SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* 2012; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

# Exploring the missing link: An empirical study of software fixes

Maggie Hamill[1†] Katerina Goseva-Popstojanova[2*]

[1]*Department of Computer Science, Northern Arizona University, Flagstaff, AZ, USA*
[2]*Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA*

SUMMARY

Many papers have been published on analysis and prediction of software faults and/or failures, but few addressed the software fixes made to correct the faults and prevent failures from reoccurring. This paper contributes toward filling this gap by focusing on empirical characterization of software fixes. The results are based on the data extracted from a safety-critical NASA mission. In particular, 21 large-scale software components (which together constitute over 8,000 files and millions of lines of code) were analyzed. The unique characteristic of this work is the fact that links were established from software faults (i.e., the root causes) to (potential or observed) failures and consequently to fixes made to correct these faults. Specifically, for the fixes associated with individual failures the spread across software components and types of software artifacts being fixed were studied. Our results showed that significant number of software failures required fixes in multiple software components and/or multiple software artifacts (i.e., 15% and 26%, respectively). The results also showed that the patterns of software components that were often fixed together were significantly affected by the software architecture. Furthermore, the types of fixed software artifacts were highly correlated with fault type and they had different distributions for pre-release and post-release failures.
Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Empirical study; software faults; software failures; software fixes; software artifacts; distribution of software fixes

## 1. INTRODUCTION

As modern society becomes more and more reliant on software systems, ensuring proper functioning of such systems becomes essential. In the software engineering field quality assurance practices are used to systematically monitor and evaluate various aspects of software products and processes to ensure that quality standards are being met. Quality assurance terms, however, are often used inconsistently in the literature. For example, the term bug is used to refer to faults in the code. The term defect in some cases refers to both faults and failures and in other cases only to faults or perhaps, faults detected pre-release. Anomalies typically refer to conditions that deviate from expectations based on requirements specifications, user documents, or from someone's perceptions and thus in addition to failures include cases of unexpected behavior in the eyes of the operator. The definitions of the terms *failure*, *fault* and *fix* as they are used in this paper are provided next.

- A *failure* is a departure of the system or system component behavior from its required behavior. In this paper both observed and potential failures are considered, that is, failures

*Correspondence to: Katerina Goseva-Popstojanova, Lane Department of Computer Science and Electrical Engineering, West Virginia University, P.O. Box 6109, Morgantown, WV 26506, USA. E-mail: Katerina.Goseva@mail.wvu.edu
†The work was done while Maggie Hamill was affiliated with West Virginia University.

Copyright © 2012 John Wiley & Sons, Ltd.

that occurred during operation, as well as failures that were prevented from happening by detecting and fixing the problems during development and testing.

- A *fault* is an accidental condition, which if encountered, may cause the system or system component to fail to perform as required. Faults can be introduced at any phase of the software life cycle; that is, they can be tied to any software artifact (e.g., requirements, design, source code, documentation, etc). It should be noted that every fault does not correspond to a failure since the conditions under which fault(s) would result in a failure may never be met.
- A *fix* refers to all changes made to correct the fault(s) that caused an individual failure. A fix, which permanently addresses the root cause(s) of the failure is distinguished from a work-around, which is implemented to circumvent the problem without actually correcting the faults. In this paper the terms fix and change are used interchangeably since only changes made to fix faults and prevent failures from reoccurring are explored.

Figure1 illustrates the relationship among faults, failures, and fixes. Widely known taxonomies [1] and standards, such as lEEE Std 610.121990 [2] and ISO/IEC/IEEE 24765 [3], include variants of the *failure* and *fault* definitions. One difference from these documents is that in this paper the fault(s), which is (are) the actual cause of a failure, are separated from the *fix* (i.e., correction), which refers to the changes made in one or more software artifacts to prevent the failure from (re)occurring. As an example, the fault may be a missing requirement (i.e., omission fault), which early in the life cycle (e.g., during requirements review) can be fixed by adding the missing requirement only, while later in the life cycle (e.g., post-release) would also require adding extra code.

A similar approach to treating the terms fault, failure and fix was taken by Eldh et al [4] with a goal to build a realistic fault models to be used for fault injection in the source code of a large telecommunication software. With respect to definitions appearing elsewhere, the most similar to the term *fix* (as it is used in this paper) appears to be the term *repair*, which is defined in the ISO/IEC/IEEE 24765 standard [3] as "the correction of defects that have resulted from errors in external design, internal design, or code."

The effort to fix faults and build highly reliable software can benefit significantly from knowledge of the past. However, unlike other engineering areas that have a long tradition of learning from failures and not repeating them, most software industries do not pay enough attention to understand the typical faults that exist in their software [4]. Thus, while significant amount of empirical studies have been focused on studying software faults and/or failures very few works tie the faults to the failures that they caused and study the relationships among faults, failures, and fixes implemented to correct faults. One of the main reasons for this is the difficulty to gather the information if traceability of the code (or other changed artifacts) is not directly possible from the reported failure. Thus, although most medium to large software development projects have a mechanism in place, typically called bug, problem or issue tracking system, to track software faults, these tracking systems often do not record how, where, and by whom the problem in question was fixed [5]. The information about fixes typically is hidden in the version control system, which records all changes to the software source code. However, attributing the changes to specific faults often is far from trivial because no identification is given in the source code change logs whether a change was made because of fixing fault(s), enhancement(s), or implementation of new feature(s) [6], [7].

Researchers have recognized that a problem described in a software change request may be repaired by making one or more changes to one or more files, but it is a typical practice to count one fault for each file affected by a software change request, i.e., if $m$ files are changed due to a single software change requests then $m$ faults are counted (see for example [8], [9], [10], [11], [12], [13] and references therein.) Although this fault counting convention may be adequate for assessment and prediction of fault-proneness at unit level (e.g., file, component, or package), it lacks information about the spread of software fixes tied to individual failures. We believe that disregarding the fact that software fixes are not always localized is a dangerous practice which may lead to overly optimistic assessments and predictions in areas such as evaluation of testing strategies effectiveness and software reliability.

Another limitation of the existing studies is the fact that they are typically focused on fixes (i.e., corrections) made in the source code only, which is due to the fact that keeping track of changes

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

made in other software artifacts (e.g., requirements, design documents) does not seem to be a common practice, especially among open source projects, which are often used as case studies in the software quality assurance research. Disregarding the changes made to other software artifacts leads to a simplified view and underestimates the effort needed to address other potential sources of software failures.

The goal of this paper is to contribute towards building a knowledge base about software failures and changes made to fix the faults that are root causes of these failures, which is crucial to several research areas in testing (e.g., fault-based testing, testability, mutation testing, comparative evaluation of testing strategies) [14] and software reliability assessment and prediction [7], [15]. In addition to exploring the spread of fixes across software components, our work accounts for changes made to different software artifacts (e.g., requirements, design documents, source code, supporting files). The study is based on a large-scale safety-critical NASA mission, for which failures were linked to the faults that caused them and then to changes that were made to fix these faults in different types of software artifacts. In particular, two sets of research questions are explored in this paper.

The first set of questions is focused on exploring the characteristics of software fixes in terms of components and software artifacts they affected.

RQ1: Do individual failures require fixes that are spread across multiple components?
RQ2: Do the fixes affect certain components more often than others? Further, are there groups of components that are commonly fixed together?
RQ3: Do the fixes affect certain types of software artifacts more often than others? Further, are there groups of artifacts that are commonly fixed together?

The second set of research questions explore how the types of fixed attributes are related to the fault types, as well as how they are distributed for pre-release and post release failures.

RQ4: Are the type(s) of fixed software artifacts related to the fault type that caused the failure?
RQ5: Are types of fixed artifacts different for pre-release and post-release failures?

The work presented in this paper is novel in several aspects:

- Unlike most of the related work which studied either faults or failures, this work is focused on analysis of fixes related to faults through individual (potential or observed) failures. Furthermore, this study is not limited to changes made in source code only as it is often done, but rather it considers changes made to any software artifact (e.g., requirements, design documents, code, etc.)
- Rather than studying changes made for any reason (e.g., product improvement, adding new features, and/or fixing faults), our work is focused on the changes made to fix faults and prevent individuals failures. Furthermore, unlike studies in related work that considered the first or primary fix only, this study considers fixes that affect multiple software components and/or multiple software artifacts.

The findings of this study lead to observations and insights that can help increasing the efficiency of addressing future faults and failures for software products that are incrementally developed through releases and/or need sustained engineering.

The paper proceeds by reviewing the related work in section 2. The details of the NASA mission used as a case study and the description of the extracted data are presented in section 3. Then, in section 4 fixes are characterized in terms of the affected components and artifacts, while in section 5 the associations between attributes of fixes and fault and failures attributes are analyzed. Finally, the threats to validity are discussed in section 6 and the concluding remarks are given in section 7.

## 2. RELATED WORK

Although the empirical analysis of fault and failure data has been quite rare in the past, it seems the value of such studies is finally being recognized and more studies are being published and

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

cross compared. For example, Fenton and Ohlsson [8] and Andersson and Runeson [9] conducted detailed, fairly comprehensive empirical analysis of faults in telecommunication systems. In fact, the study conducted by Andersson and Runeson [9] was a replication study aimed at exploring whether the phenomena observed by Fenton and Ohlsson [8] held true across different studies. Two independent studies, one based on faults reports for an IBM operating system [16] and the other on twelve open-source projects [17], showed strong consistency in the fault distribution across different fault types. In our earlier work [18], we specifically investigated the consistency of our results with the results presented in previous studies [8], [9], [16], [17], [19], [20], [21]. The analysis showed that, contrary to the popular belief, the late life-cycle faults are often causes of failures. Additionally, for a small subset of the data used in this paper (i.e., 404 failures tied to changes in source code only), the results showed that 12% of failures led to fixes in at least two components.

A study of a middleware software system conducted at Ericsson by Eldh et al. [4] analyzed 362 failures and, consistent with our previous work [18], showed that more than 34% of the corrections (i.e., fixes) tied to individual failures involved more than one file (with a maximum of 25 files). However, both Eldh's et al. study [4] and our previous study [18] were focused on fixes to the source code only.

Of the studies that did analyze faults and failures very few characterized the fixes. Those studies that did study the fixes limited their scope to either the primary fix or the first fix. Thus, in a study conducted by Leszak et al. [19] several attributes of 427 modification requests (MRs) from a network element of an optical transmission network were analyzed, including fault types, phase introduced, phases detected, as well as fix location (i.e., document, hardware, or software) and effort (i.e., effort to reproduce, investigate and repair but not re-test). The study was limited to post-release MRs and only considered the first created MR, without considering sub-MRs which were used to track additional changes related to the primary MR. Similarly, Lutz and Mikulski, who analyzed safety-critical post-launch anomalies from seven unmanned NASA spacecrafts, limited the scope of the analysis of fixes [20]. The study used four attributes from the Orthogonal Defect Classification (i.e., type, trigger, target, activity), two of which are related to fixes: the target attribute where the fix occurred (e.g., ground software, flight software, information development, hardware) and the type attribute which described the nature of the fix (e.g., procedure, algorithm, no fix). However, for a practical reason the analysis of the target was limited to the first fix only, even though it was observed that for some anomalies multiple targets existed. By limiting the analysis to single fixes per modification request [19] or per anomaly report [20] it is not possible to explore how fixes spread across different artifacts or parts of the systems (e.g., files or components) as we did in this paper.

Several studies explored the nature of changes based on change request data [22], [23], [24]. Sherriff and Williams studied which files were changed together historically with a goal to prioritize regression test cases and thus decrease the amount of regression faults delivered to the field [22]. The work by Canfora and Cerulo was focused on determining what files tended to change together, as well as what changes were made by which developers [23], [24]. It was shown that when keywords from a new change request matched the keywords from older change requests, the files that needed to be changed for the newer request were often the same files changed for the older requests. These studies, however, did not differentiate between changes made to fix faults and enhancements. Furthermore, the analysis presented in studies [22], [23], [24] was limited to changes in the source code only, whereas in this paper we consider changes made across different types of software artifacts.

Characterizing the location of changes made to fix faults is related to change propagation, which measures the likelihood that a change that arises in one component of the architecture propagates (i.e., mandates changes) to other components. Abdelmoez et al. used early life cycle information about software architecture to derive an analytical formula for estimating the change propagation probability [25].

Metrics related to software architecture have been used to predict fault and failure proneness in several papers [26], [27], [28]. Thus, Zimmerman and Nagappan studied failure proneness of the Windows Server 2003 based on dependencies between parts of the code and the complexity of dependency graphs [26], [27]. The results showed that models accounting for architectural

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

dependencies do better at predicting post-release failures than models based on the typical complexity metrics. Bird et al. focused on predicting post-release failure proneness based on the dependency structure between components and task assignments (i.e., who worked on which component how much) [28]. The prediction models were evaluated on the Vista operating system and multiple versions of the integrated development environment Eclipse. The results showed that these models performed as well as, or better than models that did not consider dependency structure and task assignments.

The contributions of the work presented in this paper are related both to the approach taken to analyze the data and to the observations made based on the findings. First, failures are linked to the faults (i.e., the root cause) and then to fixes that have been made to fix the problem. It is important to note that the scope of the fixes is not limited. Thus, unlike previous studies which only considered the primary fix or the first fix (e.g., [19], [20]), in this paper all fixes that have been made in connection to an individual failure are investigated. This allows the spread across multiple artifacts and multiple components to be explored. It appears that the only prior works that considered the spread of software fixes (i.e., corrections) tied to individual failures are the work by Eldh et al. [4] and our earlier work [18]. However, both works [4], [18] were based on much smaller samples (i.e., 362 and 404 failures, respectively) and were focused on the changes made in the source code only. In this paper we specifically investigate the fixes made to different types of software artifacts (e.g., requirements, code) and their combinations.

With respect to the works that considered software architecture in relation to changes, our goal is quite different. Thus, while the goal of the study presented by Abdelmoez et al. [25] was to compute the change propagation probability and the goal of studies conducted by Zimmerman and Nagappan [26], [27] and by Bird et al. [28] was to predict the fault and/or failure proneness at code level, our goal is to characterize software fixes in terms of components and software artifacts that are typically fixed together in relation to individual software failures, which will help improve the efficiency of future fixes for products that undergo incremental development through multiple releases and need sustained engineering.

## 3. CASE STUDY DESCRIPTION AND METRICS DEFINITION

The study presented in this paper is based on the data extracted from the change tracking system of a large, safety-critical NASA mission. The analyzed data spans almost ten years of development and operation for 21 Computer Software Configurations Items, which represent large-scale software components. These components were built following an iterative development process by teams at several locations. The number of locations varied from two to four throughout the project lifetime. The mission is still active and requires sustained engineering.

In order to analyze software fixes, that is, changes made to fix software faults that have caused or may have caused software failures, the analysis is focused on the Software Change Requests (SCRs) which were made because the software failed to conform to some requirement(s) (i.e., so called non-conformance SCRs). These SCRs represent either potential failures (i.e., those that were revealed pre-release during development and testing and avoided by preemptively fixing faults before they could lead to failures) or post-release failures (i.e., those that were observed on-orbit). Each SCR identifies the component which failed (or was suspected to fail), the type of fault that caused the failure, and the type of activity taking place when the fault was detected or the failure was revealed.

For this mission, changes made to address SCRs are tracked in the form of Change Notices (CNs), which are linked to the SCR they are addressing. Each CN is associated with only one component being fixed, and identifies the software artifacts that were fixed. It is important to note that more than one CN can be linked with a single SCR, that is, each SCR can have zero, one, or multiple CNs linked with it. The fact that the detailed information on the changes made to fix the fault(s) was associated with each non-conformance SCR (i.e., failure) allowed us to track and analyze fixes that span multiple components and/or software artifacts. Specifically, the analysis presented in this paper is based on the following attributes, which were extracted from the non-conformance SCRs and the associated CNs:

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

- *Failed component* is the component against which a non-conformance SCR was reported. It is identified by the project analysts and recorded as a data field in each non-conformance SCR.
- *Fault type* refers to the root cause of the failure. For each non-conformance SCR, project analysts select the fault type value from a pre-defined list. With the help of the project personnel these values were grouped into the following major fault type categories: *requirements faults*, *design faults*, *coding faults*, *integration faults*[†], and *other faults*. Similarly as in case of the Orthogonal Defect Classification (ODC) [29], the values of fault type attribute are mutually exclusive, that is, there is only one root cause of a failure, although it may have multiple manifestations. For example, if a failure occurred during integration testing and it was caused by a coding error, which in turn was due to missing requirement, the fault type would be classified as a requirements fault.
- *Pre-release / post-release occurrence* specifies when the fault was detected or the failure was exposed. It is identified in each non-conformance SCR. For the analysis presented in this paper SCRs that were created based on non-conformance observed on-orbit are referred to as post-release failures. All other non-conformance SCRs are referred to as pre-release failures.
- *Fixed component* is the component affected by the changes made to fix faults and prevent failures from (re)occurring. As already discussed, each CN has a field which identifies a single component that has been changed; multiple CNs can be associated with a single SCR, which means multiple components can be affected as a result of a single non-conformance SCR (i.e., failure).
- *Fixed artifact* refers to the types of artifacts affected by the changes made to fix faults and prevent failures from (re)occurring. Fixed artifacts are identified in a CN field. The types of affected artifacts include: *requirements documents*, *design documents*, *code files*, *supporting files* (e.g., pre-defined look up tables), *tools* (e.g., testing, simulation, configuration tools), and *notes/waivers*. The notes and waivers typically serve to identify non-conformances SCR that have not yet been fixed. A note is a work-around written to prevent a known failure from happening (e.g., avoid or alter a certain command sequence). On the other side, a waiver is written to acknowledge that a requirement cannot be met and that the mission is currently willing to accept the limitation.

The analysis presented in this paper was conducted at component level because the vast majority of data was kept at that level. All non-conformance SCRs that had at least one CN and 'closed' change status (i.e., for which the changes were completed) were included in the analysis. Table I shows the size in number of files and lines of code (LOC), the numbers of closed SCRs and the number of associated CNs with each of the 21 components considered in this paper. A total of 1,257 closed non-conformance SCRs were linked to total 2,620 CNs, out of which 2,496 CNs shown in Table I were linked to the 21 components analyzed in this paper and an additional 124 CNs were linked outside these 21 components. The fact that the closed non-conformance SCRs were associated with twice as many CNs, clearly shows the importance of considering multiple fixes for each individual failure.

## 4. HOW ARE COMPONENTS AND SOFTWARE ARTIFACTS AFFECTED BY FIXES?

In this section the spread of software fixes associated with individual failures (i.e., non-conformance SCRs) and the types of software artifacts affected by these fixes are explored.

### *4.1. RQ1: Do individual failures require fixes that are spread across multiple components?*

The fact that the NASA mission keeps track of changes made to address each individual failure by linking each non-conformance SCR to one or more CN allowed us to study the spread of

[†]Integration faults type in this paper integrates the 'data problems' and 'integration faults' types used in our previous work [18].

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

software fixes across components. Our results show that out of the 1,257 SCRs filed against the 21 components:

- 82% led to changes only in the Failed component (i.e., component against which the SCR was reported).
- 15% led to changes in the Failed component and at least one other component.
- 3% led to changes in some component(s), but not in the Failed component.

As expected, a large percentage of SCRs led to changes localized to the component against which the SCR was reported. However, 15% of SCRs, which is a considerable percentage, led to changes in two or more components (including the Failed component). The 3% of SCRs for which the Failed component was not fixed at all (i.e., one or more changes were made to other components) may be explained by the fact that sometimes a component simply serves as a catalyst for a failure to surface even though all associated fault(s) are contained outside that component. According to the project personnel, this happens because components share data or because in some cases the quickest and easiest way to fix a problem is through a different component.

To further explore the spread of fixes across software components, the mean and the standard deviation of the number CNs associated with individual SCRs, broken down per component, are presented in Table II. Table II also contains the coefficient of variation, which is defined as the ratio of the standard deviation and the mean and thus can be used to compare the variation, even if the means are drastically different from one another. (Statistical moments are not given for component 9 because, as shown in Table I, it has only one closed SCRs.) It can be seen that across the 21 components the mean number of fixed components associated with individual failures (i.e., the mean number of CNs per SCR) ranges from 1.00 to 2.77. Furthermore, components with the larger means tended to have higher coefficients of variation.

Figure 2 shows that SCRs for different failed components have similar medians, typically one or two fixed components (i.e., CNs). The semi-quartile difference, which is defined as one-half of the difference between the first and third quartiles and typically is used as a measure of spread for skewed distributions, is fairly consistent across SCRs associated with different failed components. However, several rather long whiskers point out to individual failures that required changes in considerable number of components. For example, 41 components were fixed (which resulted in 41 created CNs) to address one SCR filed against component 20.

The non-parametric Kruskal-Wallis test was used to test the hypothesis that the distributions of the number of fixes associated with individual failure (i.e., the number of CNs per SCR) are identical for all components. Since the hypothesis of equal distributions was rejected (p-value $= 8.797 \times 10^{-6}$), the multiple comparison test implemented in R language was used to determine which pairs of components differed. The results showed that the distributions of the number of fixes per failure (i.e., CNs per SCR) were different only for three pairs of components, all of which involved component 20 (i.e., 20 and 1, 20 and 17, and 20 and 21).

The main finding with respect to RQ1 are:

- **Significant percentage of software failures (in this case study 15%) required implementing fixes that are spread across multiple components.**
- **The average number of fixed components per failure was from 1 to 2.77**, i.e., there appears to be some central tendency. However, some failures (i.e., non-conformance SCRs) required changes to be made across a large number of components, up to 41 in case of one SCR filed against component 20.

### 4.2. *RQ2: Do the fixes affect certain components more often than others? Further, are there groups of components that are commonly fixed together?*

The fact that 15% of non-conformance SCRs led to fixes in multiple components (i.e., at least one other component in addition to the component against which the SCR was filed) motivated us to explore how these fixes were spread across the software architecture at the component level. Specifically, we explored the relationship between the component against which the SCR

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

was reported (i.e., Failed component) and the component(s) in which the fixes were implemented (i.e., Fixed component(s)), which for brevity is called the Failed component – Fixed component(s) relationship.

To allow for a larger sample of CNs (i.e., fixes), the analysis related to RQ2 was based on non-conformance SCRs with both 'open' and 'closed' status. This can be considered as a conservative view since for a given set of Failed components the number of Fixed components can only increase when the currently open SCRs are closed[‡].

In the three dimensional bar chart shown in Figure 3, the $x$-axis represents the Failed components and $y$-axis represents the Fixed components. For each Failed component the bars represent the number of times each of the 21 components was fixed as a result of failures (i.e., non-conformance SCRs) filed against that particular Failed component (i.e., the cumulative number of CNs associated with the cumulative number of SCRs written against the Failed component). Because the goal was to study how fixes were affected by the software architecture, the order of components shown on the axes in Figure 3 reflects the software architecture. Specifically, this mission follows a hierarchical architecture consisting of three levels (referred to as level I, level II, and level III in this paper). The first component listed on both the $x$-axis (i.e., Failed component axis) and the $y$-axis (i.e., Fixed components axis) in Figure 3 is component 20, which is the single top level component in the three 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 $A$ to $F$ shown in Figure 3. These six components (1, 17, 21, 19, 14, and 18) appear first in their corresponding groups. The remaining components in each group make up the third levels in the hierarchical architecture. Figure 4 shows the same data as Figure 3 but uses percentages rather than raw numbers so that fixes made outside the Failed component can be explored in more detail and the results can be compared across components.

Based on Figures 3 and 4 the following main observations are made:

- **A significant percentage of fixes were associated with the single top level component**, which is not surprising having in mind that component 20 is the largest component, with the highest number of SCRs filled against it. In particular, 32% of the total number of fixes (i.e., CNs) were implemented in component 20, mostly as a result of SCRs filed against component 20 (i.e., cases in which component 20 was the Failed component). Thus, 94% of the all fixes in component 20 were due to failures in component 20, 3.43% were due to failures in level II components, and 2.45% were due to failures in level III components.
- **74% of the total number of fixes were implemented in the Failed components** as indicated by the large values on the diagonal in Figure 3[§]. The remaining 26% of the total number of fixes were implemented in at least one component other than the Failed component, which again shows that fixes were not localized to the component reportedly failing.
- **Failures of components from level III were more likely to lead to fixes outside the Failed component than failures of components from level I and level II.** Specifically, on average 44% of fixes implemented due to failures in components from level III (with a range from 17% - 77%) led to fixes outside the Failed component. On the other side, on average only 10% of fixes caused by failures in Level II components (with a range from 1% to 35%) led to fixes outside the Failed component. The most prominent exception at level II was component 18; 35% of the fixes implemented due to failures in component 18 led to fixes outside component 18, out of which 7% were located in component 20 (i.e., the top level component) and 28% in other level III components from the same group F to which component 18 belongs. Only 1% of fixes associated with failures of component 20, the single component at level I, led to fixes outside component 20.

[‡]It should be noted that all other results in this paper are based only on analysis of closed SCRs. We repeated the complete analysis using the larger sample of both opened and closed SCRs and validated that the major trends reported throughout the paper do not change.

[§]It should be noted that because each SCRs can have multiple CNs, although 82% of SCRs required changes within the Failed component only, only 74% of CNs were implemented within the Failed components.

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

- **Fixes typically remained within the same architectural group.** 73% of the fixes that occurred outside the Failed component remained within the architectural group of the Failed component. For example, many fixes due to failures of components in group F were located outside the Failed component (i.e., from 29% to 48% depending on the component). Most of them, however, were located in other components within group F, that is, only around 4% of fixes were implemented outside of the group F. Other architectural groups had similar behavior – groups A, B, C, D, and E respectively had 6%, 13%, 6%, 1% and 7% of fixes implemented outside their group due to failures within the group.
- **Fixes outside the architectural group were mainly due to interconnected groups that perform integrated operations.** Group B had the highest percentage of fixes implemented outside the group due to failures within the group B, which was due to two main reasons: (1) failures filled against component 2 led to fixes distributed to all components except component 19 (out of which 30% were outside of group B) and (2) some of the failures associated with components in group B led to fixes located in components 6 and 12 from the architectural group A. Interestingly, failures of component 6 (which belongs to group A) resulted in significant percentage of fixes (i.e., 30%) to components in group B. The last two observations are explained by the fact that architectural groups A and B 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; this was also the case for components 14 and 15.

These results clearly indicate that the distribution of fixes is significantly affected by the software architecture. Exploring the correlation between software architecture and implemented fixes can benefit both the developers and independent verification and validation personnel and contribute towards more cost efficient improvement of the product quality. For example, identifying patterns, such as a group of components that are often changed together, can be used for planning integration testing, and can contribute towards more efficient resolution of SCRs with 'open' status or newly filled SCRs by suggesting where the fixes are likely to occur. Additionally, identifying problems such as undesired high coupling among components or non-typical behaviors (e.g., fixes distributed across almost all components due to failures linked to one specific component) can pinpoint parts of the software system that may need to be refactored or redesigned in the future releases of the system.

### *4.3. RQ3: Do the fixes affect certain types of software artifacts more often than others? Further, are there groups of artifacts that are commonly fixed together?*

We are in a unique position to explore this research question because each CN, in addition to identifying the component that was fixed, also identifies the software artifact that was fixed. As stated in Section 3, the artifacts were grouped 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 and Waivers* (i.e., documents which identify work-around or acknowledge that the requirement cannot be met).

When considering the artifacts listed in all CNs associated with individual SCRs, 74% of all 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 tied to an individual failure.

Figure 5 shows a Venn diagram, which represents the common types of artifacts and combinations of artifacts that were changed together in addressing an individual failure. Figure 5 accounts for over

Copyright © 2012 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

94% of the the total number of SCRs. The remaining 6% of the SCRs cannot be shown because of the extremely small percentages spread across less common types of artifacts or groupings. Based on Figure 5 the main findings with respect to the types of fixed artifacts are:

- **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 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 requirements 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.** Around 3% of the all SCRs resulted in changes of some design artifact(s) or combination of design artifacts with other artifacts (which is half of the SCRs not shown in Figure 5). This result to some extent may be attributed to the fact that the mission did not have detailed design documents, and therefore when fixes affected only a few lines of code the design documents did not need to be updated.
- **Work-arounds were sometimes used to prevent failures from occurring / reoccurring**. The analysis of the Fixed artifact showed that less than 2% of SCRs identified Notes and/or Waivers, which seemed surprisingly low. In discussion with the project personnel it appeared that another type of note document, which was not listed in the affected artifact field of the CNs, existed. These note documents were stored in the change tracking system for a sole purpose of tracking work-arounds. Further analysis showed that, in addition to 2% work-arounds identified in the CNs' affected artifact field, an additional 26% of the SCRs considered in this paper were associated with this different type of note document. Closer exploration showed that SCRs linked to these note documents were often linked with CNs that identified fixes, which suggests that work-arounds may have been implemented immediately to avoid failures, while the fixes were being worked out and implemented.

## 5. HOW ARE THE TYPES OF FIXED ARTIFACTS ASSOCIATED WITH FAULT TYPES AND PRE-RELEASE / POST-RELEASE OCCURRENCE?

In this section we explore how the types of fixed software artifacts are related to the types of faults that caused individual failures and whether the faults surfaced pre-release or post-release. As discussed in Section 3 *Fault type* and *Pre-release / Post-release occurrence* are identified in each non-conformance SCR, while the *Fixed software artifact* is identified in each CN, together with other attributes of the fixes. The reader is reminded that each SCR can be linked to one or more CNs, that is, the project kept track of all fixes associated with each individual failure.

In addition to descriptive statistics, this section includes inferential statistics aimed at testing if Fault type and Fixed artifact attributes are associated, as well as if the distribution of the types of fixed attributes are different for pre-release and post-release failures. Because the these attributes are categorical (i.e., given on a nominal scale), contingency tables are used and the $\chi^2$ statistics are calculated on a stratified random sample to test the null hypotheses. The association between Fault type and Fixed artifact attributes was measured using the contingency coefficient $C$ as a measure of correlation [30] since it is uniquely useful in cases when the attributes are categorical. (Note that the widely known Pearson or Spearman correlation coefficients cannot be used when the random variables are given on a nominal scale.) Details about $\chi^2$ statistics for contingency tables and contingency coefficient $C$ are given in Appendix A.

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

### 5.1. *RQ4: Are the type(s) of Fixed software artifacts related to the Fault type that caused the failure?*

Intuitively, one might expect that the Fault type responsible for causing a (potential or actual) failure is likely to be correlated with the type(s) of Fixed artifact(s). For example, a requirement fault is likely to lead to a change in requirements document. However, the association is unlikely to be trivial because for example a requirement fault may also lead to changes in the source code if it is discovered after the code was implemented.

To explore the association between the fault types that caused individual failures and the types of artifacts affected by fixes, we built a the contingency table of the Fault type and Fixed artifact(s) (see Table III), which for each major Fault type presents the frequency counts (and percentages of the total number of SCRs in brackets) of the types or groups of types of Fixed artifacts. The same information is graphically depicted in Figure 6. Based on the Table III and Figure 6 the following observations were made:

- **The majority of requirements faults were fixed by changing requirements only.** 59% of failures caused by requirement faults (i.e., around 20% of the total non-conformance SCRs) resulted in changes in requirement artifacts and no other type of artifact. This suggests that most requirement faults were caught before the code for the 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 requirement faults were detected later in the life cycle, when the code was already implemented. Interestingly, some requirements faults were fixed by changing the code only (i.e., 10% of failures caused by requirement faults, or about 3% of the total non-conformance SCRs). This may indicate situations in which the requirement was unclear but the code was the only fixed artifact because lower level implementation details were not included in requirements document.
- **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 only in code and an additional 33% (i.e, 2% of the total non-conformance SCRs) led to changes in code and at least one other type of artifact. According to project personnel, this can be explained by the fact that this NASA mission did not use very detailed design documents, and thus design decisions were sometimes made and implemented at the code level.
- **The majority of coding faults were fixed by changing the code only.** In particular, 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 fixed artifacts. 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 and at least one other type of artifact, with fixing both code and requirements documents contributing to 13% (i.e., slightly above 4% of the total non-conformance SCRs). Interestingly, around 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 for NASA missions, which do things for the first time and hence experience on-going discovery of requirements. Accordingly to the project personnel in those cases the code was implemented on the fly and then an SCR was written to update the requirements to match the code.
- **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, which at first sight seemed very unusual. However, a thorough exploration of the data showed that this phenomenon was due to the fact that this NASA mission relies 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. Thus, some of the integration

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

faults were fixed by changing these program, instrument, and command lists. Clearly, an emphasis should be put on defining, reviewing, and testing these commands and values which potentially could lead to less integration faults.

- **Uncommon fault types most often led to changes in code only.** Fault types such as I/O problems and simulation problems, which were rare (e.g., each accounted for less than 1% of the non-conformance SCRs) were grouped into the *other* fault type category. Figure 6 shows that these fault types were likely to require changes to the code only.

Obviously, the Fault type and Fixed artifact attributes are related, but their associations are not trivial. $\chi^2$ was computed to quantify this association for the values given in Table III ($\chi^2 = 609.21$), which led to contingency coefficient $C = 0.57$, with a maximum value $C_{max} = 0.91$. The normalized contingency coefficient $C^* = C/C_{max} = 0.63$ indicates that the correlation between Fault type and Fixed attribute is strong and statistically significant at 0.05 significance level.

Based on the analysis of the association between Fault types and Fixed artifacts, we point out the following suggestions that may contribute towards 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 which 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 only and may be avoided by compiling a checklist with common coding errors that can be used both during development and testing. Similar effort in Lucent Technologies [21] 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%.
- **Some faults may have been detected earlier.** Based on the results presented in Figure 6 and Table III, 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. Potentially, these requirements faults may have been detected as early as the requirements phase, before the code implementing these requirements even existed. Further, 18% of failures linked solely to changes in the code (i.e., close to 6% of the total non-conformance SCRs) were actually caused by requirement faults and design faults. Hence, some of these specific problems may have been caught earlier in the life cycle. Although we recognize the potential for detecting some faults earlier, it should be noted that in case of NASA missions earlier detection cannot always be achieved in practice as some requirements will always remain to be discovered later in the life cycle because NASA is often exploring the unknown.

### *5.2. RQ5: Are types of fixed artifacts different for pre-release and post-release failures?*

Post-release failures, although rare, entail careful analysis because typically they are more critical and have potential to cause a catastrophic mission failure. In this section we explore if the types of fixed artifacts are different for pre-release and post-release failures. For that purpose, Table IV shows the percentage of failures discovered pre-release and post-release that led to fixing the common types and groups of 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) than the sample size of the potential failures prevented by fault detection and removal activities prior to release (i.e., 1,215).

Based on the results shown in Table IV, the following observations were made:

- **Half of the post-release failures resulted solely to changes in the source code, which is significantly higher than around 32% of pre-release failures that led to changes only in the code.** Even more, a significantly higher percentage of post-release failures than pre-release failures resulted in changing both the code and supporting files. These observations are

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

not surprising considering that 53% of post-release failures were actually caused by coding faults. However, the fact that a significant percentage of post-release failures are addressed by changing only the code or code and supporting files, shows that the number of post-release failures can be significantly reduced by avoiding and/or detecting coding faults earlier in the life cycle, for example by implementing a coding checklist that can be used during development and testing.

- **Similar percentages of post-release and pre-release failures led to changes in both code and requirements artifacts** (i.e., 14.24% and 11.90%, respectively). This observation proves the fact that system requirements continued to evolve during operations in this system, largely due to the dynamic nature of the environment and the complexity of the operational scenarios. Similar observations related to continuous discovery of requirements during operation have been made for other NASA missions [20], [31].
- **Significantly less post-release than pre-release failures are tied to changes in requirements only.** This result is expected, having in mind that in case of post-release failures the code exists and it is likely that it will be fixed, sometimes in combination with other artifacts. The remaining close to 12% of post-release failures (i.e., 0.4% of the total number non-conformance SCRs), which required changes to requirements documents only, are explained by the fact that for this specific mission some requirements artifacts contained detailed command sequences which were read directly during flight operations, and therefore had to be fixed as result of some post-release failures.

To formally confirm that the distribution of the types of fixed artifacts is different for pre-release and post-release failures, the $\chi^2$ test for contingency tables was used. The computed value of $\chi^2 = 26.96$ confirmed that the distributions of Fixed artifacts were different for pre-release and post-release failures at significance level of 0.05.

## 6. THREATS TO VALIDITY

In this paper we analyzed the characteristics of software fixes based on empirical data collected during a period of almost ten years from a large-scale NASA mission. In general, 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.

As in case of any empirical study, there were some threats to validity that needed to be considered and addressed. These threats to validity, grouped in four categories (i.e., construct validity, internal validity, conclusion validity, and external validity), are discussed next.

**Construct validity** refers to the degree to which a test measures what it claims to be measuring. One obvious threat to construct validity is related to not having sufficiently defined constructs before they are translated to metrics. Therefore, the definitions for faults, failures, and fixes as they are used in this work are provided at the very beginning of this paper. Additionally, in order to characterize fixes in a way that would be meaningful to the project as well as the larger software engineering community, attributes and attribute values were carefully chosen. Mapping data fields in the SCRs and CNs to fault, failure, and fix attributes was done based on a careful initial investigation. For each attribute considered in this work, definitions of the fields and the associated values were reviewed and classified into larger categories in collaboration with the mission domain experts.

In terms of Fixed components the task was simple since each CN identified only one component that was changed and in general each SCR can be linked to zero, one or more CNs. It should be noted that in a small number of cases (i.e., for around 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. However, 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, these SCRs were not excluded from the analysis.

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

Identifying the type of fixed artifact was more difficult because the corresponding field in the CN was an English text with varying levels of details. For example, in some cases specific requirements documents or filenames were given, but in other cases only high-level references such as 'code' were recorded. This field had 105 unique entries across the 2,620 CNs written for the 1,257 SCRs considered in this paper. By manual exploration of the values and through detailed discussions with the mission personnel, six categories were defined for the types of fixed artifact attribute: Requirements documents, Design documents, Code files, Supporting files, Tools, and Notes/Waivers. Then each unique value was mapped to one of these six categories.

It should be noted that some of the SCRs still had an 'open' status at the time the data dump was made available to us. To eliminate the threat to validity due to yet to be implemented CNs, the dataset was limited to 'closed' SCRs for which all fixes have been implemented and approved. Both the open and closed SCRs were included only in the study of the relationship between the Failed component and Fixed component(s), which allowed us to explore larger dataset and led to interesting observations with respect to the distribution of software fixes across the software architecture. The fact that the percentage of SCRs that led to fixes outside the Failed component was consistent across both datasets (i.e., dataset with only closed SCRs and dataset with both open and closed SCRs) indicates that the trend of fixes spreading across components is likely to continue.

**Internal validity** threats are concerned with influences that can affect the independent variables and measurements without researchers knowledge. Data quality is one of the biggest threats to internal validity. Therefore, the project personnel was consulted to ensure that the meaning of specific data fields and their values, as well as the perspective of the analysts who populated the data fields were well understood.

SCRs and CNs were filled out by analysts who worked on different parts of the mission, from different locations, using various methods. However, the fact that there is a joint review panel, which is responsible for handling, reviewing, and tracking SCRs and CNs, ensures the quality of the data across the board. To limit threats to validity due to lack of domain knowledge, input and feedback from the project personnel was solicited throughout the course of this work.

To further ensure the data quality, SCRs that were withdrawn, tagged as duplicates, or tagged as operator errors were removed for the dataset. Studying operator errors, which is out of the scope of this paper, is an interesting topic for future work. It should be noted, however, that operator errors in this NASA mission were infrequent; only 2.8% of all SCRs (including SCRs with open and closed status, as well as SCRs linked to other components not considered in this paper) were attributed to operator errors. Therefore, a future study of operator errors would require considering more than one mission to allow for large enough sample.

**Conclusion validity** is concerned with the ability to draw correct conclusions. The most obvious threat to conclusion validity is using statistical tests in cases when their underlaying assumptions are violated. Specific care was taken in this paper to ensure that the appropriate statistical techniques were selected to test the hypotheses. Thus, statistical tests were chosen based on the measurement scales of the attributes and the validity of the underling assumptions of the tests. Also, all statistical tests were conducted on random samples.

The analysis in this paper was conducted at the component level rather than at the file level because the vast majority of data was kept at component level. Whether the findings presented in the paper would apply to file level is, in general, an open research question. However, the fact that software fixes are spread has been shown to be valid at file level as well for a small subset of the data used in this paper (i.e., 404 failures tied to changes in source code only) analyzed in [18].

The analysis for all research questions, except for RQ2, was based on the dataset consisting of 'closed' SCRs. To ensure that some phenomena were not overlooked by omitting the SCRs with an 'open' status the complete analysis presented in the paper was repeated on the larger sample, which included both the open and closed SCRs. All trends and observations based on the larger dataset were consistent with the ones based only on the closed SCRs presented in the paper.

Another threat to conclusion validity is the fact that the sample size of the post-release SCRs was small, even though our dataset spanned over ten years period. Specifically, only 3% of the non-conformance SCRs were reported on-orbit, which is an intrinsic characteristics of high-quality

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

operational software. As a results, the comparisons of pre-release and post-release non-conformance SCRs were made on samples with vastly different sizes. This is a threat to validity to almost all studies that are focused on analysis of post-release failures. For clarity, the percentages within each class, as well as within the entire sample are noted throughout the paper.

Last but not least, to ensure accurate interpretation of the results, the claims made in this paper were reviewed by NASA personnel. In many instances, multiple cycles of analyzing, presenting, reviewing, re-analyzing and updating the results occurred. For example, based on the fact that less than 2% of affected artifacts contained references to Notes and/or Waivers, we originally believed that work-arounds were very rare. However, following a suggestion from project personnel an additional type of note document was discovered and explored showing that work-arounds were actually used more commonly.

**External validity** is related to the ability to generalize the results. This study is based on data collected by a large NASA mission (over two millions lines of code in over 8,000 files), developed at several locations, by different teams, during almost ten years of development and operation. Although these facts allow for some degree of external validation, the results may not be valid across other software products, from different domains. However, the fact that the definitions for all attributes and attributes values are provided will allow future studies to explore the external validity of the findings and conclusions reported in this paper.

## 7. CONCLUDING REMARKS

This paper is focused on characterizing several different aspects of software fixes, an area of software quality assurance that has not received much attention. The results are based on data collected over a decade from a large-scale, safety-critical NASA mission.

A unique characteristic of the work presented in this paper is the fact that the analysis closed the loop from faults (i.e., root causes) that caused the (potential or actual) failures to the changes made to fix the faults and prevent failures from (re)occurring. Unlike related works in this area, which analyzed 'bug fixes' made to software units (e.g., files, components, or packages) cumulatively due to all observed failures or limited the scope of fixes to the first or primary fix only, we specifically explored the phenomenon of multiple changes made to fix an actual or potential failure reported in individual software change requests. Even more, unlike related works which typically addressed the fixes made only in one software artifact – source code – we analyzed the fixes made to different software artifacts (i.e., requirements documents, design documents, code, supporting documents, and other). Therefore, it was possible to present a comprehensive view on how fixes spread throughout the software architecture and across different types of software artifacts.

The analysis presented in this paper led to discovery of common patterns, as well as atypical behaviors. Several interesting findings and their potential implications are briefly summarized below.

**For a significant number of failures fixes were spread across multiple components and/or affected several types of software artifacts.** In particular, 15% of the (potential or observed) failures were associated with multiple components. Furthermore, 26% of failures were associated with fixing multiple software artifacts. These results indicate the danger of injecting (i.e., seeding) only simple semantic faults, localized to a single component, which affects several research areas in software testing, such as fault-based testing, mutation testing, and comparative evaluation of testing strategies. Furthermore, the assumption that each failure is caused by a single component, which is a widely used in component-based software reliability models, appears to be an oversimplification of the real phenomenon.

**The spread of fixes across components is greatly affected by the software architecture.** In particular, fixes that spread outside the Failed component against which the SCR was written were typically located within the same architectural group. Cases when fixes were made outside the group of the Failed component were mostly due to interconnected groups that perform integrated operations. Information about groups of components that are typically changed together is very useful for planning integration testing, as well as for addressing currently open or future SCRs

Copyright © 2012 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

by suggesting where fixes are likely to occur. Additionally, exploring the correlation between software architecture and implemented fixes can help identifying problems such as high coupling among components, which may be addressed, for example, by refactoring or redesigning these more problematic parts of the software system in future releases.

This study identified several directions for cost effective improvement of software quality:

- *Integrating requirements engineering throughout the life cycle, including post-release.* Our results showed that requirements faults are significant source of failures, and that many fixes affect the requirements documents. The finding that new software requirements continue to emerge on-orbit indicates that the process of requirements specification, analysis, and verification continues post-release.
- *Compiling lists of common coding faults.* A significant number of failures, including half of the post-release failures, led to fixes in the source code only. Compiling a list of common coding faults for inclusion in inspections, reviews, and test cases is of practical value for future releases of a given system or subsequent, similar systems.

The results presented in this paper indicate that, for long-lived, critical systems, analysis of software faults, failures, and fixes provides useful mechanisms for maintaining the deployed systems, as well as for improving the efficiency of detecting faults, revealing failures, and implementing future fixes in the incremental software development process. The software used in NASA missions shares features with many other high integrity domains: operation in environments that are only partially understood, use of novel technologies, highly interactive subsystems, timing constraints for correct operations, and a high degree of autonomy. This clearly shows that mining bug and/or change tracking systems is helpful for building a knowledge base that can be reused on other, similar systems.

Learning more about how faults are fixed and thus how failures are prevented is a necessary part of understanding how quality assurance methods can be improved to result in more efficient avoidance, detection, and correction of faults and thus better products. Our future work is focused on analyzing the effort spent implementing the fixes characterized in this paper, as well as on conducting similar analysis on other case studies.

## APPENDIX A: BACKGROUND ON THE CONTINGENCY CORRELATION COEFFICIENT

In this paper, the contingency coefficient $C$ is used as a measure of correlation, which unlike the more frequently used Pearson and Spearman correlation coefficients can be used in cases when the information about at least one of the random variables is categorical (i.e., given on a nominal scale).

The contingency coefficient $C$, which measures the correlation between two categorical random variables $X$ and $Y$ (with $n$ and $m$ categories, respectively), is computed as follows. First, an $m$ x $n$ contingency table is built using the observed frequencies for each pair of categories. Then, the standard $\chi^2$ statistic which can be approximated by a chi-square distribution with $(m-1)(n-1)$ degrees of freedom is computed. When used for contingency comparisons, the chi-square test is a non-parametric test, since it compares entire distributions rather than parameters of distributions.

Once the $\chi^2$ statistic has been calculated, the probability under the null hypothesis of obtaining a value as large as the calculated $\chi^2$ value is determined. If the probability is equal to or less than the significance level $\alpha$ ($\alpha = 0.05$ in this paper), the null hypothesis is rejected. Since, the null hypothesis states that there is no relation between the two variables (i.e., any correlation observed in the sample is due to chance) rejecting the null hypothesis implies that the distribution of failures across the $m$ categories differs significantly for the $n$ samples, and suggests that there is some correlation between the two variables. The contingency coefficient $C$, which measures the extent of the correlation between the two variables, is calculated as:

$$C = \sqrt{\frac{\chi^2}{N + \chi^2}} \tag{1}$$

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

where $N$ is the total number of observations and the value of $\chi^2$ statistics is computed based on the contingency table [30].

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

$$C_{max} = \sqrt[4]{\frac{m-1}{m} \cdot \frac{n-1}{n}} \tag{2}$$

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 (3), one ensure that the range will be between 0 and 1, and hence, $C^*$ values for different sized tables can be compared [32].

$$C^* = C/C_{max}. \tag{3}$$

ACKNOWLEDGEMENTS

This work was funded in part by grant from the NASA Independent Verification and Validation Facility, Fairmont, West Virginia. Part of Katerina Goseva-Popstojanova's work was supported by the National Science Foundation grant CCF-0916284 with funds from the American Recovery and Reinvestment Act of 2009.

The authors thank the NASA personnel for their invaluable 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, and Sarma Susarla. This work would not have been possible without their input and feedback. We also thank the anonymous reviewers for their comments and suggestions

Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

1. Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* January-March 2004; **1**(1):11–33.
2. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* 1990.
3. Systems and Software Engineering Vocabulary. *ISO/IEC/IEEE 24765* 2010.
4. Eldh S, Punnekkat S, Hansson H, Jonsson P. Component testing is not enough - a study of software faults in telecom middleware. *Testing of Software and Communicting Systems, LNCS 4581*, 2007; 74–89.
5. Zimmermann T, Premraj R, Zeller A. Predicting defects for Eclipse. *3rd International Workshop on Predictor Models in Software Engineering*, 2007; 9–.
6. Ostrand TJ, Weyuker EJ. The distribution of faults in a large industrial software system. *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002; 55–64.
7. Goseva-Popstojanova K, Hamill M, Perugupalli R. Large empirical case study of architecture-based software reliability. *16th IEEE International Symposium on Software Reliability Engineering*, 2005; 43–52.
8. Fenton NE, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering* Aug 2000; **26**(8):797–814.
9. Andersson C, Runeson P. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering* May 2007; **33**(5):273–286.
10. Ostrand TJ, Weyuker EJ, Bell RM. Predicting the location and number of faults in large software systems. *IEEE Transactions Software Engineering* Apr 2005; **31**(4):340–355.
11. Arishold E, Briand L, Johannssen E. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *The Journal of Systems and Software* August 2009; **83**:2–17.
12. Devine T, Goseva-Popstajanova K, Krishnan S, Lutz RR, Li JJ. An empirical study of pre-release software faults in an industrial product line. *IEEE International Conference on Software Testing, Verification and Validation*, 2012; 181–190.
13. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* November-December 2012; **38**(6):1276–1304.
14. Offutt AJ, Hayes JH. A semantic model of program faults. *1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1996; 195–200.
15. Goseva-Popstojanova K, Trivedi KS. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation* Jul 2001; **45**(2-3):179–204.
16. Christmansson J, Chillarege R. Generation of an error set that emulates software faults based on field data. *26th International Symposium on Fault-Tolerant Computing*, 1996; 304–313.

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

17. Duraes JA, Madeira HS. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering* November 2006; **32**(11):849–867.
18. Hamill M, Goseva-Popstojanova K. Common trends in fault and failure data. *IEEE Transactions on Software Engineering* July-August 2009; **35**(4):484–496.
19. Leszak M, Perry D, Stoll D. Classification and evaluation of defect in a project retrospective. *Journal of Systems and Software* April 2002; **61**:173–187.
20. Lutz RR, Mikulski IC. Empirical analysis of safety-critical anomalies during operations. *IEEE Transactions on Software Engineering* Mar 2004; **30**(3):172–180.
21. Yu WD. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal* April-June 1998; **3**(2):3–21.
22. Sherriff M, Lake M, Williams L. Prioritization of regression tests using singular value decomposition with empirical change records. *18th IEEE International Symposium on Software Reliability*, 2007; 81–90.
23. Canfora G, Cerulo L. How software repositories can help in resolving a new change request. *Workshop on Empirical Studies in Reverse Engineering*, 2005.
24. Canfora G, Cerulo L. Supporting change request assignment in open source developement. *2006 ACM Symposium on Applied Computing*, 2006; 1767–1772.
25. Abdelmoez W, Shereshevsky M, Gunnalan R, Ammar H, Yu B, Bogazzi S, Korkmaz M, Mili A. Quantifying software architectures: An analysis of change propagation probabilities. *3rd ACS/IEEE International Conference on Computer Systems and Applications*, 2005; 124–131.
26. Zimmermann T, Nagappan N. Predicting subsystem failures using dependency graph complexities. *18th IEEE International Symposium on Software Reliability Engineering*, 2007; 227–236.
27. Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. *30th International Conference on Software Engineering*, 2008; 531–540.
28. Bird C, Nagappan N, Devanbu P, Gall H, Murphy B. Putting it all together: Using socio-technical networks to predict failures. *20th IEEE International Symposium on Software Reliability Engineering*, 2009; 109–119.
29. Chillarege R. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, Lyu MR (ed.), IEEE Computer Society Press and McGraw-Hill Book Company: New York, 1996.
30. Siegel S, Castellan NJ Jr. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, 1988.
31. Lutz RR, Mikulski IC. Ongoing requirements discovery in high-integrity systems. *IEEE Software* Mar 2004; **21**(2):19–25.
32. Blaikie N. *Analyzing Quantitative Data*. SAGE Publication Ltd: London, 2003.

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

Table I. Distribution of the non-conformance SCRs with 'closed' status and associated CNs across 21 components

| Component | # of files | Size in LOC | # of closed non-conf SCRs | # of CNs |
|---|---|---|---|---|
| 1 | 207 | 48,910 | 179 | 252 |
| 2 | 200 | 60,386 | 8 | 14 |
| 3 | 287 | 78,854 | 4 | 3 |
| 4 | 228 | 46,657 | 4 | 6 |
| 5 | 269 | 71,953 | 4 | 6 |
| 6 | 321 | 92,978 | 15 | 18 |
| 7 | 289 | 34,938 | 33 | 43 |
| 8 | 270 | 43,012 | 44 | 86 |
| 9 | 25 | 21,266 | 1 | 9 |
| 10 | 356 | 83,134 | 8 | 10 |
| 11 | 444 | 103,145 | 24 | 36 |
| 12 | 277 | 55,475 | 11 | 17 |
| 13 | 599 | 57,800 | 35 | 72 |
| 14 | 280 | 27,940 | 28 | 53 |
| 15 | 84 | 38,882 | 23 | 44 |
| 16 | 169 | 47,541 | 44 | 70 |
| 17 | 587 | 147,520 | 112 | 158 |
| 18 | 552 | 174,614 | 104 | 169 |
| 19 | 747 | 164,419 | 102 | 197 |
| 20 | 1368 | 737,504 | 412 | 1157 |
| 21 | 415 | 74,618 | 62 | 76 |
| Total | 8,071 | 2,211,546 | 1,257 | 2,496 |

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

Table II. Basic statistics of the number of CNs linked to individual failures for each component

| 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 |

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

Table III. Frequency counts of Fixed artifacts across major Fault types

| **Fault Types** | **Fixed Artifact(s)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Reqs. | Reqs. Design & Code | Reqs. &Code | Reqs. Code & Supporting files | Code | Code & Supporting files | Other | Total |
| Requirements | 251 (19.97%) | 9 (0.72%) | 86 (6.84%) | 16 (1.27%) | 41 (3.26%) | 6 (0.48%) | 20 (1.59%) | **429 (34.13%)** |
| Design | 4 (0.32%) | 1 (0.08%) | 10 (0.80%) | 4 (0.32%) | 33 (2.63%) | 8 (0.64%) | 9 (0.72%) | **69 (5.49%)** |
| Coding | 27 (2.15%) | 4 (0.32%) | 52 (4.14%) | 9 (0.72%) | 281 (22.35%) | 29 (2.31%) | 22 (1.75%) | **424 (33.73%)** |
| Integration | 199 (15.83%) | 2 (0.16%) | 23 (1.83%) | 9 (0.72%) | 26 (2.07%) | 14 (1.11%) | 6 (0.48%) | **279 (22.20%)** |
| Other | 9 (0.72%) | 0 (0.00%) | 7 (0.56%) | 1 (0.08%) | 29 (2.31%) | 1 (0.08%) | 9 (0.72%) | **56 (4.46%)** |
| Total | **490 (38.98%)** | **16 (1.27%)** | **178 (14.16%)** | **39 (3.10%)** | **410 (32.62%)** | **58 (4.61%)** | **66 (5.25%)** | **1,257(100.00%)** |

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls


DOI: 10.1002/stvr

Table IV. Comparing the percentages of fixed artifacts for pre-release and post-release failures

| Fixed artifacts | Percentage of pre-release failures | Percentage of post-release failures |
|---|---|---|
| Code only | 32.03 | 50.00 |
| Code and supporting files | 4.20 | 16.67 |
| Requirements only | 39.92 | 11.90 |
| Requirements and code | 14.24 | 11.90 |
| Requirements, design, and code | 1.32 | 0.00 |
| Requirements, code, and supporting files | 3.13 | 2.38 |
| Other | 5.19 | 7.14 |

Copyright © 2012 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

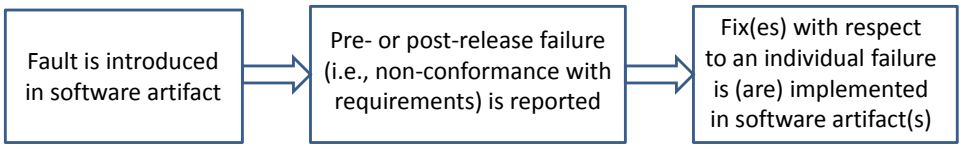


Figure 1. The cause-effect relationship among faults, failures and fixes

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

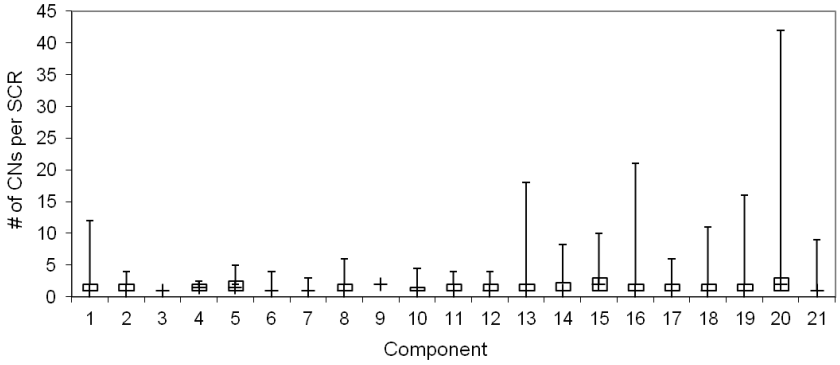


Figure 2. Box plots showing the number of CNs associated with individual failures, for each component

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

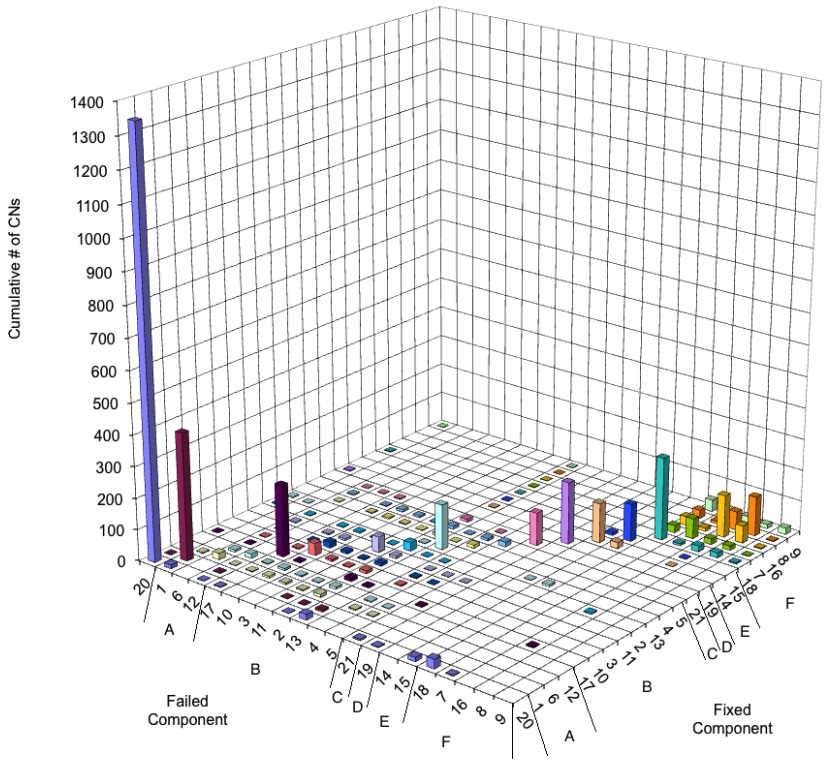


Figure 3. The cumulative number of CNs associated to each components, linked to the cumulative number of SCRs associated with the Failed component

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

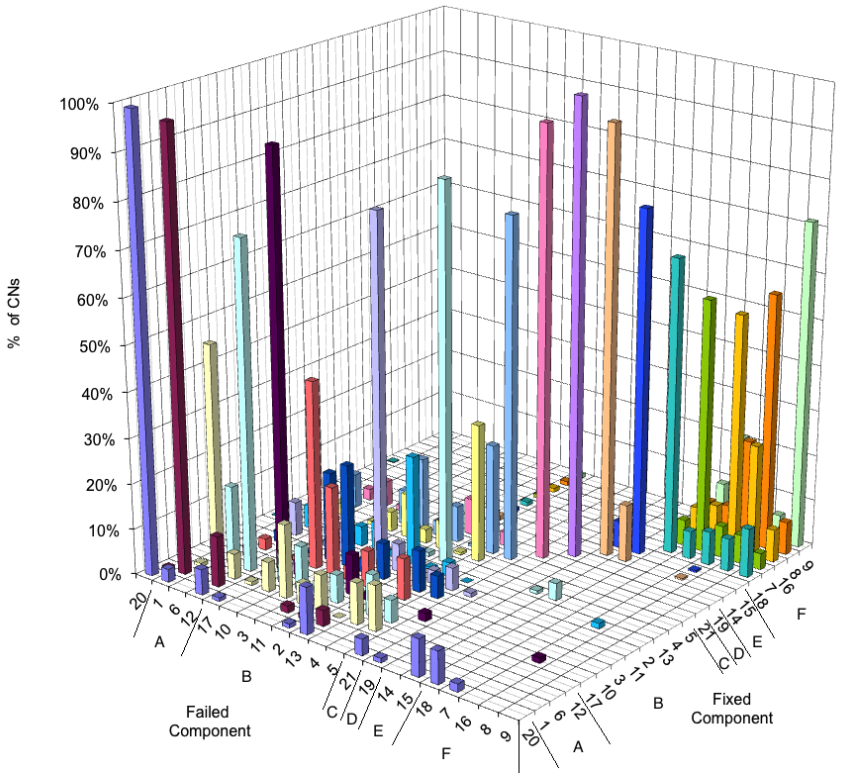


Figure 4. The distribution of CNs across Fixed components (in percentages), for each Failed component

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

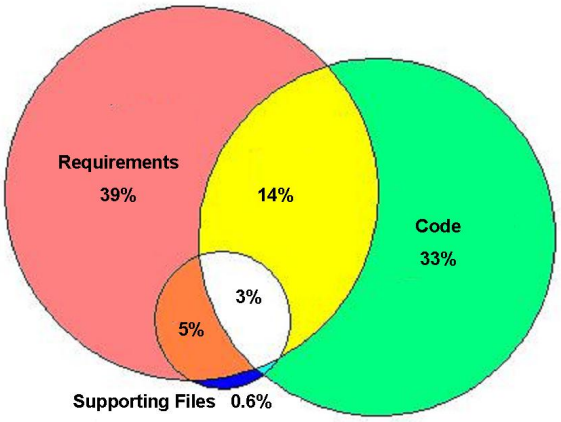


Figure 5. Venn diagram representing the types of artifacts and combinations of artifacts changed to prevent individual failures from reoccurring. (Around 6% of non-conformance SCRs cannot be shown because of the extremely small percentages spread across less common types of artifacts.)

Copyright © 2012 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

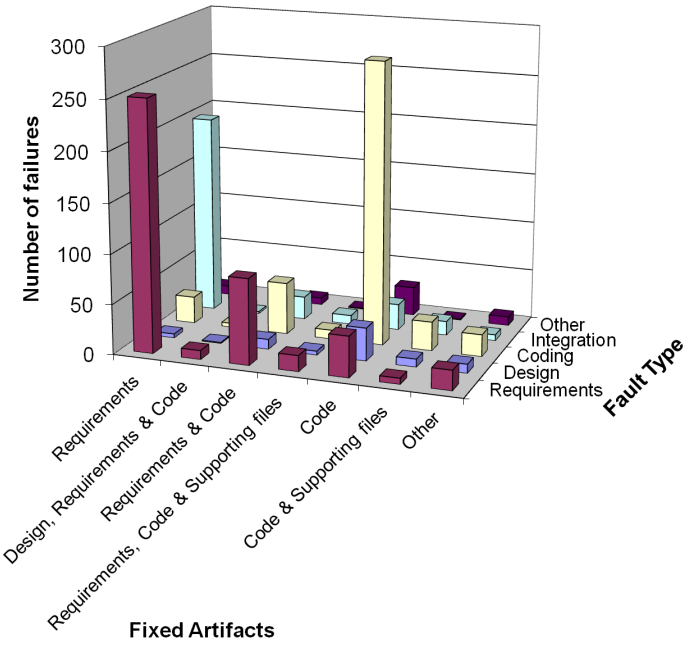


Figure 6. Frequency counts of Fixed artifacts across major Fault types

Copyright © 2012 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr