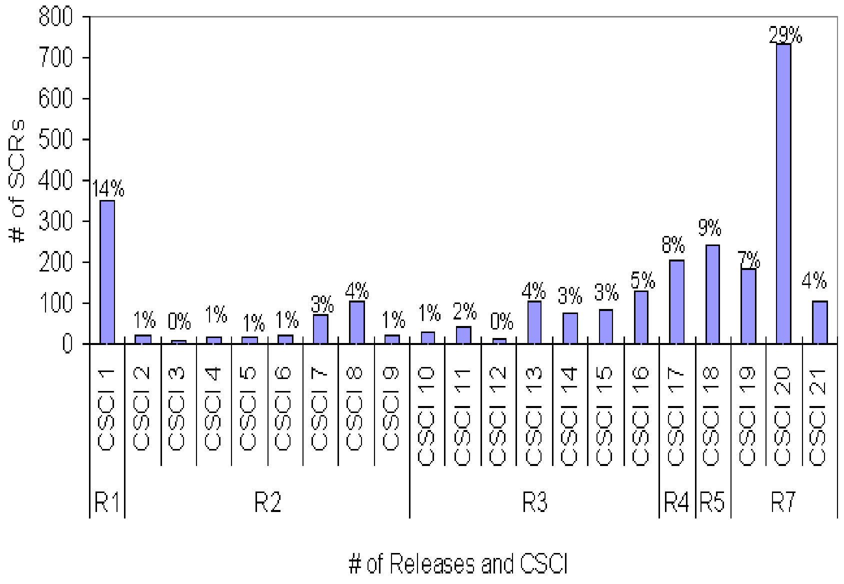

Figure 5.1: Distribution of failures across releases

It should be noted that a review board exists that to verifies the data recorded in SCRs and CNs in terms of consistency and correctness. The board can make changes to recorded data as deemed appropriate, which helps ensure the validity and accuracy of the data we analyze. Additionally, the board must approve any solution before it can be implemented, which helps ensure that the changes made to software artifacts fix the appropriate faults, or at minimum prevent the failure from reoccurring through a work around.

Through detailed exploration of fields in the in change tracking database we selected fields representing features of failures and the associated faults. Specifically, the following fault and failure features were defined based on fields in the non-conformance SCRs.

- Fault Type - the type of fault that caused the failure (e.g., incorrect requirements, coding faults, procedural non-compliance etc.) based on the 'source of failure' field.

- Verification activity - the verification activity taking place when the fault was detected or failure was exposed (e.g., inspections or audits, testing, etc.) based on the 'discovered by field'. It should be noted that the all SCRs that were entered based on non-conformance observed on-orbit are referred to as post-release failures; all other are referred to pre-release failures.

- Severity - the potential or actual impact of the failure on the system (e.g. safety-critical or non-critical) based on the 'severity' field.

- Failed Component - the component (i.e., component) the non-conformance was reported against based on the 'component' field.

- Release - the software release of the failed component which was also based on the release numbers included in the 'component' field .

- Investigation Effort - the time in hours spent post non-conformance observance (which includes any analysis conducted post non-conformance observance as well as time spent filling out the SCR and time spent complying any addition information needed to provide evidence of the non-conformance) based on the 'effort' field.

For each failure we consider all changes implemented cumulatively across CNs, thus we defined the following fix features based on fields in CNs.

- Fixed Artifacts - the types of artifacts affected by the changes made to fix faults and prevent failures from (re)occurring (e.g., requirements documents, design documents, code, etc.) based on the 'affect product' field.

- Fixed Components - the components (i.e., components) affected by the changes made to fix faults and prevent failures from (re)occurring based on the 'component' field.

- Fix Effort - the total effort in hours spent making the changes need implement the fix per individual failure based on the 'effort' field.

Chapters 6 through 9 present the detailed analysis of these features and the relationships amongst them.

Chapter 6

Investigating the Fault and Failure Features

In this chapter, we present the result of our investigation of fault and failure features, that is, we explored the first set of research questions:

RQ1: How are failures distributed across the values of each feature (i.e., fault type, verification activity, and severity level)?

A. Are some fault types more common than others?

B. Do some verification activities detect more faults than others?

C. Are some severity levels more common than others?

D. Is the effort spent reporting failures uniform per component?

The specifics of each feature and category values used throughout the analysis are detailed in the section that follow. This analysis is based on all 2,558 non-conformance SCRs associated with the 21 components we studied.

6.1 Fault Type

We started the analysis by exploring the distribution of different types of faults based on the 'source of failure' field in non-conformance SCRs, to answer research question RQ1–A:

RQ1–A: Are some fault types more common than others?

Types of Faults	**% of SCRs**
Requirements fault	34.71
Design fault	6.14
Coding fault	36.00
Data problem	14.89
Integration fault	2.38
I/O problem	0.63
Compilier/linker/sfw dev or test tool error	0.55
Simulation problem	0.16
Procedural non–compliance	1.02
Process problem	2.07
Fabrication/Manufacturing fault	0.39
None identified	1.06

Table 6.1: Distribution of fault types

The fault type was assigned by project personnel, based on a pre-defined list of values. The category values and the percent of SCRs associated with each value are shown in Table 6.1 and Figure 6.1. The most common sources of failures were *requirements faults* and ***coding faults***, each contributing to about 35% of the failures. It should be noted that requirements faults included incorrect requirements, changed requirements, and missing requirements and coding faults included common coding mistakes such as logical errors, typos etc. The third most common fault type was ***data problems***, which accounted for 15% of the failures. Data problems include faults with respect to components (or some case sub-components) interaction with shared data (e.g., reading and the reporting instrument values). Surprisingly, ***design faults*** were only associated with 6% of the failures. Design faults include faults relative to the design of the system (e.g., omission of component interaction in the design that need for was later realized). The small number of design faults may in part be due to the iterative development process followed by the mission, which allowed them to work with out a detailed design document and led to design decisions that were only implemented in the code. Additionally, 3% of the failures were due to process or procedural issues, 2% were due to integration faults, and 2% were due to simulation and testing problems or other problem which were not directly related to the software artifacts being developed. The fault types was not recorded for only about 1% of the non–conformance SCRs.

The contributions of some of the fault types to the total number of SCRs in the case of NASA mission, differ significantly from the results of some of the older empirical studies.

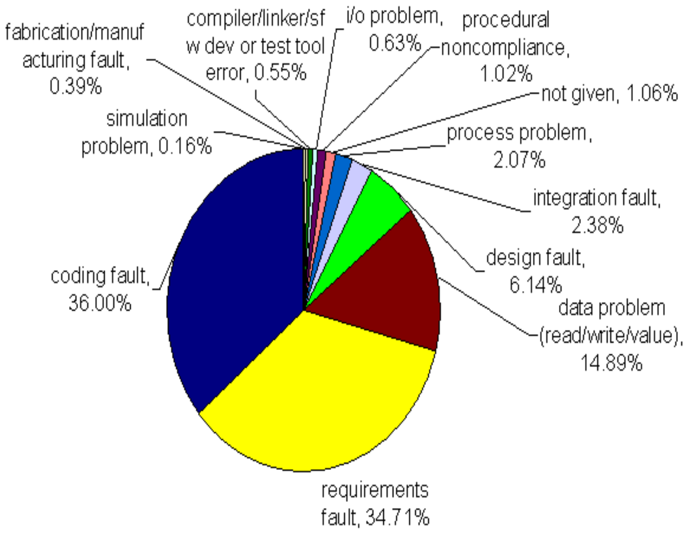

Figure 6.1: Distribution of fault types

For example, the main conclusion of a study done at TRW [13], which was based on 224 faults, was that design faults outweigh coding faults, 64 percent versus 36 percent. Although the percentage of coding faults in our case study is rather consistent (i.e., 36%), the design faults contribute significantly less (i.e., only 6%). Even more, the faults that originated in the early life cycle (i.e., requirements related and design faults) together are less than 41%, which is significantly less than 60 – 70% found in [7], [13], [23].

To explore the external validity, in [37] we compared our results related to fault types to the results of several more recent large scale empirical studies (i.e., [19], [22], [45], [43], [68]). It appeared that the main results were consistent, regardless of the fact that these studies were conducted for different reasons, by different groups, and spanned different domains, implementation languages, development processes and organizations. Specifically, we found that across all studies *the percent of problems reported due to coding, interface, and integration faults together was approximately the same or even higher than the percent of faults due to early life cycle activities (i.e., requirements and design).* The two main implications of this finding are as follows: (1) It contradicts the common belief that the majority of faults are entered during early life cycle activities (e.g., requirements, specification, and design activities), which dates back to some of the older empirical studies [7], [13], [23], [28]. (2) The consistency of results across multiple recent projects (i.e., [19], [22], [45], [43], [68]) suggests that this trend is likely to be an intrinsic characteristic of software faults and failures, rather than a project specific characteristic. Clearly, the software engineering field has grown significantly over the years and the differences are likely a result of new development processes and coding languages, as well as the increased complexity and size of software products.

6.2 Verification Activity

Next, we explore verification activities based on the 'discovered by' field to answer research question:

RQ1–B: Do some verification activities detect more faults than others?

The 'discovered by' field was a free text field that contained short descriptions of the verification activity taking place when then non-conformance to a requirement was identified. Entries included "testing", "analysis", "integration testing" , "unit testing", etc. Each unique textual description was grouped into one of the following categories based on manual

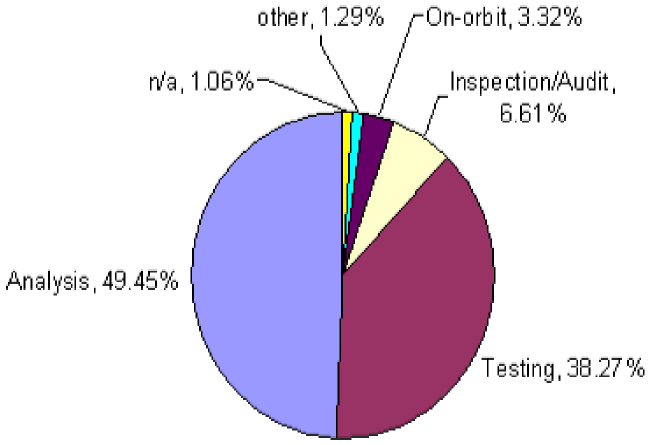

Figure 6.2: Distribution of verification activities

analysis of example SCRs and input from project personnel: inspections and audits, analysis[1], testing, and other (representing rarely seen entries). In some cases, the discovered by field contained the value ”on-orbit”, which represents cases where the verification activities did not successfully reveal the faults during development and testing, thus allowing the fault to lead to an on-orbit failures. Throughout our work we used the term on-orbit and post-release interchangeably since on-orbit failures are the only failures that occurred post-release. All other verification activity values represent non-conformance observed pre-release.

Figure 6.2 shows the percentage of non-conformance SCR discovered through each verification activity. It can be seen that almost 50% of the non-conformance SCRs were discovered during *analysis* and 38% were discovered during various types of *testing*. Only 7% were discovered during *inspections or audits*, and 3% *on–orbit*. The fact that only 3% of failures occurred post launch (i.e., the verification activity was *on–orbit*) shows that the verification procedures and process in place are very effective at discovering and fixing faults during development and testing (i.e., pre–release). Furthermore, the percentage of non–conformance SCRs entered post–release is comparable or even smaller than in other studies; thus, in [54] 3% of the total faults surfaced post–release and for three large scale systems in [1] between 4% and 10% of the faults were discovered post–release.

It should be noted that we do not conclude that any verification activity did better or worse than any other verification activity because the information of the relative effort spent on each verification activity was not available. However, this is a common unknown

[1] Analysis includes activities such as informal reviews or walkthroughs, which differ from more formal inspections that follow specified steps and assign specific roles to individual reviewers [52].

in studies that explore activities, for example [1], [26], [43], and [45] each looked at activities but no discussion was provided with respect to the relative effort put forth to complete the activity.

Additionally, we note the small size of the post-release sample compared to the pre-release sample. However, a small post-release sample is an inherent characteristic of any high-quality software project. Thus despite the large difference in sample size we do continually compare pre- and post-release failures, but remind the reader to keep the sample size in mind.

6.3 Severity

The NASA mission kept track of the severity level for each non-conformance SCRs to capture the impact of each potential or observed failure. This allowed us to explore the following research question:

RQ1–C: Are some severity levels more common than others?

Severity values were assigned by project personnel from a pre-defined list with the following values:

- Sev 1 represents failures which would result in loss of a safety–critical function.
- Sev 1N are those problems which would be Sev 1 but an established reasonable mission procedure precludes any operational scenario in which the problem might occur.
- Sev 2 represents failures which could results in loss of a critical mission support capability.
- Sev 2N are those problems which would be Sev 2 but an established reasonable mission procedure precludes any operational scenario in which the problem might occur.
- Sev 3 are failures perceivable by an operator that is neither Sev 1 or Sev 2.
- Sev 4 represents a discrepancy not perceivable to the flight software user and usually involves an insignificant violation of flight software requirements.
- Sev 5 is condition not perceivable to the flight software user and usually is the case where flight software requirements are not violated, but maybe a programming standard is violated.

For the purpose of our analysis, we group the severity levels used by the project into two categories: safety–critical (Sev 1, Sev 1N, Sev 2, and Sev 2N) and non-critical (Sev 3, Sev 4, and Sev 5). Thus, we found that Safety–critical failures were rare. Overall, across all 2,558 failures, less than 9% were classified as safety–critical; 64%of the total number of failures were classified as non-critical, and about 27% of the total number of failures were unclassified (i.e., no severity has been assigned). Although safety–critical failures are rare fixing them has a high priority as they can lead to unacceptable consequences, which is why we choose to characterize them. It should be noted that the sample size of safety-critical failures is significantly smaller than the sample size of non-critical failures, which is an inherent characteristic of any high-quality safety-critical software project.

6.4 Investigation Effort

Analysts record the time spent from the observance of non-conformance through the competition of the submitted SCR. In some cases, this time period simply includes the time spent filing the SCR, while in others it includes additional time spent gathering evidence of the non-conformance to be presented to the board (e.g., re-run test cases, building data files, etc.). Based on the effort fields we explored the following research question:

RQ1–D: Is the effort spent reporting failures uniform per component?

Of the 2,558 non-conformance SCRs, 99% recorded the investigation effort for a cumulative effort across all components and all SCRs of over 15,000 hours. Figure 6.3 shows the histogram of the investigation effort per failure; effort is shown on a the y axis using logarithmic scale. The median report effort per SCR was 2 hours and values ranged from 0 to 500 hours (less than 0.5% reported 0 hours). By sorting SCRs based on report effort we found evidence of the Pareto principle, that is, 20% of SCRs accounted for 77% of the total effort spent reporting the SCRs, which indicates a very skewed distribution. In fact 80% of SCRs required one hour or less to report.

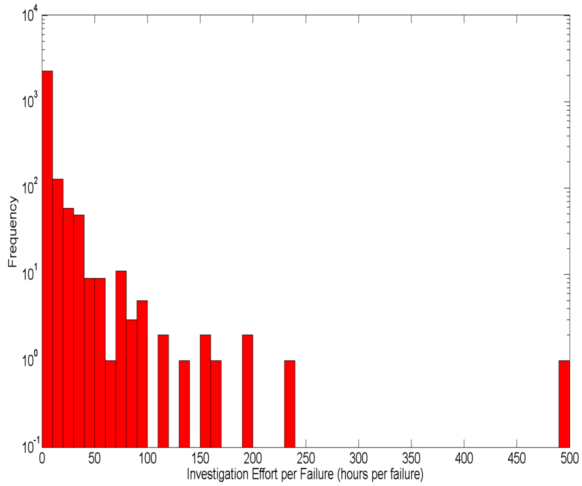

Figure 6.3: Investigation effort histogram

Chapter 7

Pairwise Associations of Fault and Failure Features

In this chapter we study the associations between pairs of the features explored in Chapter 6. The goal was to identify correlations between features that could be used to improve fault prevention and elimination methods and/or the process of resolving failures. Specifically, we explore the second set of research questions:

RQ2: Does pairwise correlation exist between fault and failure features?

A. Are certain verification activities more likely to detect certain types of faults? Further, are failures exposed during post–release activities (i.e. operational failures) caused by the same types of faults as failures exposed pre-release?

B. Are certain types of faults more likely to result in safety–critical failures?

C. Which verification activities reveal safety–critical failures?

Based on the fact that we want to measure the association between the values of two features we used statistical tests including the statistical significance to formalize the observed results. In order to be able to determine the significance of the results of any statistical test, one needs to use a random sample from a given population. For all statistical tests in this chapter we selected a random sample of failures (i.e., SCRs entered due to non–conformance with a requirement). To ensure that the relative contribution of each component to the overall sample of failures is preserved, we conducted stratified random sampling. In particular, we randomly selected half of the failures from each component. In the case where a component had an odd number of failures associated with it, we rounded up. Hence, our random sample consists of $N = 1,436$ failures. For each feature considered in our work, we checked to ensure that the trends seen in the random sample accurately represent those seen in the population. It should be emphasized that the stratified random sample was only used for the statistical tests. The figures and observations presented always represent the entire sample of 2,558 failures.

7.1 Background on Contingency Coefficient

Each feature (i.e., fault type, verification activity, severity) was considered to be a random variable. For any two random variables, X and Y, where n is the number of categories in X and m is the number of categories in Y, we explore whether the distribution of failures across the m categories of Y was the same for each of the n samples (i.e., categories) in X. For example, for RQ2–A we tested whether the distribution of n types of faults is the same across m different activities. In order to test whether or not the n samples come from the same distribution, we built a m x n contingency table using the observed frequencies for each pair of categories, and then calculated the standard χ^2 statistic which can be shown to be approximated by a chi–square distribution with $(m-1)(n-1)$ degrees of freedom. It should be noted that the χ^2 test is applicable to data in a contingency table only if the expected frequencies are sufficiently large. Specifically, it is commonly accepted that for the contingency tables with degrees of freedom grater than one, as suggested by [20], χ^2 statistics may be used if no cell has an expected frequency less than 1 and fewer than 20% of the cells have an expected frequency less than 5. If these requirements are not met by the data in the form in which they were originally collected, categories must be combined in a meaningful way in order to increase the expected frequencies in various cells.

Once the χ^2 statistic has been calculated, we can determine the probability under the null hypothesis (i.e., that the distribution of failures across the m categories in Y is the same for each of the n samples (categories) of X) that a value as large as the calculated χ^2 value is obtained. If the probability is equal to or less than the significance level α (for our purpose $\alpha = 0.05$), the null hypothesis is rejected. The fact that the distribution of failures across the m categories differs significantly for the n samples suggests that there is some correlation between the two variables. Therefore, we also measure the extent of the correlation between the two variables. The null hypothesis in this case, would state that there is no relation between the two variables and any correlation observed in the sample is due to chance.

In this chapter we use the contingency coefficient C as a measure of correlation, since it is uniquely useful in cases when the information about at least one of the features is categorical (i.e., given on a nominal scale). In out case, the fault type and the verification activity features were based on the nominal scale, while the severity feature was based on an ordinal scale. The contingency coefficient C does not require underlying continuity for the various categories used to measure either one or both features. Even more, the contingency coefficient has the same value regardless of how the categories are arranged in the rows and columns. The contingency coefficient C is calculated as

$$C = \sqrt{\frac{\chi^2}{N + \chi^2}} \tag{7.1}$$

where N is the total number of observations [60].

It is important to note that, unlike the other measures of correlation, the maximum value of C is not equal to 1; rather, it depends on the size of the table. Specifically, the maximum value of the contingency coefficient C_{max} is given by:

$$C_{max} = \sqrt[4]{\frac{m-1}{m} \cdot \frac{n-1}{n}}. \tag{7.2}$$

where m is the number of rows and n is the number of columns in the contingency table.

Hence, even small values of C often may be evidence of statistically significant correlation between variables. Further, C values for contingency tables with different sizes are not directly comparable. However, by normalizing C with the corresponding C_{max} as in equation (7.3), we ensure that the range will be between 0 and 1, and hence, C^* values for different sized tables can be compared in terms of the measured correlation with respect to the maximum correlation possible [12]

$$C^* = C/C_{max}. \tag{7.3}$$

Since the test of significance for the contingency coefficient C is based solely on the χ^2 statistics, it follows that if the null hypothesis that the n samples come from the same distribution is rejected, than the calculated C value will be significant. Of course, calculating the values of C and C^* allows us to compare the extent of correlation for different pairs of features.

Throughout following sections we explore the pairwise relationships amongst fault types, detection activities, and severities.

7.2 Fault Type and Verification Activity

Surely, if we can determine which activities are most effective for revealing common faults types, we should be able to improve the effectiveness and efficiency of verification and validation processes. Hence, we started the pairwise analysis by investigating the relationship between the type of fault that caused the failure and the verification activity being performed when the non–conformance SCR was reported (i.e., the failure was observed). Our goal was to explore whether certain types of faults are more likely to be discovered (i.e., associated) with certain types of activities, that is, research question:

> RQ2–A: Are certain verification activities more likely to detect certain types of faults? Further, are failures exposed during post–release activities (i.e. operational failures) caused by the same types of faults as failures exposed pre-release?

For this purpose, we formally test the following null hypothesis:

> $H1_0$***: The distribution of failures across detection activities is the same for all types of faults.***

As described earlier, to ensure the χ^2 statistic would be reasonably accurate we had to combine categories for each feature so that the expected frequencies in combined cells satisfy the requirements as suggested in [20]. Clearly, it is important that these groupings make sense from a software engineering perspective and do not hide important categories. Hence, based on discussions with the project team members, we ended up with the following categories for activities: *analysis*, *inspection/audit*, *testing* (which included all types of testing activities), *on–orbit* and *other* (which included simulation, ‘n/a’, and the original ‘other’ category from Figure 6.2).

With respect to fault type categories for the purpose of using the χ^2 test we have the following categories: *requirement faults*, *design faults*, *coding faults*, *interface & integration faults* (which includes ‘integration faults’ and ‘data problems’ from Figure 6.1), and other (which includes ‘procedural problems’, ‘process problems’, ‘fabrication/manufacturing faults’, ‘complier/linker/software development or testing tool errors’, ‘i/o problem’, ‘simulation problem’, and ‘not given’ from Figure 6.1). It should be noted that all categories associated with the newly formed fault type category *other* are not directly related to the software artifact.

Based on the 5 x 5 contingency table we calculated $\chi^2 = 87.76$, which has a probability of less than 0.001 under that null hypothesis. Hence, we reject $H1_0$ at $\alpha = 0.05$ significance level in favor of the alternative hypothesis that *the distribution of failures across activities is not the same for each fault type.* We then calculated $C = 0.25$, $C_{max} = 0.89$, and $C^* = 0.28$. Although, the correlation is weak, based on the fact that the probability of obtaining $\chi^2 = 87.76$ is less than 0.001 which is less than α, we know that the correlation between the verification activity and fault type attributes is in fact statistically significant. This result, in other words, suggests that *certain activities are more likely to detect failures caused by certain type of faults.* However, the fact that the correlation is weak reminds us that each of the major activities (i.e., *inspections/audits*, *analysis*, *testing*, and *on-orbit*) are necessary as each is responsible for revealing some of the failures caused by each fault type.

A 3-dimensional plot of the frequency counts for the 5 x 5 contingency table for activities and fault types is shown in Figure 7.1. By examining the contingency table with respect to the effectiveness of different activities it can be seen that *analysis* and *testing* activities revealed the majority of faults for each fault type. In general a smaller percentage of total failures are detected by *testing* than by *analysis* (i.e., 38% of failures were detected by *testing* compared to 49% detected by *analysis*). However, when focusing on the failures caused by *coding faults* we see that *testing* activities were more effective than *analysis* in revealing *coding faults* (i.e., 50% of *coding faults* were detected during *testing* while only 39% were detected through *analysis*). With respect to *requirement faults*, the *analysis* verification activity revealed more faults than *testing* (i.e., 58% of *requirement faults* were detected through *analysis* while only 30% were detected during *testing*). It should be noted that the *Inspection/Audit* activities, which revealed close to 7% of the total number of failures, seem to be more likely to reveal *requirement faults* than any other type of faults (i.e., 47% of the total number of failures revealed by *Inspection/Audit* activities were caused by *requirement faults*). Additionally, it can be seen that the majority of *on-orbit* failures were caused by *coding faults* (i.e., 53% of on-orbit failures were cause by coding faults).

Since on-orbit failures are one of the high priority failure classes in terms of prevention and detection, we explored the characteristics of on-orbit failures compared to all other failures by focusing on another meaningful grouping of verification activity categories which differentiates only between failures reported *on–orbit* (i.e., post–release) and failures reported during any *development and testing activities* (i.e., pre-release). Hence, we defined a binary random variable 'on-orbit detection' which was set to one if the failure occurred *on-orbit* and to zero otherwise. Of course, this explores a different question:

What types of faults commonly led to on–orbit failures?

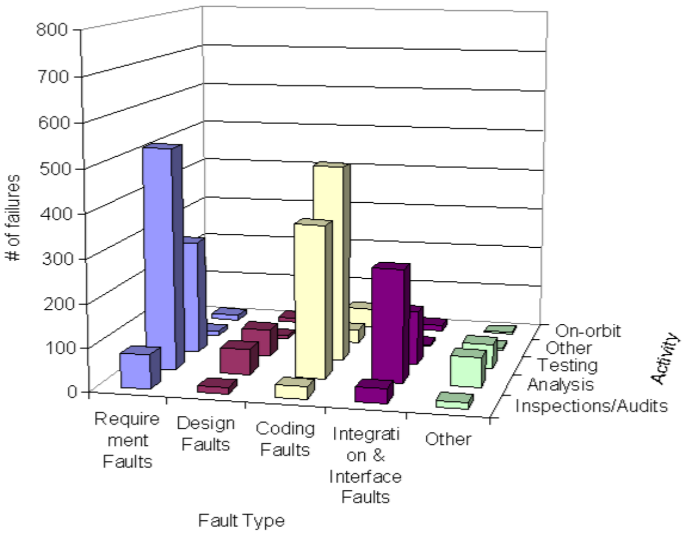

Figure 7.1: Frequency counts across verification activities and fault types

In other words, we are interested to discover which fault types are most likely to escape the pre–release detection, which obviously has a practical value.

Figure 7.2 presents the distribution of fault types for failures reported during *development and testing* and the distribution of fault types for failures reported *on–orbit*. (Note that a logarithmic scale is used to allow the trends in the small number of *on–orbit* failures to be seen.) The same information is given in tabular form in Table 7.1. Based on Figure 7.2 and Table 7.1 we make the following main observations:

- **Coding faults and requirement faults were major causes of failures observed during development and testing, as well as on-orbit.** As it can be seen from Table 7.1 the relative contribution of the *coding faults* to the total number of failures increased from approximately 35% during *development and testing* to almost 53% *on-orbit*. On the other hand, the contributions of the *requirements faults* and *data problems* to the total number of failures decreased from 35% and 15% to approximately 14% and 6%, respectively. *Coding faults*, *requirements faults*, and *data problems* (the three most common fault types overall) together are 'sources' for around 86% of the total number of failures during development and testing, and about 73% of on-orbit failures.

- **The relative contribution of failures due to design faults and integration faults was larger on–orbit than during the development and testing.** In particular, the relative percentage of failures caused by *design faults* increased from around 6% of failures reported during *development and testing* to around 13% for *on-orbit* failures, while the relative percentage of failures caused by *integration faults* increased from around 2% of failures reported during *development and testing* to 11% for *on-orbit* failures. The fact that the relative contribution of failures caused by *integration faults* actually increases *on-orbit* suggests that components may be interacting in unexpected ways and that more integration testing before components are released to fly on on-orbit may be beneficial.

Coding faults and *requirement faults*, which were the two main fault types responsible for all failures, are also the main contributors to post–release (i.e., on–orbit) failures. Similar results were found in [43], that is, the most common types of faults identified from the overall modification requests were the same as those identified from the post–release modification requests. However, when only on–orbit failures are considered, *coding faults* contribute

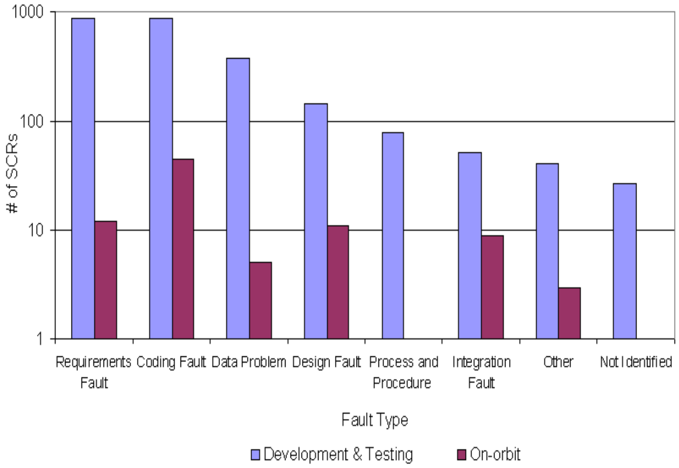

Figure 7.2: Distribution of major fault types pre- and post-release

Fault Type	**% of Development & Testing SCRs**	**% of On-orbit SCRs**
Requirements fault	35.42	14.12
Coding fault	35.42	52.94
Data problem	15.20	5.88
Design fault	5.90	12.94
Process and Procedure	3.19	0.00
Integration fault	2.10	10.59
Other	1.66	3.53
Not Identified	1.09	0.00

Table 7.1: Major fault types pre- and post-release

significantly more than *requirements faults*, followed closely by *design faults* and *integration faults*. It is clear that although common fault types during development and testing tend to be common on–orbit as well, their relative contributions can change. Therefore, we explore whether the likelihood that a failure occurs on–orbit depends on the type of faults that caused the failure. We formulate and test the following null hypothesis:

> $H2_0$***: The distribution of fault types is the same for failures reported pre-release and failures reported post–release (i.e., on-orbit).***

We calculated $\chi^2 = 7.9$, $C = 0.08$, $C_{max} = 0.79$, and $C^* = 0.10$. Since the probability of obtaining $\chi^2 = 7.9$ is less than 0.001 which is less than $\alpha = 0.05$, the null hypothesis $H2_0$ can be rejected in favor of the alternative hypothesis that *the fault types are not distributed the same way across failures reported pre- and post–release.* Furthermore, although the correlation between on-orbit detection status and fault types is statistically significant, the degree of association is less than that of verification activity and fault type (i.e, $C^* = 0.10$ compared to $C^* = 0.28$).

Referring back to Figure 7.1 and focusing on the *on–orbit* verification activity we make an interesting observation: *coding faults* and *design faults* are the most likely types of faults to escape pre-release detection, that is, a larger percent of each of these fault types were revealed *on-orbit* when compared to the percentage of failures caused by other fault types that were revealed *on-orbit.* Considering the fact that *coding faults* are significantly more commonly responsible for *on-orbit* failures than *design faults* (see Table 7.1) suggests that focusing on the prevention and elimination of *coding faults* would be most beneficial in decreasing the number of *on-orbit* failures.

7.3 Fault Type and Severity

Identifying which types of faults are likely to lead to safety–critical failures can be useful when prioritizing fixes. Surely, faults that are likely to lead to safety–critical failures should be addressed immediately. Hence, we explored the following research question:

> RQ2–B: Are certain types of faults more likely to result in safety–critical failures?

The breakdown of the severity level across the major fault types, which is shown in Figure 7.3, indicates that:

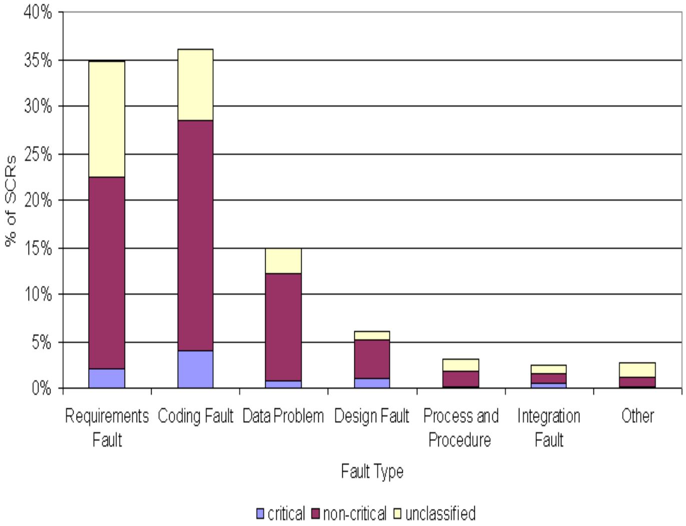

Figure 7.3: Severity levels per major fault type

- **All major fault types can cause safety–critical failures.** 11% of the failures caused by *coding faults* were classified as *safety–critical*. Note that 11% is rather significant percentage considering that *coding faults* were responsible for 36% of the total number of failures. Although 23% of the failures caused by *integration faults* were classified as *safety–critical* their contribution to *safety–critical* failures was significantly smaller since only 2% of the total number of failures were caused by *integration faults* (as shown in Figure 6.1). Similarly, 18% of the failures caused by *design faults* were classified as *safety–critical*, but only 6% of the total number of failures were caused by *design faults* (as shown in Figure 6.1).

Based on this initial descriptive analysis, we formulated and tested the following hypothesis:

> $H3_0$***: The distribution of the severity of failures is the same across all fault types.***

Here, we used *safety–critical*, *non–critical*, and *unclassified* categories for severity and the same categories for fault types as in $H1_0$ (i.e., *requirement faults*, *design faults*, *coding faults*, *interface & integration faults*, and *other*). In this case, $\chi^2 = 67.87$, $C = 0.22$, $C_{max} = 0.85$ and $C^* = 0.26$, which means we should reject $H3_0$ and the correlation between severity and fault types is statistically significant.

The frequency counts of the main fault types in each severity level are shown in the 3-dimensional plot in Figure 7.4. It can be seen that for each fault type (excluding *other*)

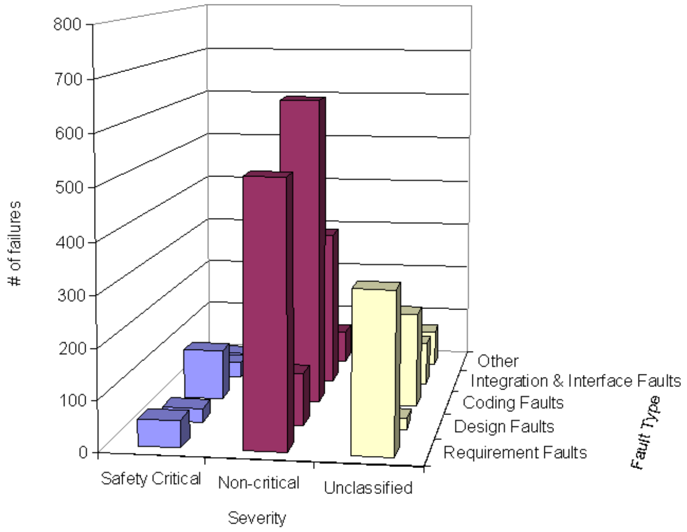

Figure 7.4: Frequency counts across fault types and severity levels

the most common severity class is non–critical, followed by unclassified and then safety–critical. However, when focusing on *safety–critical* failures only, it can be seen that 45% of the total number of *safety–critical* failures were caused by *coding faults*; about 24% of the total number of *safety–critical* failures were caused by *requirement faults*; *Interface & integration faults* were responsible for 15% of the total number of *safety–critical* failures; and *design faults* and were responsible for causing 12% of the total number of *safety–critical* failures.

When integrating these results with the results of research questions RQ1–A and RQ2–A, it appears that *coding faults* were not only responsible for a significant percentage of the total number of failures (i.e., 36%, see Figure 6.1) and more than 50% of the total number of *on-orbit* failures (see Table 7.1), they were responsible for almost 50% of the total *safety–critical* failures (see Figure 7.4) and 52% of the *safety–critical on–orbit* failures.

Although it is commonly believed that the most beneficial way to improve software quality is to focus on so called early-life cycle faults (i.e., faults introduced during requirement and design phases), our results suggest that it is also very important to focus on preventing the introduction and improving the detection and removal of *coding faults* as they are heavily associated with high priority failures (i.e., safety–critical failures and on-orbit failures).

At first sight it appears that our observations related to coding faults contradict the observation made in [45] based on analysis of nearly 200 safety–critical post-launch anomalies from seven unmanned spacecraft. More detailed analysis of the results presented in [45] reveals that the percentage of anomalies caused by coding faults in [45] is comparable to the percentage of anomalies related to requirements and design faults together. As we pointed out in [37], the lower total percentage of requirements, design, and coding faults (32.7%) in [45] is due to the fact that the study analyzed anomalies, with a large percentage of procedure and process faults (29.2%) and large percentage of anomalies for which nothing was fixed (13.6%).

7.4 Verification Activity and Severity

Considering both the verification activity and severity together, we explored the following research question:

RQ2–C: Which verification activities reveal safety–critical failures?

For this purpose we tested the following null hypothesis:

$H4_0$: The distribution of safety–critical and non-critical failures is the same across activities.

To test this hypothesis we used the same categories for verification activity as in case of hypothesis $H1_0$: *analysis*, *inspection/audit*, *testing*, *on–orbit* and *other*and as mentioned earlier, for severity we use: *safety–critical*, *non–critical*, and *unclassified*. Based on the 5 x 3 contingency table, we calculated $\chi^2 = 69.44$, $C = 0.23$, $C_{max} = 0.85$, and $C^* = 0.27$. Since the probability of obtaining $\chi^2 = 69.44$ is less 0.001 which is less than α, we reject the null hypothesis in favor of the alternative hypothesis that *safety–critical failures are more likely to be revealed by certain activities.* Further, we conclude that although $C^* = 0.27$ represents fairly weak correlation, the value is statistically significant; the degree of correlation between verification activity and severity is comparable to the degree of correlation between verification activity and fault type.

Similarly as in the case of types of faults, *safety–critical* failures were related more often with some activities than with others. Figure 7.5 shows a 3-dimensional plot of the frequency counts for verification activity and severity features. Notice that for both *non-critical* failures and *unclassified* failures *analysis* was the most successful verification activity in detecting failures (i.e, 51% of *non-critical* failures and 51% of *unclassified* failures were discovered by *analysis*). However, it can be seen that slightly more *safety–critical* failures were discovered

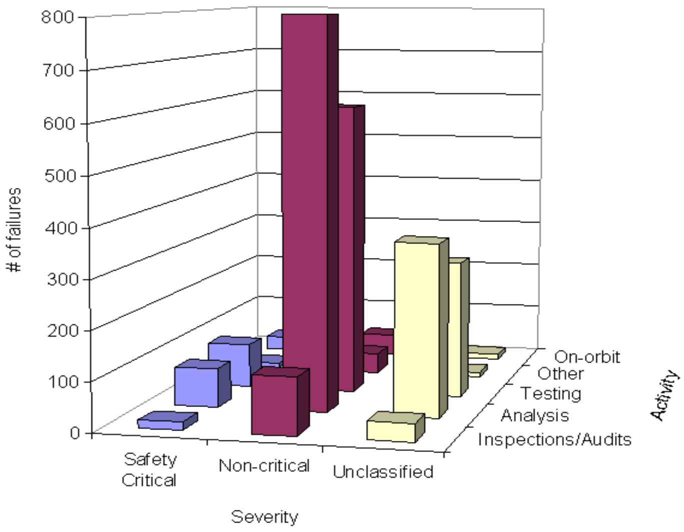

Figure 7.5: Frequency counts across verification activities and severity levels

Severity	**% of Development & Testing SCRs**	**% of On-orbit SCRs**
Safety–critical	7.93	34.12
Non–critical	64.50	52.94
Unclassified	27.58	12.94

Table 7.2: Severity levels pre- and post-release

by *testing* than by *analysis*. Specifically, 40% of the total number of *safety–critical* failures were discovered through some *testing* verification activity, while 35% were discovered during *analysis*. *Inspection/Audit* activities revealed approximately 7% of failures in each severity class. Focusing on the *safety–critical* failures, it can be seen that a fairly significant portion (i.e., 13%) of the total number of *safety–critical* failures occurred *on-orbit*.

Hence, we also explored the relationship between severity and on–orbit detection status. Table 7.2 shows the percentage of *safety–critical*, *non-critical* and *unclassified* failures reported during *development and testing*, and *on–orbit*. We formally tested the following hypothesis:

> $H5_0$***: The distribution of the severity is the same for pre–release and post–release failures.***

By calculating $\chi^2 = 51.49$, $C = 0.20$, $C_{max} = 0.76$, and $C^* = 0.26$ we reject $H5_0$ in favor of the alternative hypothesis that *the distribution of the severity of pre-release failures differs from the distribution of the severity of post–release failures.* Once again, although the correlation is weak, it is statistically significant. Unlike the fault type, severity is comparably correlated for both the on-orbit detection status and the more fine-grained verification activity feature.

Based on Table 7.2 and the statistical results we made the following observations:

- **A larger percentage of on-orbit failures were safety–critical.** As shown in Table 7.2, 34% of the total number of *on–orbit* failures are *safety–critical*, while less than 8% of the total number of failures reported during *development and testing* are *safety–critical*.

- **Greater emphasis was placed on fixing and documenting on-orbit failures.** This is shown by the fact that only 13% of *on-orbit* failures were unclassified compared to 28% unclassified failures during *development and testing*.

Further study of the differences between the *safety–critical* failures detected during *development and testing* and the *safety–critical* failures that occurred *on-orbit* may provide insights on how to pro-actively decrease the number of *safety–critical on-orbit* failures.

Chapter 8

Analysis of trends across releases

In this chapter we present the analysis of trends in features within and across releases for types of faults, activities (i.e., pre-release and post–release), and severity (i.e., critical and non-critical). For this analysis we selected a subset of components with two or more releases that had at least 70 failures reported against them (i.e., at least 70 non–conformance SCRs were filed against the component). We believe that less failures, when broken down per release, would not show realistic trends, or at least the results would be very sensitive to small variations. Thus, the analysis presented in this chapter was based on the data from the following eleven components: components 7 and 8 (each with two releases), components 13, 14, 15, 16 (each with three releases), component 17 (with four releases), component 18 (with five releases), componenta 19, 20, 21 (each with seven releases). This subset contains almost 70% of the total non-conformance SCRs explored in Chapters 6 and 7 and , that is, $2,029$ SCRs were reported against the selected eleven components. As expected this subset shows strong support for the observations made in chapter 6 based on the entire data set. It should be noted that only four components have more than four releases (i.e., R5 to R7) and in some cases the number of failures reported in later releases was very small (e.g., component 20 has only two recorded failures in release seven) which limits the analysis for later releases.

We consider trends in features across releases, first collectively for all eleven components and then individually for each component. Additionally, the per release data allowed us to explore relationships between the number of failures across releases (i.e., from one release to the next) and within individual releases (i.e., from pre- to post–release).

This chapter address the third set of research questions:

RQ3: Do trends differ within releases or from one release to the next?

A. What is the distribution of the number of failures across releases? Further, is there a relationship between the number failures reported across releases, that is, between release n and release $n+1$?

B. Does the contribution of dominating fault types change as the software matures across releases? Specifically, do dominating faults types change from one release to the next?

C. Are certain releases more likely to exhibit on–orbit failures? Further, is there a relationship between the number of failures reported during development and testing and the number of failures reported on–orbit?

D. Does the severity level of failures change as the software matures both within and across releases? Specifically, are safety–critical failures more likely to occur in earlier or later releases?

Based on the fact that some of the features explored to answer the third set of research questions were on the ratio scale we were able to use a stronger measures for correlation. To determine the appropriate measure of correlation, we test for normality using the Shapiro–Wilks W test which is commonly used on samples sizes less than fifty [21]. Pearson coefficient of correlation is the best in terms of power, but it cannot be used if the random variables fail the test for normality. In such case one can use the Spearman's rank correlation coefficient, which does not assume normality. Suppose we have two random variables X and Y whose values are sorted and ranked with corresponding rankings denoted by $X_1, X_2, \ldots, X_N$ and $Y_1, Y_2, \ldots, Y_N$. Correlation between the two variables would be perfect if and only if $X_i = Y_i$ for all i's, which makes it logical to use the differences $D_i = X_i - Y_i$ as an indication of the disparity between the two sets of rankings. Spearman's rank correlation coefficient is calculated by

$$r_s = 1 - \frac{6\sum_{i=1}^{N} D_i^2}{N(N^2-1)} \tag{8.1}$$

Unlike the contingency coefficient C used in the previous chapter, Spearman's rank correlation coefficient is a number between -1 and 1 ($-1 \leq r_s \leq 1$), with -1 and 1 representing perfect negative and positive correlation, respectively. In order to test for the significance of the correlation coefficient the sample must be randomly drawn from the population. In this chapter the population consists of eleven components, thus we randomly sampled six components five different times and conducted the statistical test on each sample. The test of significance of r_s is based on possible permutation of ranks and the probability of obtaining the calculated value of r_s. In other words, the result is significant if the calculated value of r_s is larger than the critical value of Spearman's rank correlation. For sample size $N = 6$ and significance level $\alpha = 0.05$, the critical value is $r_s = 0.83$.

8.1 The distribution of failures across releases

First, we explored the number of failures recorded per release for each of the selected eleven components in order to answer the following research question:

> RQ3–A: What is the distribution of the number of failures across releases? Further, is there a relationship between the number failures reported across releases, that is, between release n and release $n + 1$?

Figures 8.1 through 8.4 present the distribution of failures across releases, where components are grouped by the number of consecutive releases in the data. The distribution of SCRs across releases follows a bell shaped curve for the majority of the components (i.e., components 14, 15, 17, 19, 20, 21), which implies that despite the iterative development process and added functionality between releases, components tend to exhibit fewer failures in later releases and software reliability growth has occurred. The few exceptions are explainable. Component 13 shows a monotonically increasing trend, which we believe believe will follow a bell shaped curve as it matures through releases. Additionally, as shown in Figure 8.1, for components 7 and 8 (which only have two releases and are the youngest components studies) the number of failures decreases from release one to release two, similarly as component 16 (see Figure 8.2). We suspect this behavior may be explained by the functionality added between releases. In other words, release 2 may not have included significant additional functionality, the components may have been updated only to fix faults in release 1.

Next, we explored whether the number of failures in release n provided any indication of the number of failures that occurred in the next release $n + 1$. Based on the fact the every non-conformance SCR was assigned to one of the 21 components, but many could not be mapped to files we explored the relationship in the number of failure per release per component.

For this purpose we formulated the following null hypothesis:

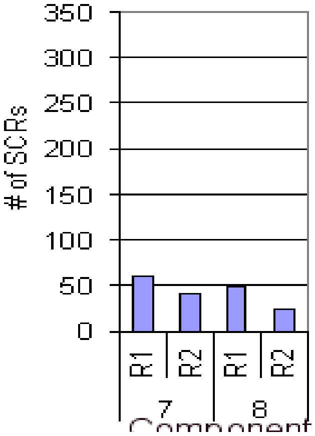

Figure 8.1: Distribution of failures across components with 2 releases

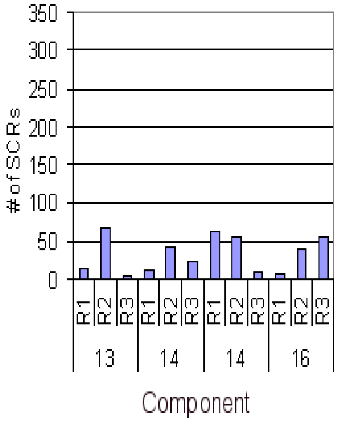

Figure 8.2: Distribution of failures across components with 3 releases

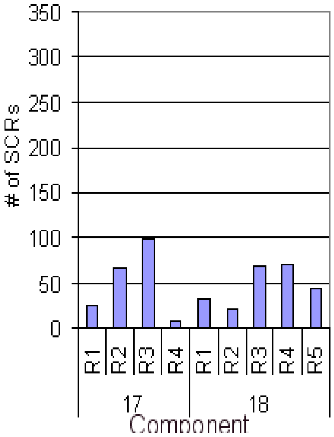

Figure 8.3: Distribution of failures across components with 4 or 5 releases

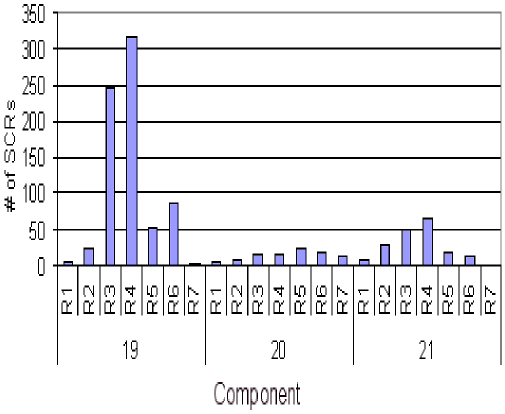

Figure 8.4: Distribution of failures across components with 7 releases

> $H6_0$***: At component level, the number of failures in release*** $n+1$ ***is unrelated to the number of failures that occurred in release*** n***.***

Since later releases are based on fewer components and have significantly less failures reported, we only test $H6_0$ for releases R1 and R2, and then for R2 and R3. We use the Shapiro-Wilk W test for normality, which is recommended for small to medium sized samples [21]. According to the test, the hypothesis of normality cannot be rejected for R1 and R2, but it is clearly rejected for R3. Hence for consistency, we do not use the Pearson correlation coefficient. Instead, we use the Spearman rank correlation coefficient r_s, which does not require normality.

Figure 8.5 shows a scatter plot representing the relationship between the number of failures in release R1 and number of failures in release R2 for all eleven components, with each dot representing a component. The calculated correlation $r_s = 0.42$, which is less than the critical value of 0.52 for $N = 11$ and $\alpha = 0.05$, leads us to conclude that $H6$ cannot be rejected. To test for the statistical significance of the calculate correlation, we sampled five random samples, each with six components. For each of the five random samples, $0.14 \leq r_s \leq 0.71$ which is always less than the critical value of 0.83 for $N = 6$ and $\alpha = 0.05$. Therefore, once again we found we can not reject the null hypothesis $H6_0$. This result basically means that at the component level the number of failures recorded in the second release was unrelated to the number of failures in the first release.

Figure 8.6 shows the scatter plot between the number of failures reported in release R2 and the number of failures reported in release R3 for the nine components that had at least three releases. The calculated value of $r_s = 0.22$ for all nine components, and $-0.60 \leq r_s \leq 0.20$ for five random samples of six components did not allow us to reject the null hypothesis, and thus we conclude that at component level the number of failures in release three was unrelated to the number of failures in release two; that is, any relationship suggested by the graph most likely is due to chance. Further, even when considering the relationship between the number of failures in release three, based on the cumulative number of failures in the two previous releases (one and two) the computed value of r_s is not large enough to reject the null hypothesis for all nine components or any of the five random samples, each with six randomly selected components.

The lack of a statistically significant correlation between the number of failures identified in subsequent releases at the component level may be explained by the iterative development process followed by the mission. Components tend to have unique schedules with respect

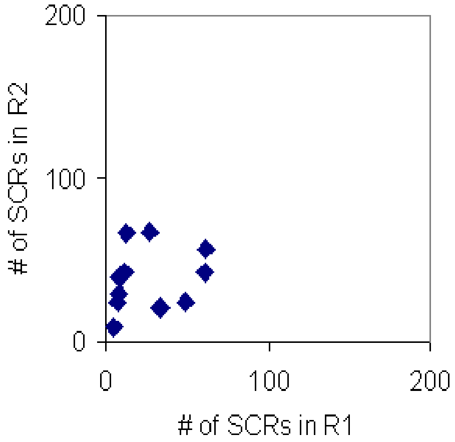

Figure 8.5: The number of failures in release R1 versus the number of failures in release R2 ($r_s = 0.42$)

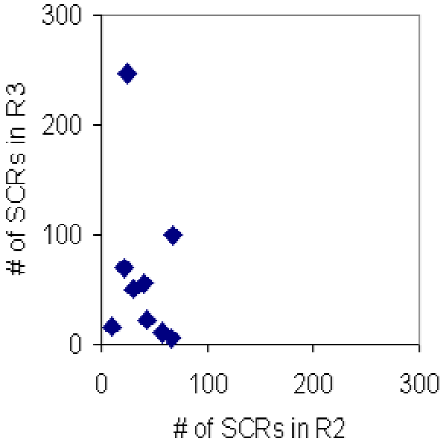

Figure 8.6: The number of failures in release R2 versus the number of failures in release R3 ($r_s = 0.22$)

to added functionality across releases leading to no clear association between the number failures. Further exploration in this direction was not possible due to a lack of detailed data about how and when functionality was added or removed in each release.

These results differ with respect to observations Biyani and Santahanam made in [11], which stated "it is clear that historical defect[1] information is valuable in terms of arriving at any prediction algorithms" and found that to predict defect volume by module for a new release it is sufficient to consider defect history from only the immediately previous release. Additionally, evidence that faulty files in early releases tend to remain faulty in later release was shown in [54] and [58]. Specifically, in [54] the authors identified the 'high fault' files (i.e., the top 20% of files when ordered by decreasing number of faults) for each of the 13 releases and found that between 22% and 63% of files identified as high fault files in release n were also identified as high fault files in release $n + 1$. In [58] the conclusion of fault persistence across releases was based on the fact that files with a higher than average fault density in the first release tend to have higher than average fault densities in later releases.

The conflicting results may be due to the fact that different levels of granularity were used. Specifically, we used the component level, file level was used in [54], and module level was used in [1], [26], [11]). However, we suspect that due of the small number of failures associated with the individual releases of each component similar analysis at the file level would lead to even fewer failures per file and thus would also fail to show a relationship. However, based on the available data we were not able to link the non–conformance SCRs to the files and therefore we could not explore whether the trends observed at the component level would hold true at the file level.

It should be noted that although no relationship was found at the component level, future work focused on exploring similar trends on different system would help in building empirical knowledge and clarifying this research question.

In the following sections we explore the trends related to the distributions of fault types, on–orbit failures, and severity of failures across multiple releases.

[1] Based on the statements given in [11] it appears that in that paper 'defects' refer to both faults and failures.

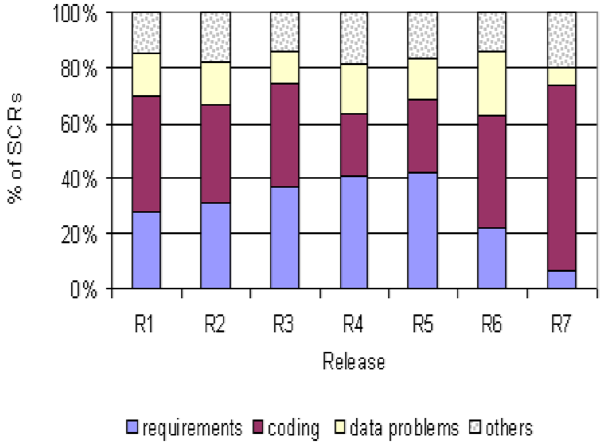

Figure 8.7: Major fault types per release

8.2 Common Fault Types within and across Releases

In this section we present the analysis of the fault types across and within releases. Specifically, we considered the three most common fault types (i.e., *requirements faults*, *coding faults*, and *data problems*) and combined the others in to the *others* category to answer the following research question:

> RQ3–B: Does the contribution of dominating fault types change as the software matures across releases? Specifically, do dominating faults types change from one release to the next?

The distribution of the fault types across releases for all failures associated with the eleven components considered in this chapter is shown in Figure 8.7. Based on the figure it is obvious that the three major fault types persisted across releases. *Requirements faults*, *coding faults*, and *data problems* together contributed to 82%–86% of failures across all releases.

We further explored fault types using the box plots shown in Figures 8.8 through 8.10. We choose to use box plots because they show both the central tendency and dispersion of random variables, which is important in this context having in mind the variability of fault types across releases at component level. In each graph, for each release, the box contains the 25^{th} to 75^{th} percentile, and the horizontal line within the box represents the median. The whiskers (i.e., the vertical lines that reach outside the box) represent the range. The maximum values, show a bell shaped curve similar to the ones shown in Figures 8.1 through 8.4 in chapter 8.1. In the majority of cases in Figures 8.8 - 8.10, the maximum value shown by

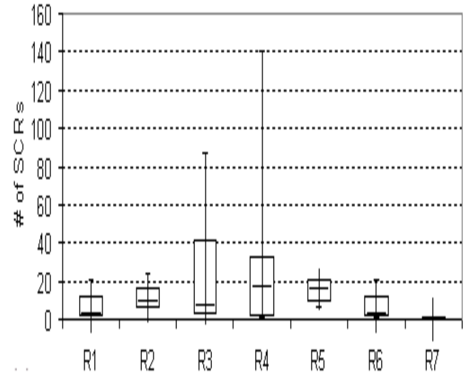

Figure 8.8: Box plot showing the number of SCRs due to requirement faults per release

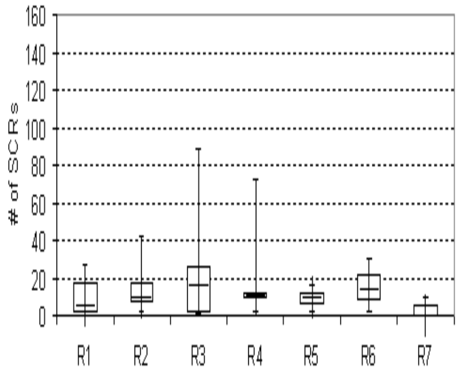

Figure 8.9: Box plot showing the number of SCRs due to coding faults per release

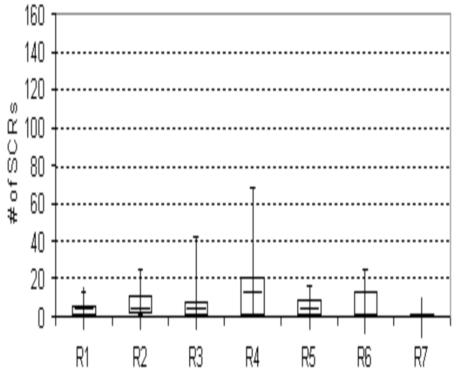

Figure 8.10: Box plot showing the number of SCRs due to data problem faults per release

the whiskers belongs to component 20, especially in releases R3 and later. We also observed that for each fault type the number of failures varied most in releases R3 and/or R4, which is likely due to the the large number of SCRs reported by component 20 in R3 and R4 (see Figure 8.4). Notice that the 25^{th} to 75^{th} percentile boxes for each release of the common types varied considerably less. Another perhaps more interesting observation is the fact that the median for each fault type is stable across all releases, which means that *although some components tend to experience larger variations in the distribution of fault types across releases, there is a clear central tendency.*

For releases R1 through R5 we formally verified the stability in the medians for each common fault type using the Kruskal-Wallis test, which tests the null hypothesis that different samples (i.e., in our case each release) were drawn from distributions with the same median. Kruskal-Wallis test is based on ranks of the combined sample values of the k samples. The

	Coding Faults	**Requirement Faults**	**Data Problems**
H	1.23	4.71	1.30
p-value	0.87	0.32	0.86

Table 8.1: Kruskal-Wallis H statistics and corresponding P-value for major fault types

statistic H used by the Kruskal-Wallis test is calculated by

$$H = \frac{12}{n(n+1)} \sum_{i=1}^{k} \frac{T_i^2}{n_i} - 3(n+1) \tag{8.2}$$

where k is the number of samples, n_i is the number of observations in i-th sample, $n = \sum n_i$ is the total number of observations in all samples combined, and T_i is the sum of ranks in i^{th} sample.

It can be shown that H is distributed approximately as chi–square distribution with $k-1$ degrees of freedom. The null hypothesis may be rejected if the probability of $\chi^2 = H$ is less than $\alpha = 0.05$. Note that samples can have a different number of observations n_i. In our case, for each attribute we consider $n_1 = 11$ components for R1, $n_2 = 11$ components for R2, $n_3 = 9$ components for R3, $n_4 = 5$ components for R4, and $n_5 = 4$ components for R5. Therefore, $n = 40$ for the 5 groups (i.e., the five releases). Table 8.1 shows the calculated H values and the probability of occurrence of equal or greater value of H under the null hypothesis for each of the three common types of faults. Since for each fault type (i.e., requirements faults, coding faults, and data problems) the $p-value$ is larger then $\alpha = 0.05$ we cannot reject the null hypotheses that releases are drawn from the distributions with the same median number of failures due to corresponding fault type. Thus, we conclude that for each the major faults types the medians were drawn from similar distributions in each release.

8.3 On-orbit failures per release

The study of post–release failures (i.e., on-orbit failures in our case) is of utmost importance for every system, and especially for safety-critical systems. Hence, we explored the following research question:

> RQ3–C: Are certain releases more likely to exhibit on–orbit failures? Further, is there a relationship between the number of failures reported during development and testing and the number of failures reported on–orbit?

The distribution of pre-release (i.e., *development and testing*) failures and post–release (i.e., *on-orbit*) failures across releases for all eleven components cumulatively is shown in Figure 8.11. Additionally, we explored the number of post-release failures reported per release for each component individually. Based on Figure 8.11 and the per release per component analysis we made the

- **On-orbit failures accounted for a very small percentage of total failures per release.** As shown in Figure 8.11, across different releases between 0 - 8%[2] of failures per release occurred on-orbit.
- **The software is improving and stabilizing in the field**. The percentage of on-orbit failures consistently decreased after release R2 for all but one component (i.e. component 18). It should be noted that due to the ongoing iterative development on-orbit failures against later releases have occurred after the date our data sample was taken. Nevertheless, the occurrence of on-orbit failures decreases as components and the project mature through releases.

The facts that (1) *on–orbit* failures accounted for a very small portion of failures per each release and cumulatively for all releases, and (2) the percentage of on-orbit failures decreased as the software matured through releases suggests that the verification and validation procedures in place have been very successful at removing faults. Other studies report on similar values. For example, 4% of faults were detected post–release for each of the two releases studied in [26], while for each of thirteen releases studied in [54] between 0% and 9% of the faults were reported post–release.

Since on-orbit failures tend to be more expensive to fix and are more likely to be associated with a higher severity it would be helpful to know which components are more likely to experience on-orbit failures. Intuitively, it could be argued that components that exhibit more failures during development and testing (i.e., pre-release) are well-tested and therefore unlikely to fail in the field. On the other hand, it could also be argued that some components may be failure prone due to fundamental reasons and therefore will continue to fail post–release. Further, since results from the related work were inconsistent with respect to the conflicting theories (pertaining to which modules are likely to be failure prone post–release) we explored the relationship between the number of pre-release failures (i.e., failures identified

[2] Note that the percentage of on-orbit failures is for the eleven components considered in this chapter and therefore differs from the percentage reported in chapter 6 for all 21 components.

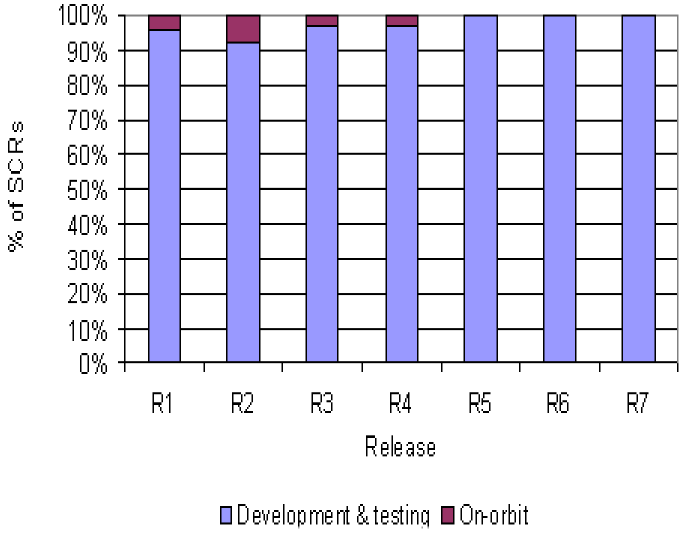

Figure 8.11: On-orbit failures per release

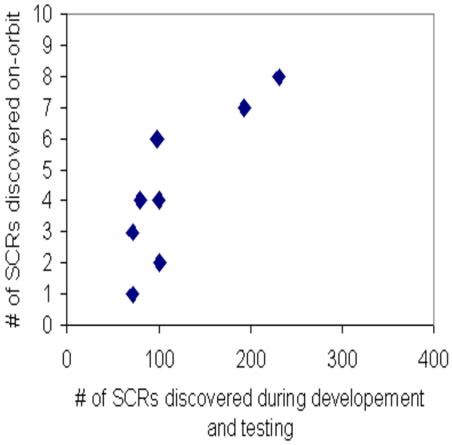

Figure 8.12: Scatter plot showing the relationship pre- and post-release failures, cumulatively over all releases, ($r_s = 0.80$)

during development and testing) and the number of post–release (i.e., on-orbit) failures first cumulatively for all releases and then within individual releases.

The scattered plot of the pre–release failures versus post–release failures, cumulatively over all releases for the eleven components considered in this chapter, which is shown in Figure 8.12, indicates a positive correlation. Therefore, we formulated and formally tested the following null hypothesis:

> $H7_0$***: The number of post–release (i.e., on–orbit) failures is unrelated to the number of failures that occurred pre–release (i.e., during development and testing), cumulatively for all releases, at component level.***

Since the distributions of the number of pre-release and post–release failures fail the Shapiro–Wilke W test for normality, we again used the Spearman rank correlation coefficient given with equation (8.1) to test the null hypothesis $H7_0$. When considering the relationship between the cumulative number of SCRs entered against each component during development and testing with the cumulative number of SCRs entered on orbit the computed value of the Spearman rank correlation coefficient $r_s = 0.80$ indicates very strong correlation. To test the significance of the correlation, we randomly select six out of the eleven components five times and calculate r_s using the equation (8.1) for $N = 6$. The computed values for these five random samples are $0.93 < r_s < 0.99$, which are statistically significant at $\alpha = 0.05$ level since the critical value for $N = 6$ is $r_s = 0.83$. Based on the observation in the data from all

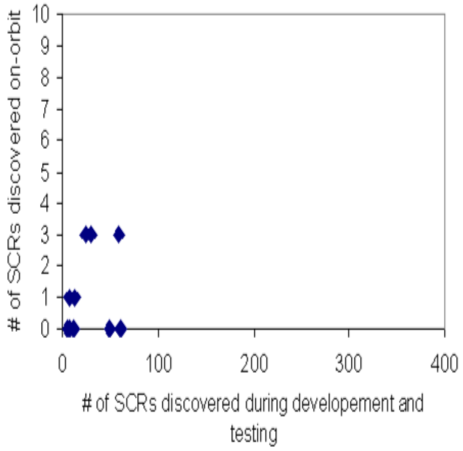

Figure 8.13: Scatter plot showing the relationship pre- and post-release failures for release R1, ($r_s = 0.53$).

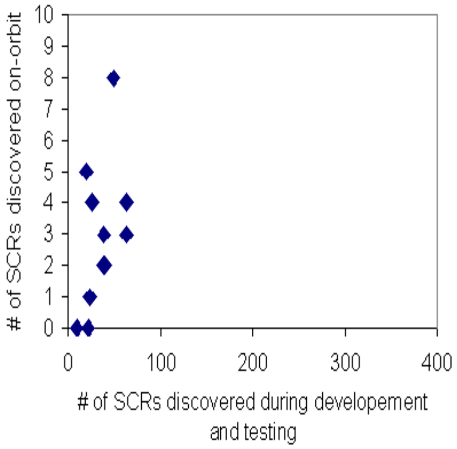

Figure 8.14: Scatter plot showing the relationship pre- and post-release failures for release R2, ($r_s = 0.51$).

eleven components and the significance of the association in the random samples, we reject the null hypothesis $H7_0$ in favor of the alternative hypothesis that

> $H7_A$: ***At the component level, cumulatively over all releases, the number of post–release failures is positively correlated with the number of pre-release failures.***

From a software engineering perspective this means that cumulatively over all releases, components which had more non-conformance SCRs during *development and testing* also tend to have more *on-orbit* failures.

Further, we explored the relationship between pre-release and post–release failures within individual releases. The analysis was restricted to releases one to four (R1 – R4), since no on-orbit failures were reported in our data set for releases five through seven (R5 – R7). The corresponding scatter plots are shown in Figures 8.13 through 8.16, where each dot represents a component. Note that for R1, component 8 and component 21 were excluded as release R1 was never loaded on orbit for either of these components. The values of the Spearman correlation coefficient r_s which were computed using the equation (8.1) are given in Table 8.2 with the corresponding sample sizes N, that is, the number of components that have valid pre- and post–release data for the release.

These results showed moderate to strong positive correlation, which means that at the component level within individual releases a high incidence of pre-release failures implied a high incidence of post–release failures. In other words, for releases one through four, the

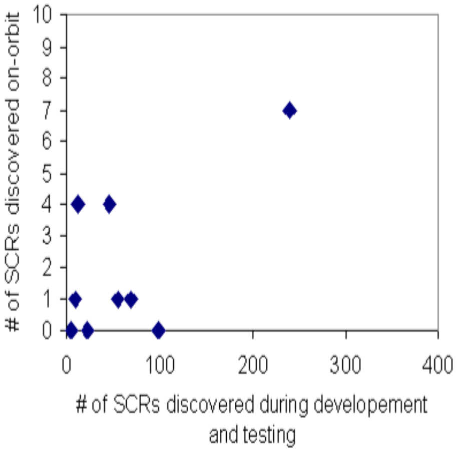

Figure 8.15: Scatter plot showing the relationship pre- and post-release failures for release R3, ($r_s = 0.47$).

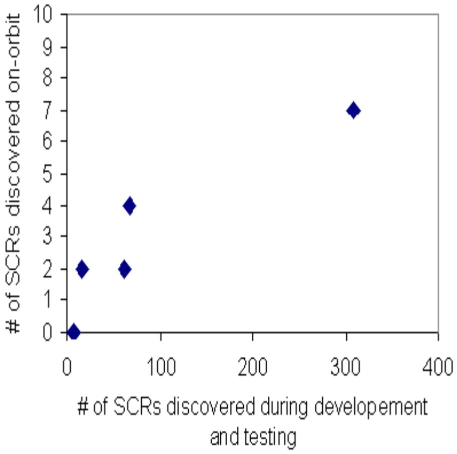

Figure 8.16: Scatter plot showing the relationship pre- and post-release failures for release R4, ($r_s = 0.96$).

Release	**R1**	**R2**	**R3**	**R4**
N	9	11	9	6
r_s	0.53	0.51	0.47	0.96

Table 8.2: Spearman correlation coefficient between pre-release and post–release failures, at component level, for releases one to four

components of the NASA mission that had a high number of non-conformance SCRs during development and testing tended to experience more on-orbit failures.

The relationship between pre–release and post–release faults has been explored before (e.g. [1], [11], [26], [54]). However, the results were conflicting. Thus, Fenton in [26] and Ostrand and Weyuker in [54] found that modules that were fault prone pre-release were among the least fault prone modules post–release. On the other hand, studies conducted by Biyani and Santhanam in [11] and Anderrson and Runeson in [1] (which is a replication of [26]) led to the opposite result: modules that were fault prone pre-release were also fault prone post–release. Further the Pearson correlation coefficient value used to measure the extent of correlation in [1] also showed moderate correlation, that is, $0.56 \leq r \leq 0.72$ for each of the three projects. Our results for the cumulative number of failures over all releases, as well for the individual releases one through four, support the observations made in [11] and [1], but due to a small number of on–orbit failures (i.e., less than 8 in any release of any component) we refrain from drawing strong conclusions. Future work with respect to the relationship between pre- and post- release failures should consider mutliple levels of abstraction to help answer this research question.

8.4 Exploring safety–critical failures within and across releases

Finally, we quantify the severity of failures across releases, first cumulatively for all eleven components considered in this chapter, and then for each individual component to answer the following research question:

> RQ3–D: Does the severity level of failures change as the software matures both within and across releases? Specifically, are safety–critical failures more likely to occur in earlier or later releases?

The per release data showed that almost every release of each component had been associated with both critical and non-critical failures. However, we did observe some interesting trends:

- **A very small percentage of failures within each release are safety–critical.** As shown in Figure 8.17, depending on the release, from 0% – 12% of failures reported against each release were safety–critical.

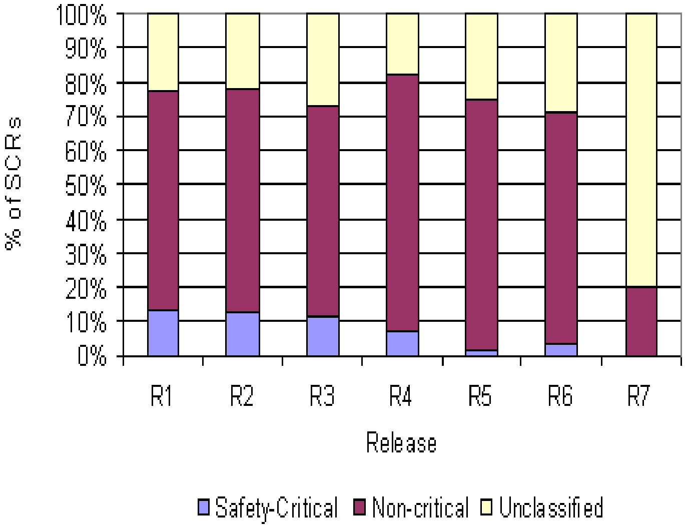

Figure 8.17: Severity levels per release

- **The percentage of safety–critical failures tends to decrease as components mature.** The majority of components showed a consistent decrease in the number of critical failures per release. A few components, which showed an increase of safety-critical failures in the initial releases, eventually showed a decrease in later releases. We suspect this may be a result of the functionality added.

The phenomenon of the increasing number of safety–critical failures for the initial releases of some components may be due to problems with integration of components as some unexpected combinations of rare events between different components may take time to surface.

Chapter 9

Investigation of fixes

Up to this point the research questions were focused on characterizing failures and the faults that caused them. In this chapter we present the results of our analysis of the changes made to address the non-conformance SCRs. As stated earlier, when changes are made to address an SCR, Change Notices (CNs) is created to track the changes per component. It should be noted that almost all cases, changes were made to correct the fault(s) that caused the failure (observed or potential), but in some much more rare cases changes were made to implement a workaround that prevents the reactivation of faults, which may or may not be fixed at a later date.

As stated earlier, one of the main goals of the analysis of fixes was to consider all changes to fix all faults associated with an individual failure, as our earlier work [37] clearly showed failures can often be associated with multiple faults. The existing relationship between SCRs and CNs made this possible (i.e., an SCR can have zero, one, or multiple CNs associated with it). Considering both types of change documents, that is, SCRs and CNs, allowed us complete the link from the fault(s) that caused individual failures to changes made to prevent the failures from (re)occurring, which to the best of our knowledge has never before been done. Specifically, we explored the fourth set of research questions:

RQ4: What are the common characteristics of fixes? And, how do features of fault and failures relate to features of fixes?

A. Which components are affected most often by fixes? Further, are any groups of components commonly fixed together?

B. Which types of software artifacts were fixed most often?

C. Is there any relation between the type of fault that causes a failure and the types of artifacts that need to be fixed?

D. Is there any relation between the verification activity that revealed the fault(s) and the types of artifacts fixed?

E. In terms of the artifacts fixed, do safety–critical failures differ from non-critical failures?

F. Is the effort spent implementing fixes uniform per SCR? per component? If not, what are the common characteristics of fixes that required the most effort?

The original data set analyzed with respect to fault and failure features consisted of 2,558 SCRs, 2,051 of which had at least one associated CN. However, some of the SCRs were still 'open' at the time of the data dump, that is, they had not been fully addressed and/or verified. To avoid any bias that might be introduced by considering SCRs that may require additional changes we focused the fix analysis to SCRs and CNs that had officially been closed, that is, the fixes had been implemented and verified. Thus, the analysis in this section is based $1,257$ closed SCRs and the $2,620$ associated CNs[1].

The number of closed SCRs and associated CNs are given in Table 9.1 for each of the 21 components. Table 9.1 only contains 2,496 CNs because 124 of the CNs associated with closed SCRs were associated with *simulation* components or *other* components which were outside the scope of the 21 components analyzed. The most interesting trend is that almost every component was fixed more times than it suspectedly failed. This is clearly a direct result of the fact that we consider multiple CNs per individual SCR, however, it serves to shows how much information may be missing for studies that assumed each failure was traced to a single localized fault. Additionally, the evidence that failures were caused by multiple faults generalizes our earlier results in [37], which showed a similar trend on a small subset of the failures considered here, that is, failures cause by coding faults that could be mapped to changed in source code files.

Based on the data available in the CNs we explored types of artifacts fixed, the components fixed and the effort required to implement the fixes to answer the fourth set of research questions (i.e., RQ4). First we explored features of fixes individually and then with respect to how they relate to fault and failures features explored earlier.

[1] It should be noted that the observations made throughout the previous chapters remain consistent on this subset on closed SCRs.

component	**# of closed non-conf SCRs**	**# of CNs**
1	179	252
2	8	14
3	4	3
4	4	6
5	4	6
6	15	18
7	33	43
8	44	86
9	1	9
10	8	10
11	24	36
12	11	17
13	35	72
14	28	53
15	23	44
16	44	70
17	112	158
18	104	169
19	102	197
20	412	1157
21	62	76
Total	1,257	2,496

Table 9.1: Number closed SCRs and associated CNs per components

9.1 Components Fixed

We began by exploring the components affected by changes made to correct faults and prevent the failures reported through SCRs from re-occurring. Hence, we explored the following research question:

> RQ4–A: Which components are affected most often by fixes? Further, are any groups of components commonly fixed together?

As stated earlier, the fact that each SCR can be linked to more than one CNs allowed us explore multiple fixes per failure (i.e., non-conformance SCR). For the 1,257 closed SCRs:

- 82% led to changes only in the same component the SCR was written against.
- 15% led to changes in the same component the SCR was written against, as well as at least one other component.
- 3% led to changes in others component(s) then the component the SCR was written against.

As expected, the majority of SCRs led to changes (i.e., CNs) only within the component the non-conformance SCR was filed against; however, 15% of SCRs resulted in changes to a different component. According to the project personnel, the spread of fixes can be attributed to several factors, such as, the interdependence of components and the fact that in some cases there is more than one way to implement a fix. The cases where the failed component was not fixed at all may be explained by the fact that sometimes a component simply served as a catalyst for a failure to surface even when all associated fault(s) were contained outside that component, especially if data is shared between components. In addition, for some cases for the quickest and easiest way to fix a problem was through a different component. It should be noted that the small percent of SCRs that did not result in any change to the 'failed' component (i.e., 3%) indicates that the SCR the non-conformance was reported against was indeed contributing to the non-conformance and thus supports the quality of the data in terms of representing the fault, failure and fix relationships.

Table 9.2 presents the mean number of CNs associated with each SCR broken down per component[2]. It also provides the standard deviation and coefficient of variation[3]. It can

[2] Statistical moments are not given for component 9 because as shown in Table 9.1 only one closed SCRs associated with component 9 had fix data.

[3] The coefficient of variation is defined as a ratio of the standard deviation and the mean and thus can be used to compare the variation for different components, even if the means are drastically different from one

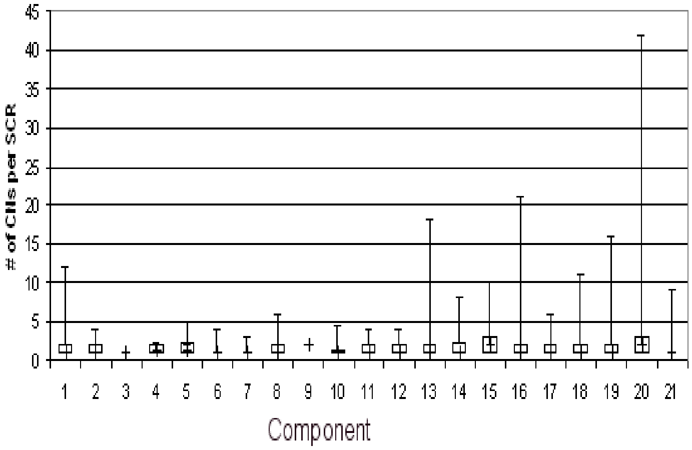

Figure 9.1: Box plot showing the number of fixes linked to individual failures per component

be seen that the means range from 1.00 to 2.77 CNs per SCR for the 21 components, and the largest means tend to have the higher coefficients of variation. For some components, individuals SCRs required a much larger number of the CNs, which can be seen in the box plot shown in Figure 9.1. For each component, the box shown in the graph contains the 25^{th} to 75^{th} percentile of the numbers of CNs filed per SCR, and the median number of CNs represented by the horizontal line within the box. The whiskers (i.e., the vertical lines that reach outside the box) represent the range and show the outliers. The box plot shows both the central tendency and dispersion in the number of CNs filed per SCR for each component. The fact that components have similar medians (typically around 1 to 2 CNs per component) shows there is a clear central tendency. In addition, since all boxes are relatively small, but several have whiskers that are much longer (see components 13, 16, and 20) we conclude that there is fairly small dispersion in terms of $75^{th} - 25^{th}$ percentile, but are reminded that some SCRs associated with several components are outliers that resulted in considerably more CNs (e.g., up to 41 CNs created to address one SCR filed against the component 20).

9.1.1 Failed-fixed Component Relationship

Considering that some non-conformance SCRs were associated with multiple components we decided to explore the relationship between the component the non-conformance was reported against (i.e., the failed component) and the components that were fixed. Thus,

another

component	Mean # of CNs per SCR	Standard deviation	Coefficient of variation
1	1.63	1.39	0.86
2	1.50	0.76	0.50
3	1.00	0.00	0.00
4	1.50	0.58	0.38
5	2.00	1.41	0.71
6	1.27	0.80	0.63
7	1.24	0.56	0.45
8	1.64	0.92	0.56
9	NA	NA	NA
10	1.63	1.19	0.73
11	1.42	0.58	0.41
12	1.55	0.69	0.44
13	2.49	3.35	1.35
14	1.96	1.55	0.79
15	2.35	1.80	0.77
16	2.20	3.05	1.39
17	1.46	0.90	0.62
18	1.83	1.59	0.87
19	2.15	2.33	1.08
20	2.77	3.32	1.20
21	1.53	1.34	0.87

Table 9.2: Mean, standard deviation, and coefficient of variation for number of fixes linked to individual failures per component

we defined the failed component - fixed component relationship. It should be noted, as mentioned earlier, close to 97% of the non-conformance SCRs led to at least some changes within the same component the SCR was written against, which suggests that the component the non-conformance was written against contained at least one fault associated with that non-conformance.

Figure 9.2 shows a 3-dimensional plot representing for each failed component the number of times each of the other 21 components were fixed (i.e., the cumulative number of CNs associated with the cumulative number of SCRs written against the failed component). Exploring the failed component - fixed component(s) relationship, led us to study how the software architecture relates to the spread of fixes across components. Therefore, we have re-arranged the order of components shown on the axes in Figure 9.2 specifically to reflect the software architecture. Thus, the first component listed on both the *Failed component* and *Fixed components* axes in Figure 9.2 is component 20, which is the single top level component in the 3-level hierarchical architecture. Component 20 is then directly connected to components 1, 17, 21, 19, 14, and 18 at level two, resulting in six groups of components shown in Figure 9.2, annotated as A through F. The components that connect each group to the top level component appear first in each of the groups A through F. The remaining components span the second and third levels in the architecture.

The main observations based on Figure 9.2 were:

- **Fixes typically occurred in the failed component.** As shown by the large values on the diagonal the majority of CNs were tied to the same component against which the originating SCR was written, which indicates that the failed component was at the very least contributing some fault(s) to a potential failure. In total, 74% of CNs were tied to the changes implemented in the same component against which the SCRs was reported[4]. The remaining 25% of changes were implemented in a component other than the one the SCR was written against. Thus, showing that one in every four faults lies outside the component reportedly failing.

- **A significant percentage of fixes were associated with the single top level component, that is, component 20**. 34% of the total number of CNs were implemented as either a result of an SCR filed against component 20 or to make a change

[4] Note that because each SCR can have multiple CNs, although 82% of SCRs required changes only within the failed component, only 74% of CNs were implemented within the failed components.

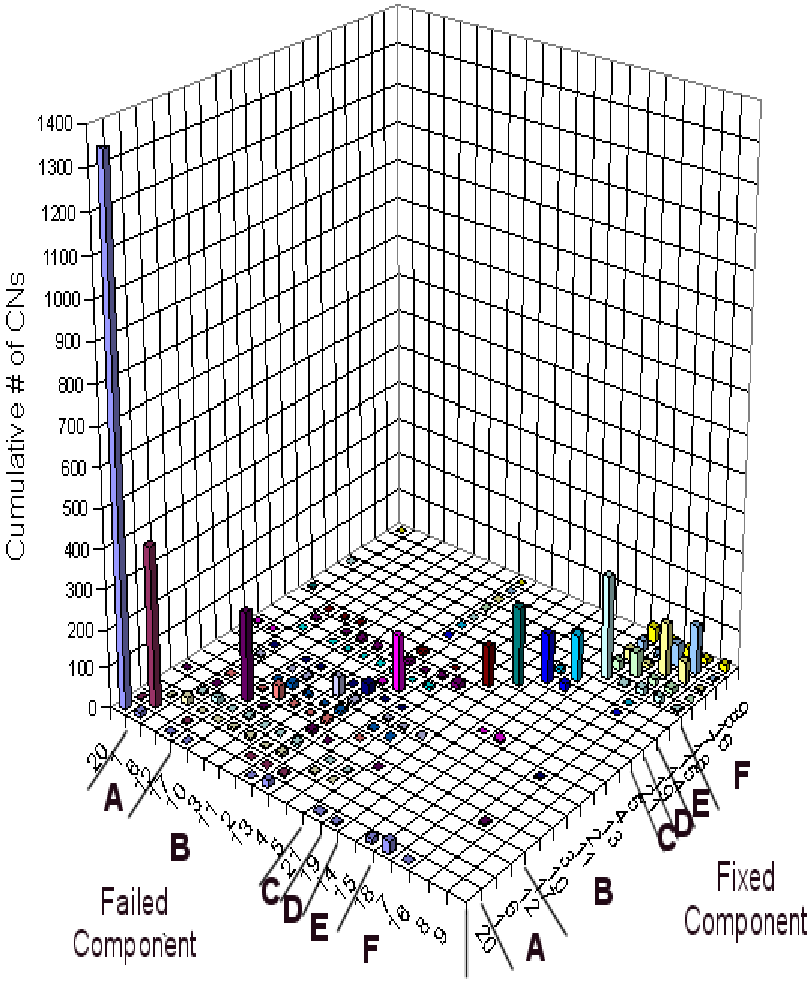

Figure 9.2: The cumulative number of fixes in each components based on the cumulative number of failures associated with the failed component

to component 20 as a result of an SCR filed against another component. We also noticed that the failed components that often led to fixes in component 20 tended to be components at the second level of the hierarchy which connect each architectural group (i.e., A through F) to the single top level component (i.e., component 20). This observation suggests that the spread of fixes is related to the software architecture.

What is perhaps most interesting about exploring the failed - fixed component(s) relationship is the set of SCRs that required fixes outside the component against which the SCR was filed. One would suspect finding and fixing faults outside this component would be more difficult, as it is less intuitive. Therefore, we specifically analyze such SCRs. When further exploring the spread of fixes that occurred outside the failed component we chose to include open SCRs for which some fixes had been implemented because the change notices that are yet to filed could only increase the spread of fixes. At the component level, it is very unlikely that all changes made to a component would be complete reverted upon verifying the fix. Thus, including open SCRs is still likely to be an understatement with respect to the spread of fixes across components, as additional change notices can only increase the spread of fixes.

Figure 9.3 is similar to Figure 9.2 but included data for both opened and closed SCRs[5], while the fixes that were implemented in failed components (i.e., data points that lie on the diagonal in Figure 9.2) were removed so that patterns off the diagonal that reflect changes outside the failed component could be explored. Note that the scale on the z-axis which shows the cumulative number of CNs has changed. The following interesting observations with respect to the effect of the software architecture on the distribution of fixes were made based on Figure 9.3):

- **Changes to components in group F were the most frequent and typically localized within group F.** In other words, SCRs filed against components from Group F rarely led to fixes outside the group. Further, we noticed that either of component 8 or component 16 failures are likely to result in fixes in the other. The most prominent exception from the changes being localized within group F is the large number of failures of component 18 that affected component 20, which can be explained by the fact that component 18 is the component at the second level which connects

[5] Once again, we note that we repeated the analysis for the claims made thus far and confirmed that the major trends did not change. Further, it remains true that about 18% of open and closed SCRs led to changes outside the components against which the SCR was written, which is the same percentage of closed SCRs.

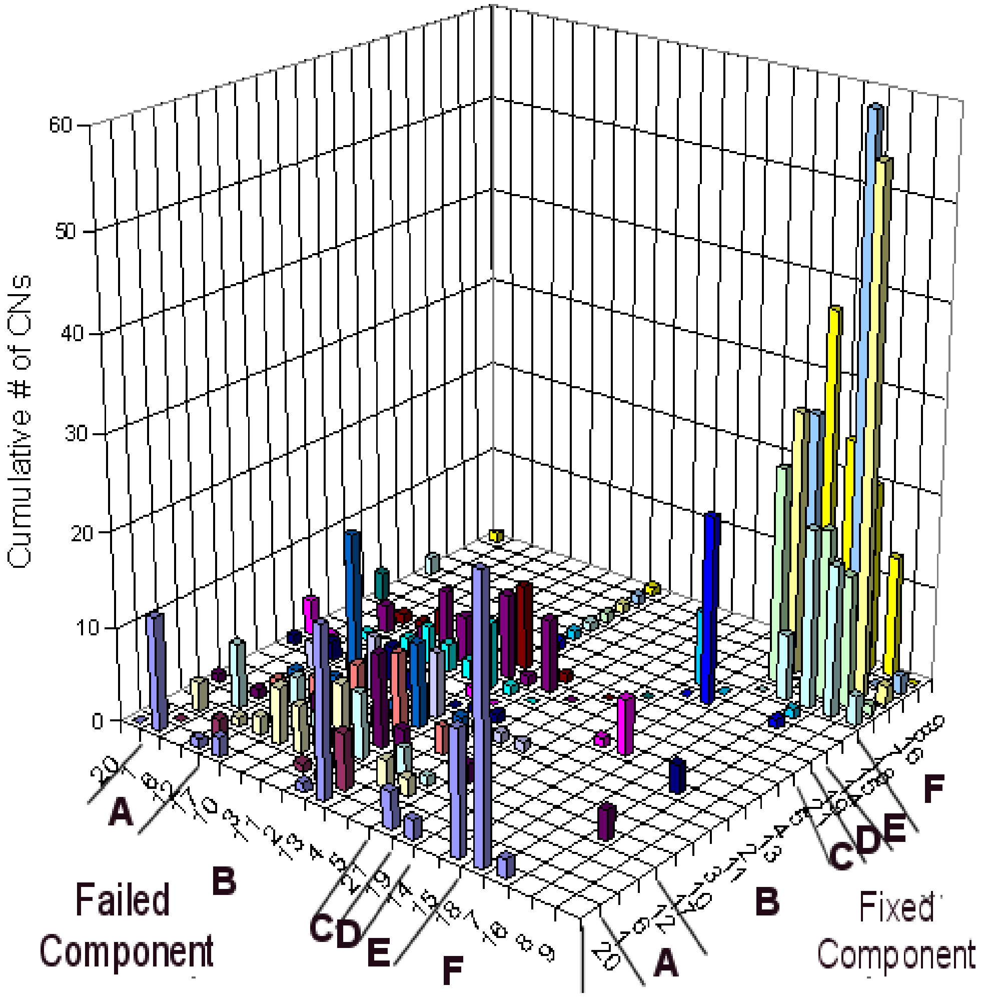

Figure 9.3: The cumulative number of fixes that led to changes outside the failed component

group F to component 20.

- **Fixes typically remained within the architectural group.** 74% of fixes that occurred outside the failed component remained within the architectural groups. However, failures in some components required fixes to other groups. For example, a failure in component 2 may require fixes in almost any other component. In fact, only component 19 has never been fixed as a result of a failure in component 2. Additionally, failures in component 6 (which belongs to group A) were likely to cause changes in group B. According to project personnel this can be explained by the fact that functional groups are interconnected to perform overall integrated mission operations.
- **Components with similar functionally showed similar change patterns.** Components 4, 5, 6 and 12 which are similar in terms of their functionality showed very similar change patterns across all failures, as was also the case for components 14 and 15.

These result suggest that software architecture has an impact on the distribution of fixes among different groups of components. According to project personnel this information is useful for conceptualizing integrated independent analysis tests. Further benefit may come from identifying non-typical behaviors, such as for example the fact that the number of changes in group F also high, or that failures linked to component 2 may lead to changes in almost any other component. These observations can be useful in increasing the efficiency of fixing future faults by helping the developers and verification and validation personnel to identify components that typically change together.

9.2 Types of Software Artifacts Fixed

Faults contained within any of the software artifacts used throughout the development life cycle can potentially cause a failure. Since analysis techniques can be specific to certain artifacts (e.g., testing of code versus manual analysis of code or requirements) we explored the distribution of faults across artifacts to answer the following research question:

RQ4–B: Which types of software artifacts were fixed most often?

Quantifying these trends could help managers allocate resources accordingly. Each CN identified the software artifact that was fixed in the ‘affected product’ field. This is a free-text field and thus, the level of detail describing each artifact varied greatly; in some cases specific file names or documents names were given, while in other cases there were only high level descriptions. Based on domain expertise of project personnel, we grouped the artifacts into the following categories:

- *requirements documents* (e.g., detailed requirement specifications)
- *design documents* (e.g., high-level design overviews)
- *code files* (e.g., Ada files)
- *supporting files* (e.g., pre-defined look up tables)
- *tools* (e.g., testing, simulation and configuration tools)
- *notes* (Notes are written for the onboard crew to help prevent known failures, for example, by instructing the crew to avoid or alter certain command sequences. Therefore, a note can be considered to be a work-around)
- *waivers* (Waivers on the other hand are written to acknowledge that a requirement cannot be met and that the mission is currently willing to accept the limitation.)

74% of the non-conformance SCRs identified only one type of artifact (although in some cases multiple instances of a single type of artifact were changed, such as two different requirement documents), 20% identified two types of artifacts, and slightly less than 6% identified three or more types of artifacts affected by fixes. Hence, in addition to exploring which single type of artifacts were fixed most often, we also explored which types of affected artifacts were often fixed together.

The venn diagram in Figure 9.4 shows the common types of artifacts and groups of artifacts fixed by changes[6]. We made the following observations with respect to the types of artifacts fixed:

- **The vast majority of SCRs resulted in changes to requirements, code, or the combination of both.** Specifically, 39% of the SCRs resulted in changes to the requirements only (no other types of artifacts were affected), 33% resulted in changes

[6] The groupings shown in Figure 9.4 account for over 94% of the SCRs that identified artifacts changed. The reminding 6% of the SCRs cannot be shown in Figure 9.4 because of the extremely small percentages spread across less common groupings.

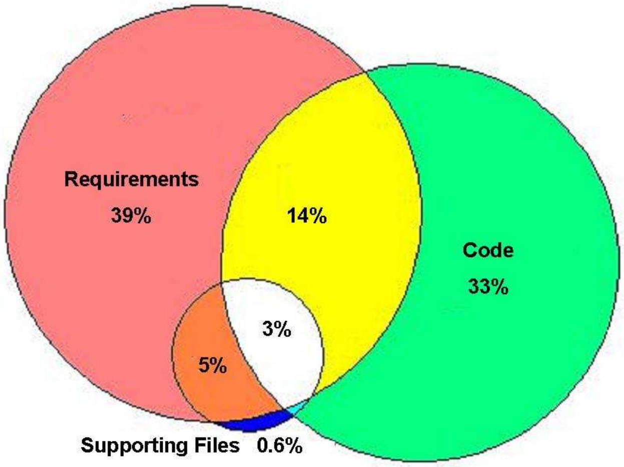

Figure 9.4: Types of artifacts fixed

to code and no other artifacts, and 14% required changes in both the requirements and the code.

- **The supporting files were almost always changed in conjunction with other artifacts.** Thus, while only around 0.6% of SCRs led to changes only in supporting files, 5% resulted in changes in code and supporting files and 3% resulted in changes in requirements, code, and supporting files.
- **The percentage of SCRs that led to changes in design artifacts was rather small.** Almost half of the SCRs that do not appear in Figure 9.4 (i.e., around 3% overall) required changes in at least some design artifact(s). The small percentage of SCRs linked to changes in design documents may to some extent be due to the fact that the mission did not maintain detailed design documents and according to project personnel when code changes impacted only a few lines, the design documents would not be updated.
- **Workarounds were sometimes used to prevent failures from reoccurring**. The analysis of the artifacts fixed showed that less than 2% of SCRs identified notes and/or waivers as fixed artifacts. At first this seemed surprising for a large safety-critical system that required sustained engineering; however, project personnel pointed out an additional change document, which was stored solely for the purpose of tracking workarounds. Initial analysis of this documented showed that were around were more commonly used (i.e., 26% of the SCRs for which fixes where analyzed were associated with these additional documents).

9.3 Types of Artifact Fixed per Fault Type

We suspect that the fault type responsible for causing a (potential or actual) failure may be correlated with the type of artifacts affected by the fixes. For example, a requirement fault is likely to lead to change in requirements. However, the association is not trivial since for example a requirement fault should also lead to changes in the source code if it was discovered after the code was implemented. Hence, we explored research question:

> RQ4–C: Is there any relation between the type of fault that causes a failure and the types of artifacts that need to be fixed?

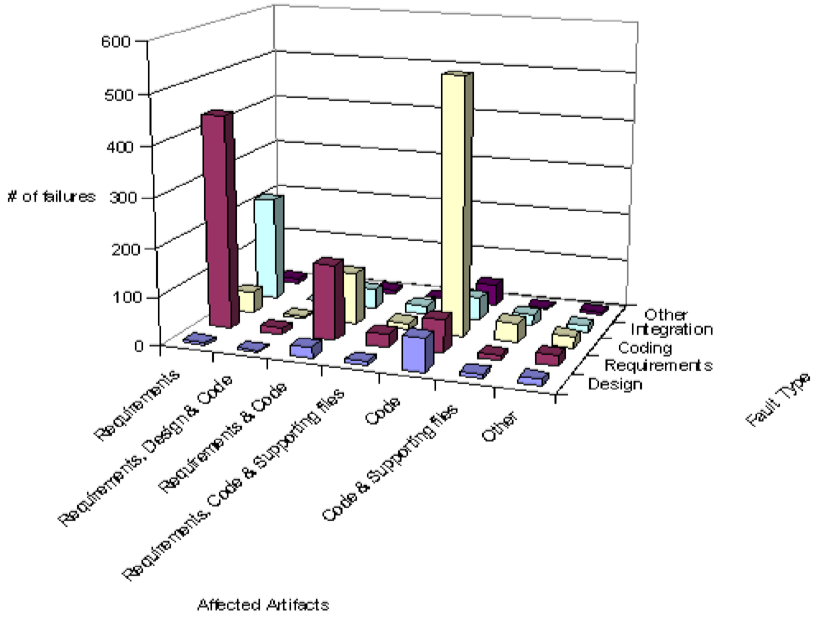

Figure 9.5: Frequency counts for types of artifacts fixed per major fault type

To explore the association between the types of artifacts affected by fixes and the fault types that caused individual failures, we tested the following null hypothesis:

$H8_0$***: The artifacts fixed are unrelated to fault type.***

By calculating $\chi^2 = 609.21$, and then $C = 0.57$, $C_{max} = 0.91$, and $C^* = 0.63$ we reject $H8_0$ in favor of the alternative hypothesis that *the type of affected artifacts are correlated with fault types*. The correlation is strong and statistically significant at 0.05 significance level. The correlation between fault types and types of affected artifacts is twice that of any other correlation between fault and failure attributes measured in Chapter 7.

The 3-dimensional plot in Figure 9.5 shows the frequency counts of the common types or group of types of affected artifacts for each major fault type. Table 9.3 shows the corresponding data presenting the actual numbers (and percentages in brackets) of non-conformance SCRs for each fault type that led to changes in the common grouping of artifacts.

With respect to major fault types we made the following observations:

- **The majority of coding faults were fixed by changing the code only.** 66% of the failures that were caused by coding faults (i.e., 22% of the total non-conformance SCRs) resulted in a change only in the code and no other type of artifact. This is the highest percentage among all combinations of fault type and affected artifacts.

Types of Artifacts Fixed	Fault Type					
	Requirements	Design	Coding	Integration	Other	Total
Requirements	251(19.97)	4(0.32)	27(2.15)	199(15.83)	9(0.72)	490(38.98%)
Requirements Design & Code	9(0.72)	1(0.08)	4(0.32)	2(0.16)	0(0.00)	16(1.27%)
Requirements & Code	86(6.84)	10(0.80)	52(4.14)	23(1.83)	7(0.56)	178(14.16%)
Requirements, Code & Supporting Files	16(1.27)	4(0.32)	9(0.72)	9(0.72)	1(0.08)	39(3.10%)
Code	41(3.26)	33(2.63)	281(22.35)	26(2.07)	29(2.31)	410(32.62%)
Code & Supporting Files	6(0.48)	8(0.64)	29(2.31)	14(1.11)	1(0.08	58(4.61%)
Other	20(1.59)	9(0.72)	22(1.75)	6(0.48)	9(0.72)	66(5.25%)
Total	429(34.13%)	69(5.49%)	424(33.73%)	279(22.20%)	56(4.46%)	1,257(100%)

Table 9.3: Frequency counts for types of artifacts fixed per major fault type

An additional 27% of the failures caused by coding faults (i.e., 7% of the total non-conformance SCRs) resulted in changes to the code, as well at least one other type of artifact. Surprisingly, we notice from Figure 9.5 that in some cases a coding fault may be fixed by fixing requirements only, that is, 6% of failures caused by coding faults (i.e., 2% of total non-conformance SCRs) resulted in changes to requirements only. This can be explained by the nature of the software developed by NASA. NASA missions do things for the first time and hence experience discovery of requirements throughout development, testing and even operation. Accordingly to the project personnel in these few cases the code may be implemented and/or updated on the fly and then the SCR was written to update the requirements to match the code. Similar observation related to discovery of requirements later in the life cycle was made for other NASA missions [45].

- **The majority of requirements faults were fixed by changing requirements only.** Table 9.3 and Figure 9.5 show that 59% of failures caused by requirement faults (i.e., 20% of the total non-conformance SCRs) resulted in changes in requirement artifacts and no other type of artifact. This suggests that a large portion of requirement faults are caught before the code for that faulty requirement was implemented. On the other hand 20% of failures caused by requirement faults (i.e., close to 7% of the total non-conformance SCRs) led to changes in requirements as well as code, and additional 4% (i.e., just over 1% of the total non-conformance SCRs) led to changes in requirements, code, and supporting files. For these 8% of the SCRs it appears the requirement faults, which can be incorrect or missing requirements, were detected later in the life cycle. Interestingly we also notice that some requirements faults were fixed by changing the code only (i.e., 10% of failures caused by requirement faults, that is, about 3% of the total non-conformance SCRs). This may indicate situations in which the requirement was unclear but the code was the only artifact affected due to lower level implementation details not included in requirements documents.

- **Integration faults often led to changes in requirement documents.** 72% of failures associated with integration faults (i.e., close to 16% of the total non-conformance SCRs) resulted in changes to requirements documents only. This can be explained by the fact that this NASA mission relies directly on *program, instrument, and command lists* which are implemented through requirements documents and directly affect software executions as they define commands and/or values to be used in certain conditions. Clearly, an emphasis should be put on defining, reviewing, and testing these commands and values which potentially would led to less integration faults. In general, more analysis and/or inspection conducted on all current integration requirements may be beneficial.

- **Design faults typically led to changes in code, often in combination with other artifacts.** 48% of the failures caused by design faults (i.e, below 3% of the total non-conformance SCRs) led to changes in only code and an additional 33% (i.e, 2% of the total non-conformance SCRs) led to changes in code as well as at least one other type of artifact (e.g. code and supporting files, or requirements, design, and code). According to project personnel, this can be explained by the fact that this NASA mission did not use very detailed design, and so design decisions were often directly implemented at the coding level.

- **Uncommon fault types most often led to changes in code only.** Fault types that were rare (e.g., i/o problems, simulation problems, fabrication/ manufacturing faults each accounted for less than 1% of non-conformance SCRs) were grouped into the *other* fault type category. Figure 9.5 shows that these fault types typically require changes to the code only.

Based on the analysis of the association between fault types and affected artifacts conducted, we emphasize several points which may benefit the NASA mission and potentially contribute towards more cost efficient improvement of software quality.

- **Compiling a checklist of common coding faults and incorporating it in the development and verification and validation process may significantly decrease the number of coding faults, and thus the overall number of non-conformance SCRs.** As noted earlier, 66% of the failures that were caused by coding faults (i.e., 22% of the total non-conformance SCRs) resulted in changes only in the code and no other type of artifact. This suggests that a large portion of coding faults are due to coding errors and may be avoided by compiling a checklist with common coding errors. Similar effort in Lucent Technologies decreased the number of coding faults by 34.5%, reduced the average testing effort by 18.3%, and shortened the development interval by 8.3%. The total saving due to reduced product rework and testing was seven million US dollars [68].

- **Some faults may have been detected earlier.** Based on the results presented in Figure 9.5 and Table 9.3, we were able to identify several groups of SCRs that may have been revealed earlier. These include 20% of failures caused by requirement faults (i.e., 7% of the total non-conformance SCRs) which led to changes in both requirements and code, and an additional 4% (i.e., just over 1% of the total non-conformance SCRs) which led to changes in combination of requirements, code, and supporting files, and 18% of the failures which were linked solely to changes in the code (i.e., close to 6% of the total non-conformance SCRs) but were actually caused by requirement faults or design faults. Theoretically, these requirements faults have a potential to be detected earlier (i.e., as early as the requirements phase) before they have been implemented thus would not have required fixes to other artifacts. Further exploration of such SCRs by domain experts may lead to earlier detection of faults. However, it should be noted in case of NASA missions earlier detection of requirement faults is not always feasible as some requirements will always remain to be discovered under the unknown circumstances that come with exploring the unknown for the first time.

9.4 Types of Artifacts Fixed per Verification Activity

Once again we were specifically interested in characterizing the high priority failure class of post-release failures. In this section we investigate how fixes differ in terms of artifacts fixed for pre-release and post-release (i.e., on-orbit) failures to answer the following research question:

> RQ4–D: Is there any relation between the verification activity that revealed the fault(s) and the types of artifacts fixed?

Formally, we tested the following hypothesis:

> $H9_0$***: The types of artifacts fixed are unrelated to pre/post-release detection.***

By calculating $\chi^2 = 26.96$, $C = 0.15$ we see that $C_{max} = 0.81$ and $C^* = 0.18$, we reject $H2_0$ in favor of the alternative hypothesis that *the affected artifacts are correlated with the pre/post release detection.* Although the measured correlation is weak, it is statistically significant at significance level of 0.05. Table 9.4 shows the percentage of failures discovered pre- and post–release for the common groups of affected artifacts. It should be noted that, as in case of any good quality, operational software, the sample size of post-release failures (i.e., on-orbit failures) is significantly smaller (i.e., 42 SCRs) than the sample size of potential failures prevented by fault detection and removal activities (i.e., 1,215 SCRs).

Table 9.4 shows that the majority of post-release failures (i.e., 50%) resulted in changes to source code only, which was not surprising considering the fact that 53% of post–release failures were caused by coding faults (see chapter 7.2. Thus, as expected the fact that the majority of coding faults were fixed by changes to just code is true for post-release failures as well as pre-release failures. Thus, we emphasize the potential of focusing on detect coding faults earlier in the life cycle by implementing and/or updating coding checklists used during development process and verification and validation activities.

9.5 Types of Artifacts Fixed for Safety–critical Failures

In this section we focus on the high priority safety-critical failure class. Specifically, we investigated whether different types of artifacts were fixed for safety-critical failures than for non-critical failures and explored the following research question:

Artifacts Fixed	**% of Development & Testing SCRs**	**% of On-orbit SCRs**
Code only	32.03	50.00
Requirements only	39.92	11.90
Requirements and code	14.24	11.90
Code and supporting files	4.20	16.67
Requirements, code, and supporting files	3.13	2.38
Requirements, design, and code	1.32	0.00
Other	5.19	7.14

Table 9.4: Artifact fixed pre- and post-release

RQ4–E: In terms of the artifacts fixed, do safety–critical failures differ from non-critical failures?

To answer this question we formulated and tested the following null hypothesis:

10_0***: The artifacts fixes are unrelated to the severity of the (potential) failure.***

By calculating $\chi^2 = 295.69$, $C = 0.43$, $C_{max} = 0.87$, and $C^* = 0.50$ we reject $H11_0$ in favor of the alternative hypothesis that *the type of software artifacts fixed are correlated with the severity level of failures.* The measured correlation is moderate and statistically significant at 0.05 level.

Further observations about the association of software artifacts fixed and the severity level of (potential or actual) failures can be made based on the frequency counts of the common groups of affected artifacts for critical, non-critical, and unclassified failures shown in the 3-dimensional plot in Figure 9.6. The main observations were as follows:

- **The relative contribution of failures that led to fixing multiple types of artifacts was more significant within the class of safety-critical failures than within the class of non-critical failures.** Over 67% of safety–critical failures led to changes in multiple types of artifacts, while only 29% of non-critical failures led to changes in multiple types of artifacts. This observation may be explained by several reasons such as (1) a larger percentage of on-orbit safety-critical failures occur when all software artifacts (including the code) exist and thus need to be changed to fix the problem, (2) safety–critical failures sometimes may be more complex than non-critical failures, and (3) safety–critical failures may sometimes be better documented than non-critical failures.

- **The relative contribution of failures that required changes in supporting files was more significant within the class of safety-critical failures than within the class of non-critical failures.** Specifically, 24% of safety-critical failure resulted in changes to the code and supporting files group compared to only 3% of non-critical failures. Additionally, 18% of safety-critical failures resulted in changes to the requirements, code, and supporting files group compared to only 2% of non-critical failures.

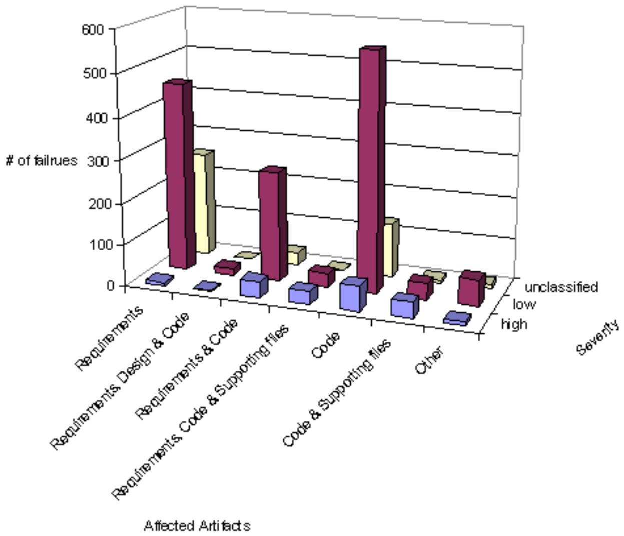

Figure 9.6: Frequency counts for types of artifacts fixed per severity level

- **Safety-critical failures rarely led to changes in requirement artifacts only.** Although rare, safety–critical failures may lead to changes in requirements only. Specifically, 4% of safety-critical failures led to changes only in requirements, which can be explained by the instrument program command lists used by the mission. Theses lists define commands and values to be used under certain conditions and thus are read directly during operation. When discussing the types of affected artifact with project personnel these instrument program command lists were considered to be requirements, however we note that in this cause a change to this requirements document can directly affect the software's execution behaviors under pre-defined conditions. It should be noted that, as in case of on-orbit failures, the sample size of safety-critical failures is significantly smaller (i.e., 127 SCRs) than the sample size for non-critical failures (i.e., 711). Therefore, the above observations are supposed to be taken with caution and should further be confirmed on additional safety–critical failures from this or other missions.

9.6 Fix Effort

A significant amount of effort in any major software development project is spent on verification and validation methods. Further, it has been estimated that changes due to maintenance and rework account for between 60 and 80 percent of total development costs [35]. Thus it is obvious that projects could benefit significantly from decreasing the effort spent on verification activities, maintenance and/or fixing faults.

In the NASA mission CNs have an effort field for the analyst to record the time spent implementing the changes described within that CN. As defined in section 2 the fix effort represents the total effort in hours spent implementing fixes for all the change notices associated with an individual SCR. Of the 1,257 closed SCRs , 92% (i.e., 1,153) recorded the effort spent implementing changes; cumulatively these 1,153 SCRs took 17,500 hours to fix[7]. Quantifying the effort spent implementing fixes per component, as well as per individual failure, could be useful in helping to predict future fix effort and identify where the most room for improvement is. Thus, we explored how the effort spent implementing fixes was distributed across features. First we explored the distribution of effort per component and then per SCR to answer the following research question:

> RQ4–F: Is the effort spent implementing fixes uniform per SCR? per component?
> If not, what are the common characteristics of fixes that required the most effort?

Thus, be began by exploring the relationship between the size of each component (given KSLOC), the number of SCRs written against the component, and the number of cumulative hours spent fixing faults in that component as shown in Figure 9.7. Each dot in the Figure 9.7 represents a component. Earlier we found that larger components tend to have more SCRs filed against them [37] and consequently we now see more cumulative effort was spent fixing these components. The pairwise relationships between between the number of SCRs and the fix effort, the size (KSLOC) and fix effort, and the size (KSLOC) and number of SCRs, are shown in Figure 9.8. As expected, the largest component (i.e., component 20) had the most SCRs and accounted for more of the total fix effort than any other component.

Figure 9.9 provides a histogram of the fix effort per individual failure on a logarithmic scale. The median effort spent per SCR was 3 hours, and values ranged from 0 to 1,193 hours (1% SCRs reported 0 hours). By sorting the SCRs based on the number hours spent implementing fixes we found evidence of the Pareto principle, that is, 20% of that SCRs account for 83% of the total amount of fix effort, which indicates of a skewed distribution. Thus, although fixing the faults associated with most of the SCRs required little effort (in

[7] In total, these 1,153 SCR took 2,140 hours to report based on the SCR effort field discussed in Section 6.4.

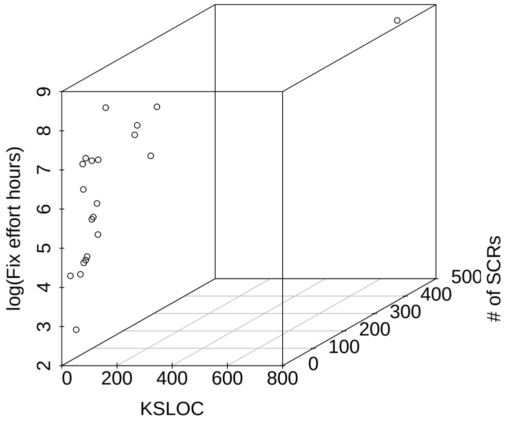

Figure 9.7: Relationships between component size (in KSLOC), the cumulative number of SCRs filed against the component, and the cumulative effort spent fixing the component

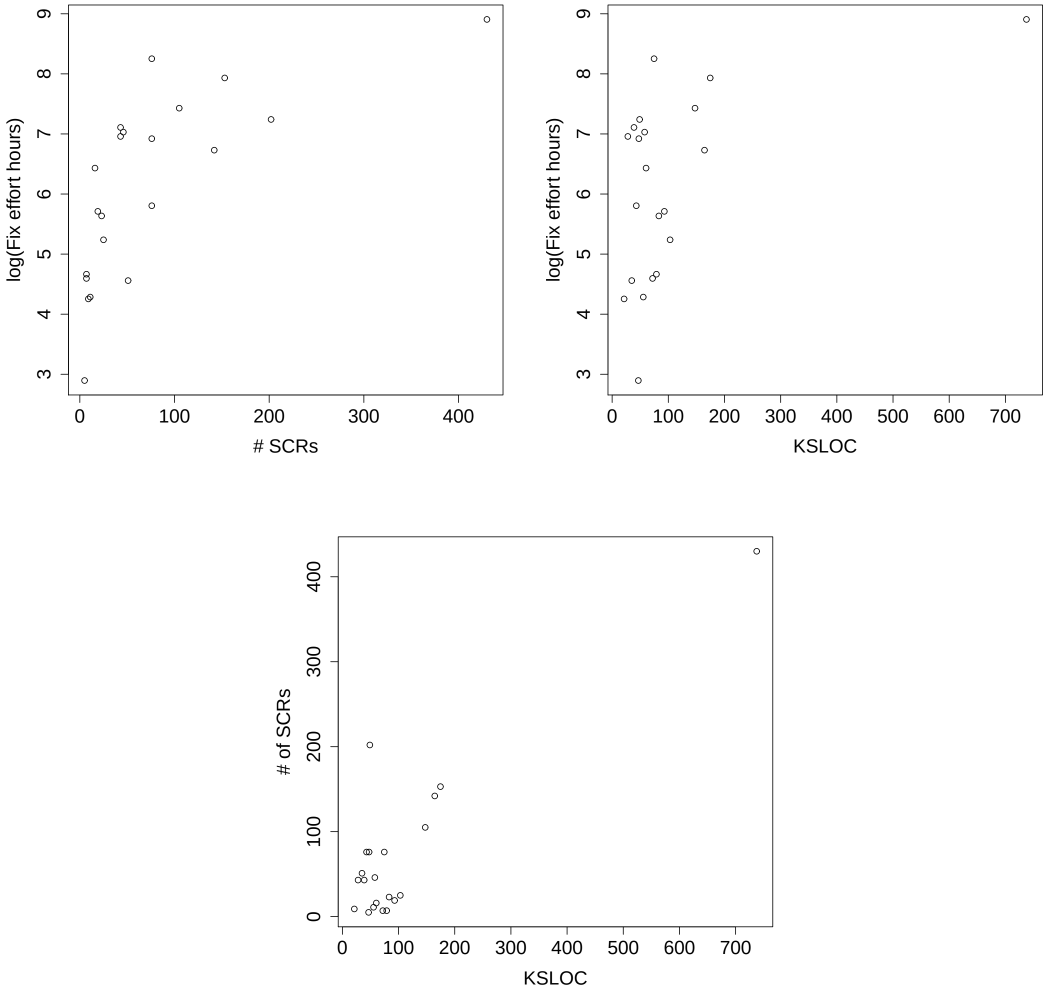

Figure 9.8: 2 dimensional projections for the 3 dimensional plot shown in Figure 9.7

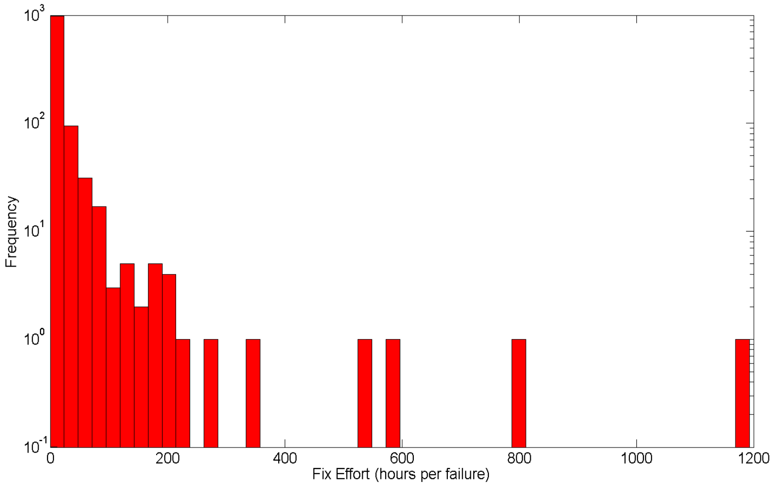

Figure 9.9: Fix effort per individual failure histogram

the range of several hours), some rare SCRs required a much more significant amount of effort (hundreds or even thousands of hours).

9.6.1 High Fix effort

As expected, fix effort was not uniformly distributed across components or SCRs. Further, we explored fix effort with respect to the fault types that caused the failures and the artifacts fixed, as well as, the fix effort associated with high-priority failure classes (i.e., post-release failures, and safety–critical failures) and the fix effort with respect to the spread of fixes (i.e, the number of components fixed and the number of types of artifacts fixed).

Fix effort with respect to fault type and types of artifacts fixed

Based on the result presented in Section 9.3 we know that there is a statistically significant association between fault type and types of artifacts fixed. Here we investigate how fix effort is distributed across these categories.

Table 9.5 shows the distribution of the effort spent for the major fault types and major groups of fixed artifact types[8]; Table 9.6 shows the number of instances in each category. More interestingly, Figure 9.10 and Table 9.7 show the effort per fault type and fixed artifact type normalized by the number of instances in each category, that is, the total hours spent

[8] Table 9.3 shows very similar information but for SCRs with CNs rather than just those with recorded effort values

Types of Artifacts Fixed	Fault Type				
	Requirements	Design	Coding	Integration	Other
Requirements	868.7	10.1	182.3	1,734.7	58.5
Requirements Design & Code	360.2	280.0	10.4	31.0	0.0
Requirements & Code	1,540.5	136.2	2,250.9	386.1	132.6
Requirements, Code & Supporting Files	1,332.1	87.0	232.9	776.1	17.0
Code	1,274.6	284.0	2,150.6	71.4	105.2
Code & Supporting Files	320.1	41.6	921.0	420.9	32.0
Other	170	22.2	387.9	810.9	45.5

Table 9.5: Total effort spent per fault type and fixed artifact group

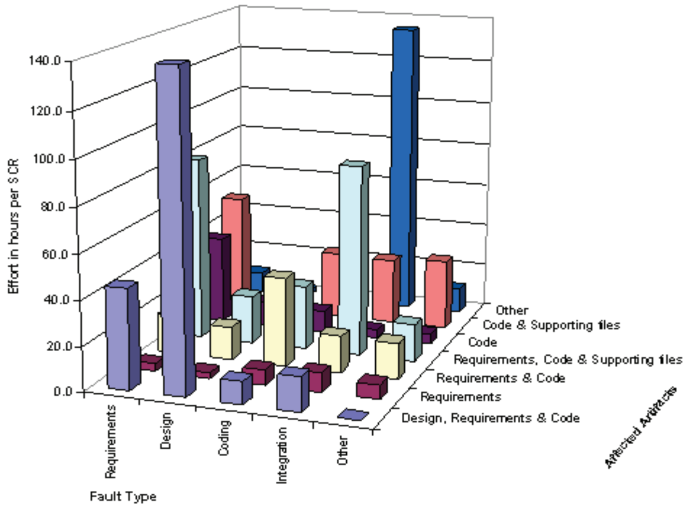

Figure 9.10: Cumulative effort (in hours) spent per fault type and fixed artifact group

per category divided by the number of SCRs in that category which represents the mean effort spent per category.

Our main observations include:

- **Requirement faults caught early typically required less effort to fix**. As expected, it can be seen that requirements faults which were corrected by only fixing

Types of Artifacts Fixed	Fault Type				
	Requirements	Design	Coding	Integration	Other
Requirements	249	4	27	199	9
Requirements Design & Code	8	2	1	2	0
Requirements & Code	87	9	56	23	8
Requirements, Code & Supporting Files	16	4	8	9	1
Code	32	25	216	19	22
Code & Supporting Files	6	8	30	14	1
Other	19	9	20	6	4

Table 9.6: Total number of SCRs per fault type and fixed artifact group

Types of Artifacts Fixed	Fault Type				
	Requirements	Design	Coding	Integration	Other
Requirements	3.5	2.5	6.9	8.7	6.5
Requirements Design & Code	45.0	140.0	10.4	15.5	0.0
Requirements & Code	17.7	15.1	40.2	16.8	16.6
Requirements, Code & Supporting Files	83.3	21.8	29.1	86.2	17.0
Code	39.8	11.4	10.0	3.8	4.8
Code & Supporting Files	53.4	5.2	30.7	30.1	32.0
Other	9.3	2.5	19.4	135.2	11.4

Table 9.7: Mean effort per failure spent for each fault type and fixed artifact group

requirements on average required considerably less effort per SCR than requirements faults that led to fixes in multiple types of artifacts. For example, 3.5 hrs/SCR for fixes in requirements early versus 17.7 hrs/SCR or more for requirements faults that led to changes in requirements and code.

- **Coding faults that led to changes only in code or only requirements required a relatively low effort to fix** compared to other coding faults, that is, on average less than 10 hours and 7 hours respectively, while other ranged between 20 and 40hrs[9].
- **Some categories required more effort to fix than others.** For example, it can be seen that SCRs due to requirements faults and also SCRs due to integration faults that led to changes in requirements, code and supporting files required a relatively high effort to fix (i.e., over 80 hours per SCR). However, it should be noted that these observations are each based on few SCRs, but were expected based on the skewed distribution that has large value observations with small, but non-negligible probability.

9.6.2 Fix effort for high-priority failures

Fixing high-priority post-release (i.e, on-orbit) and/or safety-critical failures is very important despite the fact that each occurs rarely. To the best of our knowledge whether or not such high priority fixes require more effort to fix has never been explored. Thus, we investigated how the fix effort differs for SCRs reported pre-release (i.e., during development and testing) versus those reported post-release (i.e., on-orbit), as well as for safety-critical versus non-critical SCRs.

Table 9.8 and Table 9.9 show the distribution of fix effort for safety-critical/non-critical failures and pre/post release failures, respectively. Since mean and standard deviation can be sensitive to outliers (i.e., do not represent skewed distributions well) we also considered the median and the Semi Interquartile Range (SIQR) (a measure of dispersion around the median)[10].

[9] Coding faults that led to changes in design, requirements & code showed a normalized effort 10.4 hrs per SCR, but only only one SCR fell into this category. On the other hand, 216 SCRs linked to coding faults led to changes only in code and 27 SCRs linked to coding fault led to changes only in requirements

[10] The semi-interquartile range is a measure of spread or dispersion. It is computed as one half the difference between the 75th percentile (often called (Q3)) and the 25th percentile (Q1). The formula for semi-interquartile range is therefore: (Q3-Q1)/2. Since half the scores in a distribution lie between Q3 and Q1, the semi-interquartile range is 1/2 the distance needed to cover 1/2 the scores. In a symmetric distribution, an interval stretching from one semi-interquartile range below the median to one semi-interquartile above

Verification Activity	# of SCRs	Total Effort	Meaneffort per SCR	Standard Deviation	Median Effort per SCR	SIQR
Pre-release	1,116	15,609.9	14.0	43.9	3	4.5
Post-release	37	1,881.9	50.9	195.6	7	12.5

Table 9.8: Effort spent fixing faults for pre- and post-release failures

Severity	# of SCRs	Total Effort	Mean effort per SCR	Standard Deviation	Median Effort per SCR	SIQR
Safety-critical	122	5,549.4	45.5	120.5	10.5	22.0
Non-critical	650	9,463.1	14.6	49.8	4.0	5.0
Unclassified	381	2,479.1	6.5	13.0	2.0	1.5

Table 9.9: Effort spent fixing faults for safety critical, non-critical and unclassified failures.

Based on 9.8 and Table 9.9 we made the following observations:

- ***Post-release failures tend to require more effort to fix than pre-release failures***. From data given in Table 9.8 it can be seen that post-release failures required on average three times the effort per SCR to fix than pre-release failures. The larger median and larger SIQR support the observation that post-release failures require more effort to fix, with significantly higher variability. We suspect the extras time spent fixing post-release failures is actually a result of the fact that more artifacts existed and analysts may documented high priority failures more carefully.

- ***Safety-critical failure tend to require more effort to fix than non-critical failures.*** From the data given in Table 6 it can be seen that fixing safety–critical failures on average required three times the effort of fixing non–critical failures. Even the median effort per safety-critical SCR was significantly larger and more variable than the median effort to fix non-critical failure. However, we note that this may be due a result of analysts who document safety-critical failures more carefully than non-critical failures because of the importance in preventing them from (re)occuring.

the median will contain 1/2 of the scores. This will not be true for a skewed distribution, however. The semi-interquartile range is little affected by extreme scores, so it is a good measure of spread for skewed distributions. However, it is more subject to sampling fluctuation in normal distributions than is the standard deviation and therefore not often used for data that are approximately normally distributed [21]

# of types artifacts fixed	**# of SCRs**	**Total Effort**	**Mean effort per SCR**	**Median Effort per SCR**
1	830	6,924.3	8.3	2
2	259	6,329.5	24.4	6
3	61	3,427.3	56.2	36.5
4	3	810.8	270.3	10.5

Table 9.10: Fix effort per number of different types of artifacts fixed.

9.6.3 Fix Effort with respect to the spread of the fix

Although, there was no data available detailing the size of the fixes in terms of lines of code added or removed we believe one way to measure the difficulty of fixes may be based on the spread of fixes across components and artifacts fixed. Intuitively, fixes that involve several different types of artifacts and/or multiple components seem more complex and thus, may require more effort. Thus, we quantify fix effort as a function of the number of different types of artifacts fixes and the number of components fixed in Tables 9.10 and 9.11, respectively.

We made the following observations:

- **Fixing multiple types of artifacts required more effort**. As expected, Table 9.10 shows that as the number of types of artifacts increases from 1 to 4 the mean effort per SCR increases significantly; 8.3 hours per SCR when one type of artifact was fixed and almost 56.2 hours per SCR when three types of artifacts were fixed. Similarly, the median effort per SCR also increased with the number of types of artifacts fixed. Note that there were only three SCRs that led to fixing four different types of artifacts and therefore those are not included in the discussion, but are provided in Table 9.10 for completeness.

- **Fixing multiple components required more effort.** Over 16% of SCRs led to fixes in multiple components. From Table 9.11 it can be seen that both the mean effort per SCR and median effort per SCR increase as the number of affected components increases from 1 to 4. Once again, it should be noted that the number of SCRs which led to fixes in four or five components was very small.

# of components fixed	**# of SCRs**	**Total Effort**	**Mean effort per SCR**	**Median Effort per SCR**
1	968	10,730.9	11.1	2.0
2	155	5,035.3	32.5	10.3
3	23	1,406.6	61.2	17.0
4	5	304.0	60.8	53.1
5	2	15.0	7.5	7.5

Table 9.11: Fix effort per number of components fixed.

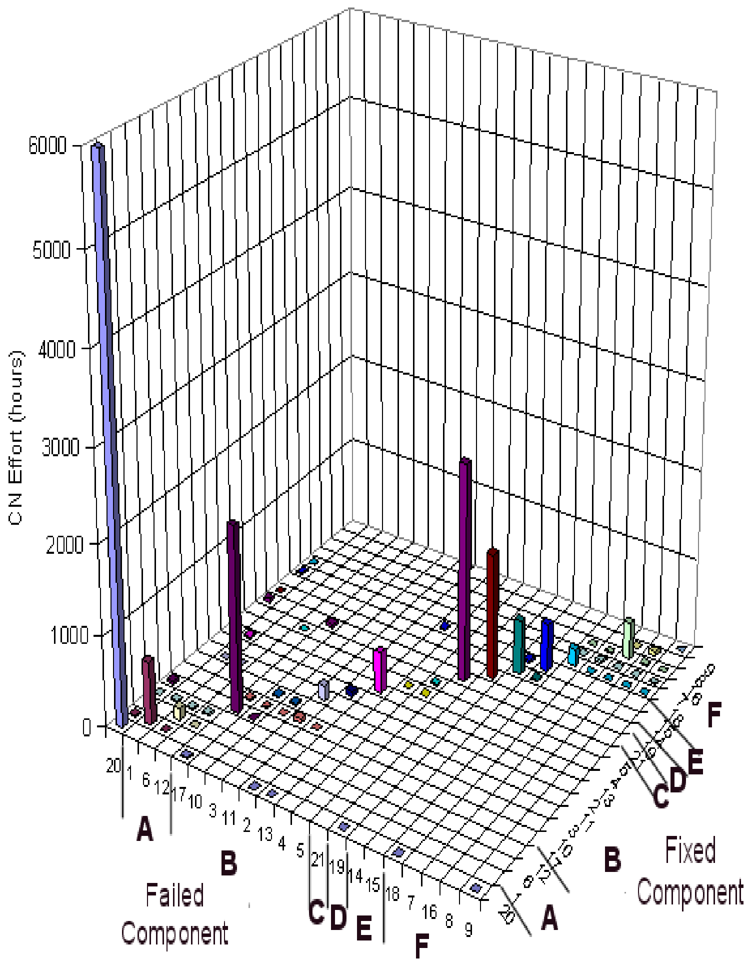

Figure 9.11: Mean fix effort per failed and fixed components

9.6.4 Fix Effort with respect to the Failed-fixed Component Relationship

We investigated how effort was distributed across fixed components in relation to the failed component, as defined in section 9.1.1. The cumulative effort spent fixing each component is shown in Figure 9.11 with respect to the failed component. Note that the components have once again been arranged according to the hierarchical software architecture and that each group (i.e., A though F) represents a group of components that are interconnected to one another and connect to the single top level component (i.e., component 20) through the first component listed in the group. For example, in Group A component 1 connects components 6 and 12 to the top level component (i.e., component 20)).

We made the following observations:

- **Most effort was spent fixing the components the SCR was written against.** As expected the largest cumulative effort values fall on the diagonal. About 12% of the effort spent fixing components was spent fixing components other than the one the SCR was written against (i.e., the total effort off the diagonal in Figure 9.11).

- **A significant portion of the effort was spent on the single top level component.** As expected, we see that the largest component, which also has the most non-conformance SCRs filed against it, also had the most effort spent fixing it. Over 38% of the overal effort spent on fixing all 21 components was spent on fixing component 20 for SCRs written against component 20.

- **More effort was spent on components that connect architectural groups to the top level component.** It can be seen that components 1, 17, 18, 19, 14, 18 tend to stand out among the rest in terms of effort spent. Additionally, fixing components 13, 15 and 16 required a considerable portion of effort.

As mentioned earlier (see section9.1, the majority of SCR lead to changes only within the failed component, but some lead to changes in other components as well, and very few lead only to changes in other components. Thus, we also explored how fix effort per SCR differed in regard to the following groups:

- Group 1 (82%) consists of SCRs that led to fixes only in the failed component (i.e., the component against which the SCR was filed)

- Group 2 (15%) consists of SCRs that led to fixes in the failed component, as well as at least one other component.

- Group 3 (3%) consists of SCRs that did not lead to changes in component the SCR was written against but did require changes in other components.

Table 9.12 shows the number of SCRs, the cumulative fix effort, and the mean and median effort per SCR for each group.

As one would expect, both the mean and median effort were higher in the case when the failed component and at least one more component were fixed than when only the failed component was fixed. Surprisingly, however, when the suspected failed component was not fixed at all (i.e., group 3) both the mean and median effort were the smallest. Note that group 3 has the smallest sample size.

Fix Group	**# of SCRs**	**Total Effort**	**Mean effort per SCR**	**Median Effort per SCR**
Group 1	932	10,615.9	11.4	2.2
Group 2	181	6,717.9	37.1	11.0
Group 3	40	158.0	4.0	2.0

Table 9.12: Fix effort relative to the on spread of fixes across components.

Chapter 10

Prediction

Data mining is an emerging field within software engineering that has recently received quite a bit of attention. Data mining is used to extract patterns from data and involves learning from such patterns to make predictions for future data. Up to this point the analysis and observations were based on the assessment of individual features, pair-wise association between two features, and rarely the interaction of three features (e.g., effort per fault types for each group of fixed artifacts). In this chapter, we explore the possibility of using features based on the information contained in an SCR to predict features of associated fixes. Specifically, we explore the following research questions:

RQ5: Can traditional machine learning algorithms be used to predict features of fixes based on fault and failure features entered in a non-conformance SCR?

A. Can we predict the types of artifacts that will need to be fixed?

B. Can we predict the components that will need to be fixed?

C. Can we predict the amount of effort required to implement the fix?

D. Can high priority failures classes be characterized by common features?

It is important to note, that we are not predicting typical fault or failure proneness as most studies do (e.g., see [5] and references therein). Rather than predicting modules or components that are fault prone, we focus on predicting characteristics of the fix after a failure has occurred or fault was detected. The ability to predict the components to be fixed and/or the artifacts to be fixed could help developers locate faults and implement fixes faster, as well as help determine if fixes are complete (e.g., all necessary artifacts were fixed, and all necessary components were fixed). The ability to predict the effort required to implement fixes would allow managers to schedule resources more efficiently and effectively as well as help managers identify areas for improvement. For example, it may be possible to guide development activities and/or verification activities towards methods focused on preventing faults that required a significant amount of effort to fix.

This chapter proceeds by discussing the input features and response variables, and then provides some background information for major data mining schemes and discusses the applicability of each to our data and research questions. Finally, we present the results of the data mining.

10.1 Features

The features describing and characterizing faults and failures (discussed in chapters 6 through 9) account for the majority of the input variables used. Based on the observation made in Section 9.1.1 that fixes tend to remain within architectural groups we included two additional input features representing the component's location within the software architecture. Specifically, we used the following features as input variables:

- **Fault Type** - the type of fault that caused the failure.
- **Verification Activity** - the verification activity taking place when the fault was detected or failure was exposed.
- **Severity** - the impact of the failure.
- **Failed Component** - the component the non-conformance was reported against.
- **Release** - the software release of the failed component.
- **Investigation Effort** - the hours spent to report the non-conformance.
- **Architectural Group** - the architectural group to which to failed component belongs.
- **Architectural Tier** - the level within the three tier hierarchical architecture to which the failed component belongs.

It should be noted that almost all of the input features are categorical, with the exception of severity (which is ordinal) and investigation effort (which is numeric). Additionally, we point out that the architectural metrics are specific to the hierarchical structure and groupings of components within the mission and thus may not be applicable to other architectures.

The following features characterizing fixes were explored as response variables:

- **Fixed Artifacts** - the types of artifacts affected by the fixes implemented to address the non-conformance SCR.
- **Fixed Architectural Group** - the architectural groups affected by fixes implemented to address the non-conformance SCR.
- **Fix Effort** - the total effort to implement the fixes described in all CNs associated with an individual SCR. Although, fix effort is numeric attribute measured in hours we choose to discretize the values in an attempt to predict whether high, medium or low effort would be required rather than trying to predict the exact number of hours. Predicting the exact number of hours needed is likely to be less accurate, especially when considering the fact that almost all the input features were categorical. The numerical values were discretized into three classes: high, medium, and low based a typical logarithmic transformation. To be exact, the values were transformed by taking the log of each value and then dividing the range of the logs by three (i.e., to map to the three class values), which is a standard transformation for converting numerical effort data into categorical data used in effort prediction [14]. Thus, effort values less than 1.44 hours were consider low effort, those between 1.45 hours and 41.44 hours were considered as medium effort and those with higher values were considered high effort. This resulted in 339 low effort instances, 661 medium effort instances and 93 high effort instances.

10.2 Background on Data Mining

Typically data mining methods follow one of four different learning styles:

- **Classification learning** - takes a set of classified examples and attempts to learn a way to classify unseen examples based on the association between input features and the classification values. For example, classification learning could be used to predict whether a module is fault (or failure) prone based on the static code features which have been associated with fault (or failure) prone modules based on the pre-classified examples.

- **Numeric Prediction** - is a variant of classification learning where the output is numeric. It attempts to predict a value of a given continuous valued variable using a mathematical model based on the input features.
- **Association learning** - searches for relationships between attributes without considering a specified classification. For example, an online shopping site might gather data on customer purchasing habits using association rule learning.
- **Clustering** - is like classification but the class value for each instance is unknown. Thus, the algorithm will try to group similar items together based on a defined distance measure for each attribute. For example, clustering could be used to group modules based on common static code features without knowing whether the modules are fault (or failure) prone.

The fact that the vast majority of features in our study are categorical limited the methods applicable. Clustering methods were not applicable because it was not possible to define meaningful distance metrics (which are needed to determine the similarity (or difference) between the categorical values) for the vast majority of input variables. Additionally, numerical prediction techniques were not used, because the likelihood of accurately predicting numeric values based on mainly categorical input features is very low. Thus, we explored classification and association rule learning, which are described next. For more information, on the tools and data mining methods the reader is referred to [66], which is the main reference used throughout this chapter.

10.2.1 Background on Classification Rule Learning

Classification is a type of supervised learning that can be used on numerical or categorical data. The fact that it is supervised implies that output (or class) variable is known for each instance in the training data. Ideally, the learner identifies patterns in input values of the training data that can be used to predict the class output value on test data. Two commonly used types of classification algorithms are decision tree learners and probabilistic learners; both are state-of-the-art and tend to be computationally fast, thus we explored the applicability of each. Decision tree learners produce human readable output in the form of classification rules. These rules describe trends in the input data in relation to the class output variable. Basically, the nodes of the tree are input features and the path (or branch) followed from each node depends on the value of that feature. The leaf nodes identify the class values. Rules are built for each path from root to leaf. To avoid over-fitting, branches (and rules) can be pruned.

J48 is a decision trees algorithm implemented in WEKA [76] and is commonly used for many classification problems. J48 builds decision trees from the training data using the concept of information entropy. It uses the fact that each input feature of the data can be used to split the data into smaller subsets. J48 examines the normalized information gain (i.e., the difference in entropy) to choose which input feature to split on. Then the algorithm recurs on the smaller subsets and the splitting procedure stops if all instances in a subset belong to the same class or the minimum support value is reached, which results in the creation of a leaf defining the class value. It may happen that none of the features give any information gain. In that case J48 creates a decision node higher up in the tree using the expected value of the class. J48 can handle both continuous and discrete (i.e,. categorical) features and provides an option for pruning trees after creation. For further information, we refer the reader to the original publications [59].

Probabilistic learners calculate the posterior probability of each class value given the input values. One of the most commonly used probabilistic learner is NaiveBayes, which uses Bayes Formula to calculate posterior probabilities and then assigns an input instance the class value with the highest probability. It should be noted the Nave Bayes assumes independence among input features, which is not the case for our input features. However, despite the unrealistic assumptions Nave Bayes has been shown to work quite well in practice (e.g. [27]). Thus, we decided to include the Nave Bayes classifier in the experiments [36].

To evaluate the prediction abilities of the J48 and NaiveBayes learners we ran typical 10 by 10 cross validation experiments. For the 10 by 10 cross validation the data was divided into 10 partitions and the learner was trained on all but one partition and then tested for accuracy on the remaining partition. This process was repeated 10 times; each time a different partition was used to evaluate the learner. We use the ZeroR learner (which always predicts for the majority class) as a baseline to evaluate the performance of both J48 and NB in the 10 by 10 cross validation experiments. Clearly, to be useful J48 and NB should do

		Predicted Value	
		P'	N'
Actual	P	TP	FN
Value	N	FP	TN

Table 10.1: Actual versus predicted outcomes

a better job predicting class values then the learner that always predicts the majority class.

To compare the performance of the two learners, in other words to determine which learner performed better (e.g., J48 or NaiveBayes) receiver operator curves (ROC curves) are commonly used. ROC curves plot the false positive rate (FPR) on the horizontal axis and the true positive rate (TPR) on the vertical access. In a binary class problem the FPR is the ratio of false positives to the total number of negatives instances and true positive rate is ratio of the true positives to the total number of positive instances. Table 10.1 shows the how true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) are defined based on the actual classification (rows) and predicted classification (columns). Thus, $FPR = FP/(FP + TN)$ and $TPR = TP/(TP + FN)$.

Although, ROC are very powerful for measuring accuracy for binary class problems their usefulness for multi-class problems is not as straightforward. In [49] and [38] it was shown that three dimensional ROC curves could be useful for comparing tri-class problems by calculating the volume under the curves, but applying this idea to larger class problems increases the complexity quickly. To avoid this, it is possible to build two dimensional ROC curves for each value of each class explored. However, this method can also become quite tedious and difficult to evaluate as the number of class values increases beyond three or four values. Thus, as suggested in [8], we also use Cohen's kappa statistic to evaluate learner performance on multi-class problems. The authors showed that there is a close relationship between kappa and ROC Curves and suggested that kappa expresses important properties of a point in the ROC curve rather than describing the curve in full. The kappa statistic, K, is calculated as follows:

$K = \frac{(P_0 - P_c)}{(1 - P_c)}$

where P_0 is the total agreement probability (i.e., the accuracy) and P_c is the agreement probability due to chance, that is,

$P_c = \sum_{(i=1)}^{I} P(x_{i.}) - P(x_{.i})$

where i is the number of class values and $P(x_{i.})$ and $P(x_{.i})$, are the column and row marginal probabilities, respectively. Kappa ranges from -1 to 1, with values less than 0 representing no agreement between the actual and expected, and values closer to 1 representing close agreement. Thus, learners with Kappa values close to 1 show better performance. Classification learning can be used to explore research questions RQ5–A, RQ5–B, and RQ5–C, which explore the possibility to predict the artifacts fixed, components fixed and fix effort, respectively.

10.2.2 Background on Association Rule Learning

Association rule learning is used to identify interesting relationships between features in a data set. Unlike classification learning association rule learning is not concerned with predicting a class or output variable, rather the purpose is to identify features from different attributes that tend to occur together. Thus, although association rule learning could not be used to help us explore research questions RQ5–A through RQ5–C, which focus on predicting a class value, we used them to look for associations amongst features that our earlier assessment may have missed. In other words, we explored research question RQ5–D, which was aimed at characterizing failures that required more effort to fix faults.

We used the well-known Apriori algorithm to mine association rules [76]. The Apriori algorithm works by generating rules based on statistics of item co-occurrence. Co-occurrence refers to instances where two or more items appear in the same context, and is referred to as 1st-order association. It should be noted that association learning based on higher-order associations is not consider in this dissertation and may lead to more interesting results. For more information on the Aprior algorithm used the reader is referred to [4].

10.3 Prediction Results

In this section we present the results for both classification and association rule learning based on experiments ran in WEKA.

10.3.1 Classification Rule Learning Results

We conducted classification learning for each response variable (i.e., artifacts fixed, architectural groups fixed and fix effort). Using classification learners we addressed research questions RQ5A through RQ5C, as follows:

	ZeroR		**J48**		**NaiveBayes**	
Fixed Artifact Group	25.50%	0	69.64%	0.57	67.91%	0.42
Fixed Architectural Group	42.32%	0	86.73%	0.84	85.68%	0.83
Fix Effort	57.33%	0	73.20%	0.47	66.87%	0.39

Table 10.2: Accuracy and Kappa values for the fixed artifact group, the fixed architectural group, and the fix effort

Fixed Artifact Group	ZeroR	J48	NaiveBayes
Requirements	1	0.85	0.88
Requirement, Design, & Code	0	0	0
Requirements & Code	0	0.65	0.34
Requirements, Code, & Supporting Files	0	0.40	0.53
Code	0	0.76	.76
Code & Supporting Files	0	0.22	0.42
Other	0	0.09	0.10
Weighted Average	0.42	0.70	0.68

Table 10.3: Probability of detection for each fixed artifact group

- RQ5–A: Can we predict the types of artifacts that will need to be fixed?
- RQ5–B: Can we predict the components that will need to be fixed?
- RQ5–C: Can we predict the amount of effort required to implement the fix?

In each case, we used the values defined in chapter 9 for class values of response variables. Table 10.2 shows the accuracy and kappa statistic for each learner (i.e, ZeroR (the baseline), J48, NB) for the three response variables (i.e., types of artifacts fixed, architectural group fixed, and fix effort). It should be noted, that each prediction model considered only one class variable at a time, the others were removed from the data set. The associated recall values (i.e., probability of detection, PD) are provided for each value of each class for the three learners in Tables 10.3, 10.4, and 10.5. ROC curves are not provided for the artifacts fixed and architectural group fixed because of the larger number of class values, but are shown in Figure 10.1 for high, medium and low effort.

Based on Tables 10.2 through 10.5 we made the following observations:

- **The group of artifacts to be fixed can be predicted.** As shown in Table 10.2, both J48 and NB outperformed ZeroR in terms of accuracy on 10 by 10 cross validation. Thus, since ZeroR always predicts for the majority class without considering the input features, suggests that the input features used (i.e., fault type, verification activity, architectural group, architectural tier, severity, and fix effort) were useful in

Fixed Architectural Group	ZeroR	J48	NaiveBayes
A-	1	0.966	0.969
A-B-	0	0	0
A-B-C-	0	0	0
A-B-other	0	0	0
A-C-	0	0	0
A-C-F-other	0	0	0
A-C-other	0	0	0
A-D-other	0	0	0
A-E-	0	0	0
A-E-G-	0	0	0
A-E-other	0	0	0
A-G-	0	0	0
A-G-other	0	0	0
A-other	0	0.154	0.096
B-	0	0.946	0.966
B-C-	0	0	0
B-other	0	0.906	0.625
C-	0	0.966	0.971
C-D-	0	0	0
C-F-other	0	0	0
C-other	0	0	0
D-	0	0.958	0.958
D-other	0	1	0.5
E-	0	0.959	0.959
E-other	0	0.176	0.176
F-	0	1	0.976
F-other	0	0.143	0
G-	0	0.985	0.99
G-other	0	0	0
other	0	0	0
Weighted Average	0.255	0.867	0.857

Table 10.4: Probability of detection for each fixed architectural group

Fixed Artifact Group	ZeroR	J48	NaiveBayes
High	0	0.33	0.36
Medium	1	0.88	0.71
Low	0	0.59	0.68
Weighted Average	0.57	0.73	0.67

Table 10.5: Probability of detection for fix effort

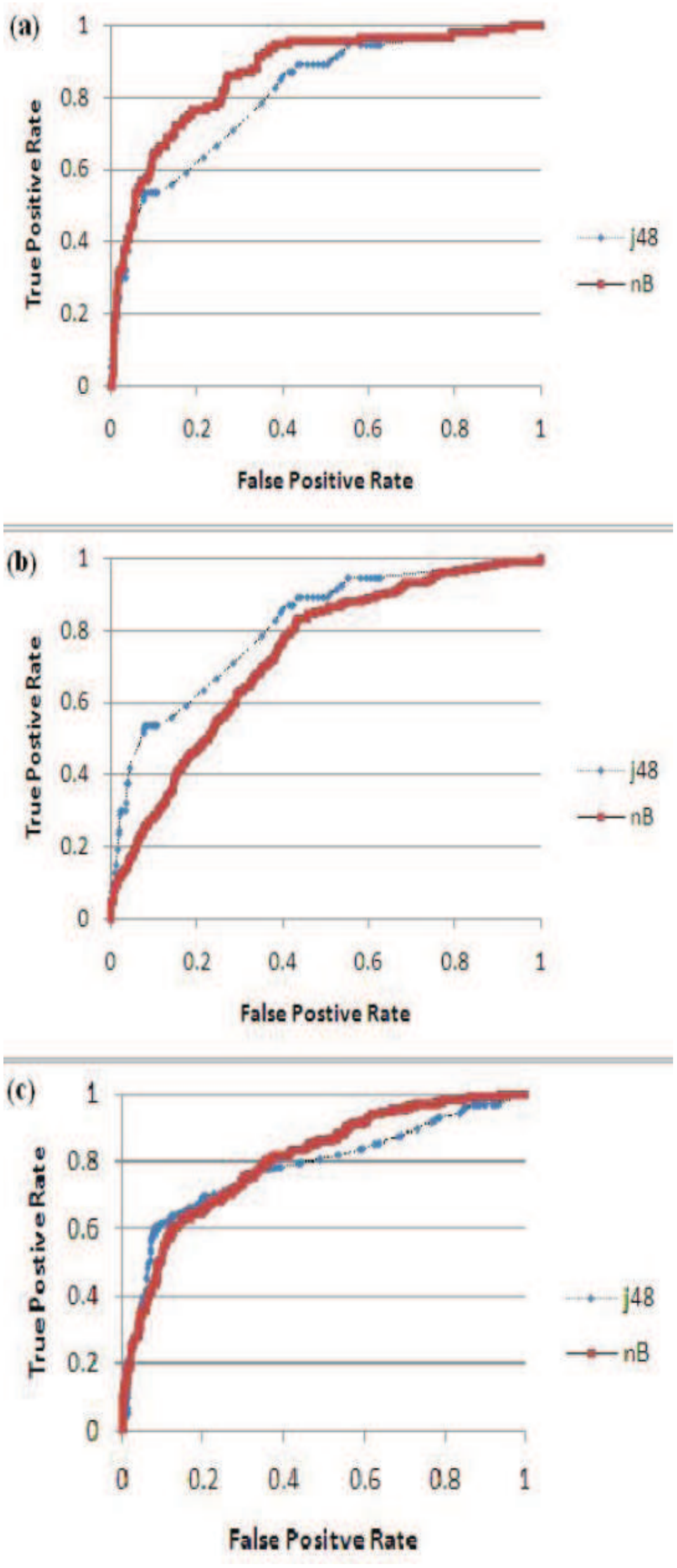

Figure 10.1: ROC Curves for (a) high fix effort, (b) medium fix effort and (c) low fix effort

helping to predict the set of artifacts to be fixed. As suggested by the high recall (PD) values for requirements, code, and code & requirements in Table 10.3 we suspected the association identified earlier in section 9.3 between fault type and artifacts fixed is heavily influential on the results. Thus, we re-ran the 10 by 10 cross validation experiments for J48 and NB with fault type as the only input feature, which resulted in accuracy of 82% with $K = 0.74$ for J48 and an accuracy of 80% and $K = 0.72$ for NB, which suggests that to predict the artifacts fixed just the fault type should be used and the additional input features create noise.

- **The architectural group fixed can be predicted.** Similarly to the case of artifact fixed, both J48 and NB outperformed ZeroR. Once again as suggested by the high recall (PD) values for single architectural groups (i.e, A, B, C, D, E, and F) we believe the success was based on a relationship identified earlier, that is, the relationship in section 9.1.1 between the failed and fixed components. However, re-running J48 and NB with only the failed component as an input feature resulted in slightly lower accuracy values and kappa values, that is, 84% and $K = 0.81$ for J48 and 84% and $K = 0.82$ for NB. This suggests that the failed component is a very good indication of the fixed architectural group, but we note that considering additional input features led to slightly better results. Thus, if the information characterizing the fault type, verification activity, severity, and report effort are readily available each should be used in the prediction model. However, if such information is difficult to obtain the effort spent to do so is likely to outweigh the gain of including such data.

- **The relative level of fix effort can be predicted.** Once again, J48 and NB both outperformed ZeroR. However, in this case we could not predict effort with similar accuracy based only one input feature. Thus, it appears the data mining techniques are most applicable here, which is why we included Figure 10.1, using a ROC curve to compare J48 and NB, which shows that when predicting for the high effort SCRs (which would be most useful in terms of allocating resources and planning purposes) NB should be used as it consistently outperformed J48.

In addition to the manual feature selection (i.e., using only the fault type to predict the fixed artifacts, and using the architectural group of the failed component to predict the architectural groups fixed) we explored various feature subset algorithms available in the WEKA. However, the performance of each learner decreased significantly, which is likely due to the relatively small number of features. The only exception was with respect to predicting the artifacts to be fixed. Using the CFS best first feature subset selection algorithm we found that using fault type, investigation effort and severity as the only input features resulted in an accuracy on of 71% and $K = 0.59$ with very little change in the recall (PD) for each class value. We suspect that fact the severity contributes to better predictions is a result of the fact that high severity failures could be very likely to be better documented, thus a greater number of different types of artifacts were fixed. Thus, we conclude that if machine learning techniques are to be employed all available features should be used.

10.4 Association Rule Learning Results

We implemented the Aprior associate learner in WEKA to explore research question:

RQ5–D: Can high priority failures classes be characterized by common features?

We ran the Apriori association learner on the high priority failures class subsets and their counterparts, that is, safety-critical failures, non-critical failures, and unclassified failures; and pre- and post-release (i.e., on-orbit) failures. Additionally, in order to help characterize failures associated with high fix effort and/or failures that lead to fixes spread across component we also ran the learner on the following pre-defined subsets of SCRs: high fix effort SCRs, medium fix effort SCRs and low fix effort SCRs; and failures that led to fixes outsides the failed component and those that did not require fixes outside the failed component.

In each case the learner discovered rules with high support values. However, through detailed examination of the rules it became apparent that we had already discovered the most informative relationships through the pairwise correlation conducted in chapter 9. For example, the correlation between fault type and the types of artifacts fixed led to multiple rules with fault type on one side of the rule and the type of artifact fixed on the other. Although, the rules often included an additional feature or two the support values did not improve. Thus, it was clear the the underlying associations were the driving force for the rules learned. We tried removing strongly correlated attributes (e.g., we would remove either the architectural group fixed or the architectural group input feature) to see if more interesting rules would be learned. However, as expected the support values dropped quickly. In addition, similarly to [63], we mined the rules learned to consider only those that contained features of interest (e.g., high fix effort) on the right side of the rules. However, once again the support values were extremely low. Thus, we conclude that although the association rule learning did not add new observations it did serve to verify the statistical hypothesis tested in chapter 9.

10.5 Discussion

The classification rules learning showed promising results in terms of predicting the type(s) of artifacts fixed, the architectural group(s) fixed and the fix effort. In fact, the accuracy and recall values for our prediction models were comparative to what others have deemed successful in the past for fault and failure proneness predictions (for examples see [5] and references there in).

The comparison of our results with other works that made predictions for fixes is more difficult because the evaluation criteria used differed across studies. The work presented in [16] and [17] predicted the files to be changed and the developer best suited to make the changes using a ranking scheme and reported accuracy values in [16] and recall (PD) values in [17] to evaluate the model. Our accuracy values were comparable or better than those in [16] for all but one of the systems explored. Further, the weighted average recall values for each of our class prediction variables (i.e., types of artifacts fixed, architectural group fixed and fix effort) were far better than those reported across all predictions in [17] and comparable when only the first N (i.e, 5,10,100) retrieval items were considered. We suspect the difference in results is due to the fact that we consider all change requests that represent failures while [16] and [17] consider all types of changes requests, including those made for enhancements etc.

In [24], an order response model was used to predict the effort category (1 of 4) for each fault correction effort. Based on Kendall's Tau and concordanant/discordant measures the authors claimed that the results were 'very favorable' and the model could be reasonably applied to other systems. The work in [72], which also predicted the fix effort used the magnitude of relative error (MRE) to evaluate the results. Although the MRE was on average $7-23\%$ for the original study it was considerably worse when the model was applied to an additional study (i.e., $40-159\%$). Text similarity was used in [65] to help predict fix effort and the average absolute residual (AAR) was used to evaluate the model. The authors note that the original results were poor, but applying thresholds to the nearest neighbors approach led to better prediction. However, using their ideal threshold the only actually made predictions for 13% of the original data set.

Even though the comparison with related studies suggested our models did well it should be noted that with respect to the types of artifacts fixed and the architectural groups fixed similar results can be obtained using the input feature with the highest correlation value, that is, fault type and failed component, respectively. With respect to the fix effort, we believe that project managers could easily benefit from using simple data mining techniques to predict SCRs the will required a high fix effort allowing him or her to adjust resources and/or schedule as needed.

We believe these results could be very complementary to the recent research focused on defining hierarchical aerospace ontologies of concepts and nomenclature to identify problem type and/or equipment type tags [47]. Combining the two features consider in both work could significantly improve prediction capabilities.

In [63] association rules were shown to be more effect for prediction the isolation effort, however as noted in the paper the association learning was not straight forward as association rule learning does not predict a specific class variable. Thus the authors have to review all rules generated and select those that predicted for effort. We also tried this approach but our rules did not generate reliable effort predictions

The association rule learning showed less interesting results as although multiple features appeared in the rules it was clear that the pair-wise correlations discovered earlier was heavily influential. Thus, it appears that association rule learning may not be useful to the developers and project managers at this point.

Chapter 11

Threats to validity

The validity of each of the conclusions reached is influenced by the design and implementation of the study. Thus, in this section we discuss the threats to the validity of our conclusions. First, in section 11.1 we discuss internal validity and construct validity per feature analyzed to document how internal factors influenced the study and ensure we explored (and tested) what we set out to. Additionally in section 11.2, we discuss the conclusion (or statistical) validity which considers the methods used to evaluate what was tested and determine if the conclusions are justified. Finally, in section 11.3 we discuss the external validity which considers the generality of the results.

11.1 Internal and Construct Validity

In order to characterize faults, failures and fixes in a way that would be meaningful to the mission as well as the larger software engineering community, we had to (1) choose appropriate features to analyze and (2) define the categories of values for each features. The study began with a detailed analysis of data fields in the change tracking database. Fields from SCRs and CNs were identified based on the usefulness of the information contained in the field and the quality of the data. Based on the fact that SCRs and CNs were filled out by analysts who worked on different parts of the mission at different locations using various methods, the recorded data varied in terms of quality and detail. Thus, we continually interacted with project personnel who had the domain expertise to asses the quality of recorded values and clarify any uncertainty around the meaning.

The analysis was conducted at the component level since change requests were filed at that level. Although it is possible that some characteristics that may exist at finer levels of granularity (e.g. file level) were masked at the component we do not believe this invalidates our results as we clearly identify the level of granularity used. On the other hand, we note that the observations with respect to the architectural properties and spread of fixes may not have been observed at a lower level.

The assessment was based on seven data fields from the non-conformance SCRs and associated CNs (i.e., 'source', 'discovered by', 'severity', 'SCR effort', 'affected products', 'CN component' and 'CN effort'). Based on rigorous exploration of each of these fields, including detailed discussion with the project personnel, we mapped these fields directly to seven features describing faults, failures and fixes, that is, fault type, verification activity, severity, investigation effort, artifacts fixed, components fixed and fix effort, respectively. Obviously the interpretation of the data values used in each field could influence the results based on the groupings of feature values. Therefore, we took the time to ensure that we thoroughly understood the data and we were not misinterpreting any values (which was not a simple task since some formal definitions of field values were missing) and had project personnel review groupings. The lack of formal definition of data fields and values was likely due to the fact the change tracking system was designed many years ago and it was not designed to facilitate such detailed analysis. The characteristics of each field differed, from pre-defined mandatory values to free-text fields that were optional. Thus, we discuss each data field and the associated values.

Fault types were based on the 'source' field and the values were selected from a predefined list. The meanings of values were self-explanatory. Thus, we were able to easily combine the 13 values into 5 major fault types for statistical testing.

The categories of the verification activity (i.e., analysis, testing, inspections and audits, on-orbit and others) based on the free-text 'discovered by' field were more difficult to assign because there were over 100 unique values, which contained numerous abbreviations. Thus, we identified the unique values and common acronyms and discussed each with project team members to determine the major activities. It should be noted that the 'analysis' category (which was the verification activity identified for 49% of SCRs) included only those SCRs that were specifically tagged 'analysis' in the 'discovered by' field. Although, 'analysis' was one of the field values that lacked a formal definition, by manually exploring multiple analysis SCRs and reviewing NASA Standards and Guidebooks we found analysis to include various activities which involve analyzing different software artifacts (e.g., informal review or walkthroughs), which differ from more formal inspections (or audits) that required specified steps to be taken, and roles to be assigned to individual reviewers [52]. It should be noted that SCRs created based on non-conformance observed during testing may have also been revealed through some type of analysis (e.g. code analysis) if the resources were available to perform such analysis. Similarly to other studies (e.g., [1], [26], [43], and [45]), we could not measure the amount of relative effort spent on each activity based on the available data, thus we do not make claims that one activity performs better than another.

We handled the severity attribute with special care since it often constitutes a major threat to validity based on the fact that severity can often by artificially inflated to call attention to the problem reported. In fact, based on suggestions from project personnel we explored the severity levels assigned to changes notices (CNs) in addition to the severities assigned to the SCRs. This allowed us to verify and update previously classified severity values. Despite the fact that the documents were typically filed out by two different analysts we observed very few differences between severity levels originally associated with the SCRs and severity levels assigned to CNs. In the cases when they differed, based on the recommendation made by the project analysts, we used the value associated with the change notices, which was considered to be more accurate. Considering this additional severity field in CNs allowed us to classify 78 previously unclassified SCRs.

The analysis of the component(s) fixed was based on the component field in the CNs. Although this field is not based on a pre-defined list, the values were easily associated with components. The fact that in some cases multiple CNs were linked to a single SCR encouraged us to explore the relationship between the failed component (i.e., the component against which the non-conformance was written) and the components that were fixed, which we defined as the failed - fixed component(s) relationship. It should be noted that in a small number of cases (i.e., only about 3% of SCRs) the 'failed' component was never actually fixed, indicating that the component suspected to fail may not have been the actual cause of the non-conformance to a requirement reported in the SCR, but rather just a catalysts for the problem to surface. Based on the fact that there is often more than one way to implement a fix and sometimes the quickest and easiest way to fix a problem is through a different component we did not exclude these SCRs from the analysis as we expect this trend may continue.

With respect to the artifact fixed, classifying values into categories was more difficult, because similarly to the verification activity, the 'affected product' field was free text, with varying levels of detail from one SCR to the next. For example, in some cases a specific requirement document or filename was given, but in other cases only a broad high-level reference (e.g. 'code') was provided. The affected product fields contained over 100 unique entries. Once again by manual exploration of the values and through detailed discussions with the project personnel, we defined six categories representing the types of artifacts fixed (i.e., requirements documents, design documents, code files, supporting files, tools, and notes/waivers) and mapped each unique value to one of the categories.

Lastly, with respect to the effort data (i.e., investigation effort and fix effort) we consider effort with respect to fault types, verification activity, severity, artifacts fixed and component(s) fixed, which showed that distribution of effort per feature value differed compared to then the distribution of the number of failures across each value.

In any case, we discussed the perceived meaning of values with multiple software engineers to ensure that we were not misclassifying or misinterpreting fault types or activities. Further, the software review board for the NASA mission, which is responsible for determining which SCRs need to be addressed and for appointing an analyst(s) to handle individual SCRs, follows a well defined process with procedures in place for tracking and maintaining SCRs. Since considering features of fixes in relation to fault and failure features is a unique part of our approach it is important to emphasize that the mission has a software review board in place to help ensure SCRs are documented correctly and that the proposed solution for an SCR is appropriate, which means the changes suggested will indeed address the problem reported. Thus, the data values and our method for linking from failures and faults to fixes are sound based on the explicit information contained within the change documents and the well-defined relationship between them.

To limit the threats to the internal validity we called upon the domain expertise of project personal who helped mitigate the threats discussed. We discussed our perceived meaning of values with multiple software engineers to ensure that we were not misclassifying or misinterpreting, or overlooking any pertinent data. Their input and feedback on our interpretation of the results was incorporated throughout the course of our work. For example, based on the fact that less than 2% of artifacts fixed contained references to notes and/or waivers, we originally believed workarounds were very rare. However, based on suggestions from the project personnel we explored additional type of change document, which showed workarounds were used more commonly. Our experience shows that independent researchers who may be lacking product specific domain knowledge can manage to work with real, large-scale case studies by working closely with the project personnel to ensure the data quality and verify and validate the interpretation of the results.

11.2 Conclusion Validity

The methods used to measure the extent of correlation between features were carefully chosen. In each case, we chose an appropriate statistical test based on the measurement scales of the features and the validity of the underling assumptions of the tests. Additionally, all formal statistical tests are based on random samples. The groupings for attribute values (e.g., fault types, activities, severities, and affected artifacts) used in the statistical tests were carefully chosen not only to support the internal validity (i.e, to accurately represent the data), but also to allow future related studies to compare their findings to our results in a meaningful and practically useful way. Although for each pair of features explored (i.e., verification activity and fault type, severity and fault type, verification activity and severity, artifacts fixed and verification activity, and artifacts fixed and severity) we found statistically significant associations, it should be noted that this does not necessarily imply a cause–effect relationship.

The methods used to build prediction models were chosen based on the questions we wanted to answer (i.e., RQ5 in Chapter 10), as well as the data values. Thus, we chose learners that worked on nominal features and were applicable for making predictions for multi-valued classes. We recognized that additional learners, parameter refinement and other techniques may improve the prediction results and reserve this for future work.

It should be noted that by the nature of some of the features we study some of the samples are very small (e.g., post-release or safety -critical failures). Specifically, only 3% of the non-conformance SCRs were reported on-orbit and about 10% were safety-critical. In other words, comparisons of pre- and post-release, and safety-critical and non-critical failures were made on vastly different size samples; however, these are intrinsic characteristic of any high-quality software. Similarly, with respect to the analysis within individual releases and across multiple releases (i.e., chapter 8) the fact that the number of components explored per release was small (i.e., eleven components per release for earlier releases and three or four components for later releases) presents a threat to validity. For clarity, throughout the text we noted the percentage within each class, as well as within the entire sample to avoid misrepresenting the data.

Additionally, it should be noted that some SCRs were still 'open' during time of this analysis and as with any data set not all 'closed' instances contained entries for all data fields studied. To eliminate the threat to validity due to missing data, lack of data or incomplete data (i.e. SCRs that are yet to be fully fixed)in some section we consider subsets of SCRs. However, whenever subsets were used we checked to ensure that the major trends reported in the entire population still held true, and they did.

Some patterns in the data may be due to the processes used by the mission, the uniqueness of the mission and constraints placed upon the project that cannot be accounted for in our research. For example, our exploration of the relationship between the type of artifacts fixed and fault/failure features led to the discovery of interesting sets of non-conformance SCRs, such as SCRs that had the potential to be avoided or more efficiently detected by compiling a list of common coding faults or SCRs that theoretically could have been reported earlier. However, assuming that all requirement faults could have been discovered in the requirements phase is somewhat unrealistic in case of NASA systems which are often developed to do something for the first time which mean requirements inevitably will be discovered later in the life cycle, during testing and/or in-field operation. Thus, in the next section we discuss the generality of the results with respect to other studies.

11.3 External Validity

In many cases, the use of inconsistent (or not sufficiently precise) terminology associated with software reliability and quality assurance introduces problems and makes meaningful cross-study comparisons difficult. For this reason, we provided the definitions of the fundamental terms used throughout our work and spent the time to carefully interpret the results of others.

In what follows we work through the most interesting results from each chapter to compare and contrast related results.

In chapter 6 we explored the distribution of fault types, verifiation activities and severity levels across failures. One of the most interesting results was that our work as well as other studies (e.g., [7], [13], [23], [19], [22], [45], [43], [68]) clearly showed that some fault types are more common than others. However, the earlier studies (i.e., [7], [13], [23]) found that early life cycle faults (i.e., requirements and design faults) were more common, where as our study was in agreement with the results of more recent studies (i.e., [19], [22], [45], [43],

[68]) which showed later life cycle faults (i.e., coding and implementation faults) are at least as common if not more common than early life cycle faults. We suspect the difference is likely due to process improvements over the years with respect to defining requirements and detailing design, as well as the increased size and complexity of systems. Further, we found that the percent of post-release failures as well as the percent of safety-critical failures was similar to that of others e.g., [26], [1], [43].

The analysis conducted in chapter 7 explored the pairwise associations between fault type, verification activity and severity. Similarly to [43] we found that the common fault types persisted from pre to post-release. Additionally, we found that coding faults were the most common causes of operational failures and safety-critical failures. To the best of our knowledge no other works consider such a relation, which is also true with respect to the verification activities associated with each fault types and with different severity levels.

In Chapter 8 we explored how trends change across and within releases. Once again our results agreed with some studies, and disagreed with others. Specifically, similarly to [11], [1] we found evidence suggesting that there was a positive correlation between pre– and post–release faults, while others [26], [54] suggested the opposite result, that is, modules with the most pre–release faults are not the most likely place for post–release faults. Our results with respect to fault (or failure) persistence across releases differed from the results in [11], [54] and [58] which found that faulty files persisted across releases. We believe the reason for inconsistent results may be due to the various levels the studies were conducted at (e.g., file level, module level, and component level) and the systems studied as well as the development processes followed. However, the studies did not contain enough detailed information for us to explain such differences with certainty. Once again it is obvious additional research is needed.

Although, the related studies discussed in Chapter 3 did not specifically focus on characterizing fixes as we did in Chapter 9, many of the works showed support for some of the observations we made. In particular, with respect to the fact that some failures map to multiple faults, [45], [43], [1], [26] and [24] each made statements suggesting they too observed instances where individual failures were linked to multiple faults. Further, similarly to our results [24] showed that when related faults were spread across components more effort was required to implement fixes. To the best of our knowledge, no other work has related the spread of fixes to the architecture.

The observations made with respect to predictions in chapter 10 are unique in that no other study considered types of artifacts fixed or the architectural groups fixed, and input features used across studies varied. However, [16], [17], [47], [62], [24], [72], [65], [63], and [48] all showed some promising results with respect to predicting features of fixes; from the files fixed to the developers best suited to implement the fix. The variability of types of features studied and modeling techniques used make cross study comparison quite difficult. Further, the most meaningful way to compare learners for multi-class problems is still an open research question. Clearly, more studies conducted on different systems using data and methods that facilitate comparisons are necessary in order for the community to learn from empirical fault and failure data, so that theories can be tested to determine if and when observed phenomena hold true and then refined accordingly.

Chapter 12

Conclusion

In this dissertation we analyzed features of faults, failures and fixes, including fault types, verification activities, severity levels, investigation effort, artifacts fixed, components fixed and the effort required to implement fixes for a large industrial case study. The analysis included data from 21 large-scale software components developed at multiple locations on different release schedules. We used descriptive statistics, statistical inference through formal hypothesis testing, and data mining techniques to analyze the data.

To the best of our knowledge, establishing the complete link from the faults that cause a failure (potential or observed) to the fixes implemented to prevent the failures from (re)occurring, which was not a simple task, is a unique characteristic of our work. First of all, locating and gaining access to empirical fault and failure data can be quite difficult, as many organizations are reluctant to make fault and failure data available for research. Additionally, collecting and analyzing the data is very time consuming, and often requires domain expertise. Even more, associating failures with the fault(s) that caused them, as well as the fixes made to prevent them from (re)occurring is not straightforward as the information is stored in different types of change document and multiple changes were associated with individual failures. In our work, the domain knowledge of the NASA personnel was invaluable in the process of data extraction and ensuring the quality of the data, thus minimizing the threats to validity of our analysis.

Although some specific aspect of our work were considered in related studies, overall we analyzed a much more comprehensive set of fault, failure and fix features and we explored novel research questions. For example, we explored which types of faults cause high priority failures, the distribution of feature values across releases, and the possibility of predicting characteristics of fixes based on information available before the fix is implemented. Additionally, we considered research questions that have been explored in the past, but have inconsistent results across different studies (e.g., the relationship between pre and post-release failures). Some of the most interesting empirical results include:

- Contrary to popular belief, later-life cycle faults are major sources of software failures. This observation is based not only on the NASA mission presented here but also the comparison to recent related studies. Further, we found that for the NASA mission coding faults account for the majority high priority failures (i.e., coding faults accounted for 50% of safety-critical failures and 52% post-release failures). Thus, these observations indicate the importance of conducting later life-cycle verification activities.

- Failures may be caused by multiple faults requiring fixes across multiple components and artifacts. Related studies relied on assumptions or used heuristics to handle multiple related faults, or simply limited the analysis to the first fix. Our results clearly show change tracking system must be designed to track multiple fixes associated with a single failure and that the practitioners and researchers have to take the spread of software fixes into account. For the NASA mission, the spread of fixes across components was significantly affected by the software architecture.

- Traditional data mining techniques used on fault and failure features were successful in predicting the effort required to implement fixes. Although predictions with respect to the artifacts fixed and architectural groups fixed could each be explained in terms of a single feature, fix effort could not. Despite the fact that effort prediction with respect to project schedule and development effort has been researched extensively, effort prediction specifically focussed on fixing faults has received little attention. Future research should further explore effort prediction focused on fixing faults as the quantification of fix effort showed that fixing faults associated with 20% of the failures accounted for 83% of the cumulative effort spent fixing all faults.

- Improvements in change tracking systems can help facilitate advancing the empirical knowledge from which best practices can be drawn. In our work the ability to track multiple fixes associated with a single failure led to interesting observations (e.g. with respect the spread of fixes across components and artifacts). Additionally, based on our exploration of change tracking system used by the mission as well as data used in related work we suggest using pre-defined data fields whenever possible as they make data extraction less complicated and improve data quality and the accuracy of the analysis. We also suggest tracking the life-cycle phase the fault was introduced in (in addition to the phase the fault was detected in, which is more commonly tracked) as it would make it possible to determine which faults could have been revealed earlier and the impact of finding them later.

We used a systematic approach and formal statistical analysis to quantify observed trends, which is not yet a common practice in software quality assurance. Additionally, throughout our work whenever possible we compared our results with recent related studies in order to explore the external validity of the study to help build empirical knowledge across the field. Using a well-defined terminology and taxonomy consistently across work related for software quality assurance would allow for much easier comparison of the results and greatly advance the empirical knowledge.

Finally, we note that we believe significant value was added by conducting this work in collaboration with the NASA personnel. It is important that researchers and practitioners work together to use empirical data to investigate theories, determine if and when specific phenomena hold true, and update the theories accordingly so that software engineering and quality assurance fields can follow other well established scientific disciplines. In the following section we discuss some ideas for future work.

12.1 Future Work

The novel contributions of this work are based on the link established between faults, failures and fixes, which is why the analysis was focused on defining and characterizing the relationships between faults, failures and fixes. Future work should explore how the metrics used in this work (i.e., fault, failure and fix features) compare to and/or can be combined with other metrics (e.g., static code metrics, classification ontologies, etc.), as well as explore whether the use of additional methods can improve the predictions.

As we discussed in Chapter 3, various metrics have been explored to predict fault and failure proneness. However, the successfulness of prediction models differ from study to study. Thus, although the fault and failure features we explored showed potential for predicting characteristics of fixes we believe the predictions may be improved with the use of additional features and that the true measure of success should be based on the analysis of similar features for other systems. In particular, more features specific to change requests may improve prediction results. For example, the features we consider may be complementary to the recent research in [47], which also uses features extracted from change requests. In particular, the work in [47] focuses on defining hierarchical aerospace ontologies of concepts and nomenclature to identify problem type and/or equipment type tags in the title and description fields of change requests to help identify similar change requests. Analysts' names and organizations may also be helpful. Additionally, it would be interesting to consider how fault and failures features can be combined with additional types of metrics (e.g., static code metrics, design metrics, etc.) to explore whether additional metrics of the failed component could improve prediction results with respect to locating fixes.

Further analysis conducted by domain experts could also provide practical benefit to the project. For example, further classification of coding faults based on common types of coding faults (e.g., logic faults, omission faults, typos, etc.) could be useful for updating coding standards and checklists used by the developers. Additionally, with respect to the verification activity quantifying the effort spent on each activity and further exploring the SCRs discovered by the common 'analysis' activity could each help characterize the most effective ways to find faults. A further study of safety-critical, on-orbit SCRs based on manual exploration may help identifying some trends and preventing such SCRs in the future.

A major area for future work is defining a standardized taxonomy that is representative of the relationships between faults, failures and fixes (e.g., a single failure caused by multiple faults). Such a taxonomy would promote comparison across multiple case studies and enrich the empirical software engineering knowledge based on accurate representation of the fault, failure and fix relationships.

In Chapter 10 we discussed some traditional machine learning techniques, that is, classification learning, association learning, numerical prediction and clustering, and applied those that were applicable. We believe exploring additional techniques could improve prediction results. In particular, methods based on multi-variate correlation may reveal additional interesting trends. For example, Bayesian Belief Networks (BBN) could be used to model dependencies among fault and failure features (and possibly other change request features) to determine the probability that a certain component would need to be fixed. Implication networks, which are made up of directed graphs in which the nodes represent individual variables or hypotheses and the arcs signify the existence of a direct implication (influence) between two adjacent nodes, may be useful to help analysts' as he or she implements the fix as the value taken on by one node is then dependent on the values taken on by all variables that influence it. Each value indicates the degree of belief that an unobserved variable is 'true'. Thus, as a component (or artifact) is fixed the implication network could be used to determine what other components will be fixed. Another benefit of using implication networks is that the values are updated every time new information is obtained (e.g., some evidence is observed), which means the network would be constantly updating as new change requests are entered and fixed. Additionally, considering higher-order associations (i.e., beyond 1st order association) through techniques such as sequence mining and multi-relational association rules mining may improve the usability of the association rules learned. Using the approach suggested in [64] it may be possible to incorporate the implication networks and association rules to form an associated network structure.

We believe the natural progression of the research work presented in this dissertation would be to explore addition information (i.e. features) as well as additional methods (i.e., additional data mining techniques) at various levels of abstraction (e.g., sub-component, module level, file level, etc.) on additional case studies.

References

[1] C. Andersson and P. Runeson,"A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems", *IEEE Transactions on Software Engineering*, Vol.33, No.5, May 2007, pp. 273-286.

[2] W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H. Ammar, B. Yu, S. Bogazzi, M. Korkmaz, and A. Mili. "Quantifying Software Architectures: An Analysis of Change Propagation Probabilities", *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*, Jan. 2005, pp. 124-131.

[3] W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H. Ammar, B. Yu, S. Bogazzi, M. Korkmaz, and A. Mili. "Quantifying Software Architectures: An Analysis of Change Propagation Probabilities", 3rd ACS/IEEE International Conference on Computer Systems and Applications, Jan. 2005, pp. 124-131.

[4] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases", 20th International Conference on Very Large Data Bases, 1994, pp. 478-499.

[5] E. Arishold, L.Briand, E. Johannssen. "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models". *The Journal of Systems and Software*, Vol. 83, August 2009, pp. 2-17.

[6] V. R. Basili, L.C. Briand, and W. L. Melo. "A validation of object oriented design metrics as quality indicators", IEEE Transactions on Software Engineering, 22(10), 1996, pp.751-761.

[7] V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation", *Communicationd of ACM*, Vol. 27, No. 1, 1984, pp. 41-52.

[8] A. Ben-David, "About the relationship between ROC Curves and Cohen's kappa", Engineering Applications of Artificial Intelligence Vol. 21 No. 6, September 2008, pp .874-882.

[9] R. Bell, T. J. Ostrand and E.J. Weyuker, "Looking for Bugs in All the Right Places", *Proc. 2006 International Symposium on Software Testing and Analysis*, 2006 pp. 61-72.

[10] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. "Putting it All Together: Using Socio-Technical Networks to Predict Failures" *17th International Symposium on Software Reliability Engineering*, Bengaluru-Mysuru, India, Nov. 2009.

[11] S. H. Biyani and P. Santhanam, "Exploring Defect Data from Development and Customer Usage on Software Modules over Multiple Releases", *Proc. 9th IEEE International Sympium on Software Reliability Engineering*, 1998, pp. 316320.

[12] Norman W. H. Blaikie, *Analyzing Quantitative Data*, SAGE Publication Ltd, 2003.

[13] B. W. Boehm, R. K. McClean and D. B. Urfrig, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software", *IEEE Transactions on Software Engineering*, Vol.1, No.1, 1975, pp. 125-133.

[14] B. Boehm, C. Abts, A. Brown, S.Chulani, B.Clark, E. Horowitx, R. Madachy, D. Reifer, D. Steece, "Software Cost Estimation with Cocomo II", Prentice Hall, Aug. 2000.

[15] D. Card, "Learning from Our Mistakes with Defect Casual Analysis", *IEEE Software*, Jan./Feb. 1998, pp. 56-63.

[16] G. Canfora and L. Cerulo, "How Software Repositories can Help in Resolving a New Change Request", In Workshop on Empirical Studies in Reverse Engineering, September 2005.

[17] G. Canfora and L. Cerulo, "Supporting Change Request Assignment In Open Source Developement", *Proc. 2006 ACM symposium on Applied computing*, Dijon, France, Apr. 2006, pp. 17671772.

[18] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. IEEE Transaction on Software Engineering, 20(6):476-493, 1994.

[19] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data", *Proc. 26th International Symposium on Fault–Tolerant Computing*, 1996, pp. 304-313.

[20] W. G. Cochran, Some methods for strengthening the common tests, Biometrics, 10, 1954, pp. 417-51.

[21] Conover WJ, Practical Nonparametric Statistics (3rd edition). Wiley 1999.

[22] J. A. Duraes and H. S. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach", *IEEE Transactions on Software Engineering*, Vol.32, No.11, 2006, pp. 849-867.

[23] A. Endres, "An Analysis of Errors and Their Causes in System Programs", *IEEE Transactions on Software Engineering*, Vol.1, No.2, 1975, pp. 140-149.

[24] W.Evanco , "Prediction Models for Software Fault Correction Effort", Fifth European Conference on Software Maintenance and Reengineering, March 2001, Libson, Portugal, pp. 114-120.

[25] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, . "From Data Mining to Knowledge Discovery in Databases", AI Magazine, Vol. 17, 1996, pp. 37–54.

[26] N. Fenton and N.Ohlsson, “Quantitative Analysis of Faults and Failures in a Complex Software System”, *IEEE Transactions of Software Engineering*, Vol.26, No.8, 2000, pp. 797-814.

[27] A. Garg and D. Roth, ”Understanding Probabilistic Classifiers”, Proceedings of the 12th European Conference on Machine Learning, pp. 179-191, September 2001.

[28] R. L. Glass, *Software Runaways. Lessons Learned from Massive Software Project Failures*, Prentice Hall, 1998.

[29] K. Goševa-Popstojanova, M. Hamill, and R. Perugupalli, “Large Empirical Case Study of Architecture-Based Software Reliability”, Proc. 16th IEEE Intl Symp. Software Reliability Eng., pp. 43-52, Nov. 2005.

[30] K. Goševa-Popstojanova, M. Hamill, and X. Wang, “Adequacy, Accuracy, Scalability, and Uncertainty of Architecture-Based Software Reliability: Lessons Learned from Large Empirical Case Studies”, Proc. 17th IEEE Intl Symp. Software Reliability Eng., pp. 505-514, Nov. 2006.

[31] K. Goševa-Popstojanova and M. Hamill, “Report on Effects of Failure Clustering on Software Reliability”, Technical Report submitted to NASA OSAM SARP, March 2007.

[32] K. Goševa-Popstojanova and M. Hamill, “Lessons Leanred from an Emprical Study of a Safety–Critical NASA Mission”, Technical Report submitted to NASA OSAM SARP, May 2007.

[33] K. Goševa-Popstojanova and M.Hamill, “10 Things to Know about Faults and Failures for a Safety–critical NASA Mission”, submitted to the NASA Mission Personnel.

[34] T. L. Graves, A. F. Karr, J. S. Marron and H. Siy, “Predicting Fault Incidence Using Software Change History”, *IEEE Transaction on Software Engineering*, Vol.26, No.7, 2000, pp. 653-661.

[35] M.L. Griss, “Software Reuse: From Library to Factory,” IBM Systems Journal, Vol. 32, pp.548-565, 1993.

[36] M. Hall and G. Holmes, ”Benchmarking Attribute Selection Techniques for Discrete Class Data Mining,” IEEE Transactions on Knowledge and Data Engineering, Vol. 15 No. 6, Nov./Dec. 2003, pp. 1437-1447.

[37] M. Hamill and K. Goševa-Popstojanova, “Common Trends in Fault and Failure Data”, *IEEE Transactions on Software Engineering*, July/August 2009.

[38] P.S. Heckerling, ”Parametric three-way receiver operating characteristic surface analysis using Mathermatica”, Medical Decision Making, 2001, Vol. 21 No.5, pp. 409-417.

[39] IEEE Std $610.12-1990$ IEEE Standard Glossary of Software Engineering Terminology.

[40] D. Jackson, M. Thomas and L. Millett, *Software for Dependable Systems: Sufficient Evidence?*, National Academies Press, 2007.

[41] Y. Jiang, B. Cukic, T. Menzies, N. Bartlow. "Comparing Design and Code Metrics for Software Quality Prediction". PROMISE '08 May 12-13 2008. Leipzig, Germany.

[42] M. Jorgensen. "A review of studies of expert estimation of software development effort". *Journal of System and Software*, Vol.70 no.1-2 pp. 37-64, 2004.

[43] M. Leszak, D. Perry and D. Stoll, "Classification and Evaluation of Defect in a Project Retrospective", *Journal of Systems and Software*, Vol.61, Apr. 2002, pp. 173-187.

[44] www.kde.org

[45] R. R. Lutz and I. C. Mikulski, "Empirical Analysis of Safety Critical Anomalies during Operation", *IEEE Transactions of Software Engineering*, Vol.30, No.3, Mar. 2004, pp. 172-180.

[46] R. R. Lutz and I. C. Mikulski, "Ongoing Requirements Discovery in High-Integrity Systems", *IEEE Software*, Vol.21, No.2, Mar. 2004, pp. 19-25.

[47] J. Malin, "Automated Tool and Method for System Safety Analysis", OSMA SARP Funder Research Project, available at http://sarpresults.ivv.nasa.gov/ViewResearch/129.jsp

[48] A. Mockus, D. Weiss, P. Zhang, "Understanding and predicting effort in software projects", *International Conference of Software Engineering*, 2002, pp. 274-284.

[49] D. Mossman, "Three-way ROCs", Medical Decision Making, Vol. 19 No.1, 1999, pp. 78-89.

[50] www.mozilla.com/firefox/

[51] NASA Certification Processes for Safety-Critical and Mission-Critical Aerospace Software, available online at 'http://www.hq.nasa.gov/office/codeq/doctree/871913.htm'.

[52] NASA Software Safety Guidebook, available online at 'http://www.hq.nasa.gov/office/codeq/doctree/871913.htm'.

[53] N. Ohlsson and H. Alberg."Predicting fault-prone software modules in telephone switches". IEEE Transactions on Software Engineering, Vol. 22, No. 12, 1996, pp 886-894.

[54] T. J. Ostrand and E. J.Weyuker, "The Distribution of Faults in a Large Industrial Software System", *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA 2002)*, 2002, Rome, Italy, pp. 55-64.

[55] T.J. Ostrand, E. J. Weyuker and R.Bell, "Where the Bugs Are", *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 86-96.

[56] T.J. Ostrand, J. Weyuker and R.Bell, "Predicting the Location and Number of Faults in Large Software Systems", *IEEE Transactions of Software Engineering*, Vol.31, No.4, Apr. 2005, pp. 340-355.

[57] M. Panda and M. Patra, "A comparative study of data mining algorithms for network instrusion detection", Proceedings of the 1st International Conference on Emerging Trends in Engineering and Technology, pp. 504-507, July 2008.

[58] M. Pighin and A. Marzona, "An Empirical Analysis of Fault Persistence Through Software Releases", *Proc. International Symposium on Empirical Software Engineering (ISESE'03)*, 2003, pp. 206-211.

[59] R. Quinlan, "C4.5: Programs for Machine Learning", Morgan Kaufmann Publishers, San Mateo, CA.

[60] S. Seigel, *Non Parameteric Statistics for the Behavoiral Sciences*, McGraw-Hill, 1956.

[61] M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, H.H. Ammar, H.H., "Information theoretic metrics for software architectures, Computer Software and Applications Conference", COMPSAC 2001, pp. 151-157.

[62] Sherriff, M., Lake, J. M., and Williams, L. "Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records." *The 18th IEEE International Symposium on Software Reliability Engineering*, Trollhattan, Sweden, Nov. 2007, pp. 81-90.

[63] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction", IEEE Transaction on Software Engineering, February 2006, Vol. 32. No. 2, pp. 69-82.

[64] P. Tse and J. Liu, "Mining Assoicated Implication Networks: Computational Intermarket Analysis", Second IEEE International Conference on Data Mining, Maebashi City, Japan, Dec. 2002, pp. 689-692.

[65] C. Weib, R. Premraj, T. Zimmerman, and A. Zellar,"How Long will it Take to Fix This Bug", Fourth International Workshop on Mining Software Repositories, 2007.

[66] I.H. Witten and E. Frank, "Data Mining: Practicel Machine Learning Tools and Techniques, Second Edition, Morgan Kuafmann Publishing 2005.

[67] J. Xu, D. Ho, L.F. Capretz, "An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction", Journal of Computer Science Vol. 4 No.7, 2008, pp. 571-577.

[68] W. D. Yu, "A Software Fault Prevention Approach in Coding and Root Cause Analysis", *Bell Labs Technical Journal*, Vol.3, No.2, Apr-Jun 1998, pp. 3–21.

[69] Zage W, Zage D., "Evaluating design metrics on large-scale software", IEEE Software, Vol. 10 No. 4, 1993, pp. 75-81.

[70] Zage W, Zage D. Identifying desing connectivity in metrics analyses of software systems. Proceedings of the Eight Annual Software Tecnology Conference, Salt Lake City, UT, April 1996.

[71] Zage W, Zage D, Bhargava M, Gaumer D. Design and code metrics through DIANA based tool. Ada: Moving Towards 2000 (Lecture Notes in Computer Science vol. 603). Springer-Verlag, 1992, pp. 60-71.

[72] H. Zeng and D.Rine, ”Estimation of Software Defects Fix Effort Using Neural Networks”, Proceedings of the 28th International Computer Software and Applications Conference, 2004

[73] M. Zhoa, C. Wohlin, N. Ohlsson, and M. Xie. ”A comparison between software design and code metrics for the prediction of software fault content”. Information and Software Technology, 40(14):801-809, 1998.

[74] T. Zimmermann and N. Nagappan, “Predictiing Subsytem Failures using Dependency Graph Complexities”, *18th IEEE International Symposium on Software Reliability Engineering*, Trollhattan, Sweden, Nov. 2007, pp. 227-236.

[75] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs” *30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 531–540

[76] WEKA:http://www.cs.waikato.ac.nz/ml/weka

John H. Hagen

Digitally signed by John H. Hagen
DN: cn=John H. Hagen, o=West Virginia University Libraries, ou=Acquisitions Department, email=John.Hagen@mail.wvu.edu, c=US
Date: 2010.05.06 09:25:52 -04'00'