Software Quality Journal manuscript No.
(will be inserted by the editor)

Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system

Maggie Hamill · Katerina Goseva-Popstojanova

the date of receipt and acceptance should be inserted later

Abstract Many papers have been published on analysis and prediction of software faults and/or failures, but few established the links from software faults (i.e., the root causes) to (potential or observed) failures and addressed multiple attributes. This paper aims at filling this gap by studying types of faults that caused software failures, activities taking place when faults were detected or failures were reported, and the severity of failures. Furthermore, it explores the associations among these attributes and the trends within releases (i.e., pre-release and post-release) and across releases. The results are based on the data extracted from a safety-critical NASA mission which follows an evolutionary development process. In particular, we analyzed 21 large-scale software components, which together constitute over 8,000 files and millions of lines of code. The main insights include: (1) Only a few fault types were responsible for the majority of failures pre-release and post-release, and across releases. Analysis and testing activities detected the majority of failures caused by each fault type. (2) The distributions of fault types differed for pre-release and post-release failures. (3) The percentage of safety-critical failures was small overall and their relative contribution increased on-orbit. (4) Both post-release failures and safety-critical failures were more heavily associated with coding faults than with any other type of faults. (5) Components that experienced high number of failures in one release were not necessarily among high failure components in the subsequent release. (6) Components that experienced more failures pre-release were more likely to fail post-release, overall and for each release.

Keywords Software quality assurance · Fault type · Detection activity · Failure severity Evolutionary development.

1 Introduction

The size, complexity and capability of software systems have grown tremendously over the years. As software usage expands in many areas, and modern society relies more and more on proper functioning of software systems, quality assurance becomes an even greater concern. Deficiencies

This work was done while Maggie Hamill was affiliated with West Virginia University.

M. Hamill
Department of Computer Science and Electrical Engineering, Northern Arizona University, Flagstaff, AZ
E-mail: Margaret.Hamill@nau.edu

K. Goseva-Popstojanova
Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV
Tel: +1-304-293-9691
Fax: +1-304-293-8602
E-mail: Katerina.Goseva@mail.wvu.edu

in software quality often result in costly emergency fixes, damage to brand's reputation, and sometimes in loss of human life. Based on the estimates made by the National Institute of Standards and Technology (NIST), the US economy loses $60 billion annually in costs associated with developing and distributing patches that fix software faults, as well as cost from lost productivity due to computer malware and other problems caused by software faults (Zhivich and Cunningham, 2009). It is estimated that users incur about 64% and developers 36% of the total cost. This paper focuses on characterizing software faults and failures, so that quality assurance efforts can be tailored to improve efficiency of detecting faults and preventing failures and thus lead to more reliable and less costly systems.

Terms related to software quality assurance are often used inconsistently. Therefore, we start with the definitions of a fault and failure, which are adapted from ISO/IEC/IEEE 24765 (2010). A *failure* is the inability of a system or component to perform its required functions within specified requirements. On the other hand, 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 can be introduced at any phase of the software life cycle and can be tied to any software artifact (e.g., requirements, design, source code). It should be noted that every fault does not correspond to a failure since the conditions under which the fault(s) would result in a failure may never be met. Much of the related work uses the term defect (e.g., (Biyani and Santhanam, 1998; Chillarege et al., 1992; Leszak et al., 2002; Shihab et al., 2010; Zimmermann et al., 2007)), which sometimes refers to faults or to pre-release faults only, and other times is used to refer collectively to faults, failures, and anomalies. The term bug is typically used to refer to faults in the source code. Throughout this paper, for clarity, we strictly use the terms fault and failure.

The benefits of analyzing software faults and failures are quite obvious. Unlike other engineering areas that have a long tradition of learning from past failures, most software industries do not pay enough attention to understand the typical faults that exist in their software (Eldh et al., 2007). One of the obvious reasons is that fault and failure data are rarely readily available or in a form that allows straightforward data extraction and analysis. Furthermore, while significant amount of empirical studies have been focused on studying software faults and/or failures very few works have tied the faults to the failures that they caused and studied the associations among faults, failures, and other relevant attributes. This is mainly due to the fact that the traceability to the root causes of reported failures is typically missing. Extracting information from version control system can be helpful, but is far from trivial because often 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) (Ostrand and Weyuker, 2002; Goseva-Popstojanova et al., 2005). In addition, studies were often focused only on faults in the source code due to the fact that keeping track of faults in other software artifacts (e.g., requirements, design documents) does not seem to be a common practice, especially among open source projects, which are frequently used as case studies in the software quality assurance research.

The paper presents a case study based on the flight software of a large, safety-critical NASA mission. The analyzed data collectively 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 evolutionary (i.e., iterative) development process through multiple releases, by teams at several locations. The mission is still active and requires sustained engineering.

As is the case with most change tracking systems, the change tracking system for the NASA mission was not designed with fault and failure analysis in mind and the data was not readily available. However, with support from the project's independent verification and validation personnel we were able to extract the relevant data. Furthermore, the processes and procedures followed by the NASA mission with respect to tracking changes provide assurance of the data validity and accuracy. Since we were interested in the characterization of faults and failures, our analysis was focused on the Software Change Requests (SCRs) that were made because the software failed to conform to some requirement(s) (i.e., non-conformance SCRs). These SCRs represent either potential or observed failures reported throughout the life of each component. In other words, while some of the failures were reported and addressed during development and testing, others occurred on-orbit (i.e., post-release). Our analysis is focused on *types of software faults* (e.g., requirements faults, design faults, coding faults, integration & interface faults), *detection activities* (e.g., inspections, analysis, testing, on-orbit), and *failure severity* (i.e., safety-critical and non-critical failures).

The objective of our work is to contribute towards better understanding of fault types, detection activities and severity of software failures, including their associations and trends within and across releases, and furthermore, based on empirical evidence, to assist the developers and verification and validation personnel in improving the product quality in a cost-efficient way. Focusing on high-priority failures (i.e., post-release, safety-critical and those associated with the dominating fault types) and their association with detection activities provides insights that can be used by practitioners to guide the allocation of verification and validation techniques aimed at avoiding such failures and their consequences. Furthermore, studying software failure trends across releases and within each release (i.e., pre-release and post-release) can help practitioners to identify software components that require more verification and validation and may need fault-tolerance mechanisms. Preventing and tolerating high-priority failures is of utmost importance for any software system, and especially for safety-critical systems that operate with high degree of autonomy.

Specifically, in this paper we address the following research questions:

RQ1: Are certain activities more likely to detect certain fault types? Which types of faults lead to post-release failures?
RQ2: Which activities are more likely to reveal safety-critical failures?
RQ3: Are certain fault types more likely to cause safety-critical failures?
RQ4: Does the contribution of dominating fault types change as the software matures across releases?
RQ5: Is there an association between the number of failures reported in release n and release $n+1$ (i.e., across releases)?
RQ6: Is there an association between the number of failures reported pre-release and post-release (i.e., within individual release)?

It appears that the research questions RQ1 - RQ4 have not been previously explored in the literature. RQ5 and RQ6 have been explored in the past and therefore this part of our work can be considered as an independent replication (Shull et al., 2008; Kitchenham, 2008) whose goal is to explore these phenomena in a different context. This type of replication, which sometimes is called conceptual (Shull et al., 2008), is important because it allows the same phenomena to be studied under different conditions and thus explores the generalization of the results (Shull et al., 2008; Kitchenham, 2008; Grbac et al., 2013; Yin, 2014). When the context or environmental factors of the replication are different from those of the original study, then the replication can contribute some confidence, to both researchers and practitioners, that the effect is not limited to one particular setting (Shull et al., 2008; Kitchenham, 2008).

The main contributions of this paper are as follows:

– 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 associations among types of faults that caused failures, the detection activities taking place when the faults were detected and/or failures were reported, and the severity of failures. Related works treated only particular attributes and/or limited the analysis to only some of the values an attribute can take. For example, Christmansson and Chillarege (1996); Duraes and Madeira (2006); Eldh et al. (2007) considered only coding faults; Lutz and Mikulski (2004) studied only safety-critical post-launch anomalies; Leszak et al. (2002) limited the analysis only to post-release change requests; Zheng et al. (2006) considered only ODC fault types and did not consider severity. Furthermore, severity of software failures has rarely been considered in published literature due to the lack of good quality data. Finally, although some of the related works analyzed combinations of attributes, the associations between these attributes were not quantified (Yu, 1998; Leszak et al., 2002; Lutz and Mikulski, 2004).

– We explored trends in number of failures within individual releases (i.e., pre-release to post-release) and across multiple releases (i.e., release n to $n+1$) for a subset of eleven components, which had a sufficient number of non-conformance SCRs and at least two releases. This part of our work can be considered as a conceptual replication carried out on a different type of system, within a different development context (i.e., mission critical software), which has been argued to increase the external validity (Andersson and Runeson, 2007; Kitchenham, 2008; Yin, 2014). Related work in this area (Biyani and Santhanam, 1998; Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Pighin and Marzona, 2003; Ostrand et al., 2005a,b; Bell et al., 2006; Andersson and Runeson, 2007; Grbac et al., 2013) looked at the relationships between the number of faults or fault density within individual releases (i.e., pre-release vs. post-release) and/or across multiple releases (typically from one release to the next). However, the results were not consistent, which necessitates further exploration.

This paper proceeds by presenting the definitions of the terms in section 2 and discussing the related work in section 3. Then, in section 4 we describe the details of the NASA mission case study. Section 5 addresses research questions RQ1 through RQ3 based on the cumulative data for all releases and for all 21 components, while section 6 is focused on questions RQ4 through RQ6 pertaining to analysis within individual releases and across multiple releases for a subset of eleven components for which there was sufficient data. Finally, we discuss the threats to validity of the study in section 7 and present the concluding remarks in section 8.

2 Definitions of the terms used in this paper

The analysis presented in this paper is focused on *types of software faults*, *detection activities*, and *failure severity*, which are attributes extracted from data fields in the non-conformance SCRs. The definitions of these attributes and their corresponding values are as follows.

– *Type of faults* refer to the root causes of software failures. For each non-conformance SCR, project analysts select the fault type value from a pre-defined list. With the help of the project personnel and based on standard definitions from ISO/IEC/IEEE 24765 (2010), we grouped these values into the following major fault types:
  – *Requirements faults* include incorrect requirements, modified requirements, and missing requirements.
  – *Design faults* result from human errors during the design of the system (e.g., omission of necessary component interaction).
  – *Coding faults* include mistakes in the source code, such as typos, logical errors, incorrect algorithms, missing code, etc.
  – *Data problems* result in a failure in response to some particular pattern of data. In this mission, data problems arise due to shared data and/or interfaces among components (or in some cases subcomponents).
  – *Integration faults* include faults associated with the integration of subcomponents, components and/or subsystems. (Hardware related integration faults were not included since the analysis in this paper is focused on software faults only.)
  – *Other faults* include types of faults that did not fit in any of the above categories. Examples include process or procedural issues, simulation errors, testing issues (e.g., incorrect test configuration), and other problems that were not directly related to the software artifacts being developed.

- *Activity* that was taking place when the fault was detected or the failure was exposed, for each non-conformance SCR, was entered as a textual description by the project analysts. With the help of the project personnel and based on definitions from the NASA Software Safety Guidebook (NASA-GB-8719.13, 2004) and ISO/IEC/IEEE 24765 (2010) we grouped these values into the following major detection activities:
  - *Inspections / audits* refer to formal planned visual inspections that follow specified steps and assign specific roles to individual reviewers by a team specifically trained in inspection techniques.
  - *Analysis* refers to activities throughout the life cycle, most of which were conducted manually. Some examples include informal reviews, desk checks, code walkthroughs, and static code analysis.
  - *Testing* includes various types of testing (e.g., dry run testing, integration testing, stage testing, acceptance testing, regressing testing).
  - *On-orbit* activity encounters the cases when failures occurred post-release while the software was on-orbit.
  - *Other* category includes all other pre-release detection activities not mentioned above.
- *Severity* is defined as the potential or actual impact of the failure on the system. In this paper we consider *safety-critical* and *non-critical failures*. Safety-critical failures represent failures which would result in loss of a safety-critical function or loss of a critical mission support capability (NASA-GB-8719.13, 2004).

Similarly as in case of the Orthogonal Defect Classification (ODC) (Chillarege et al., 1992), the values of each attribute are mutually exclusive. For example, a fault can be detected (or failure exposed) due to only one type of activity. More importantly, 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.

3 Related Work

The first group of papers related to our work were focused on characterizing some aspects of software faults and in some cases other software artifacts (Christmansson and Chillarege, 1996; Duraes and Madeira, 2006; Eldh et al., 2007; Leszak et al., 2002; Lutz and Mikulski, 2004; Yu, 1998; Zheng et al., 2006). These studies differ in terms of their main goals, systems studied, and attributes explored. None of them considered the evolution of software across multiple releases.

Some of the studies had a goal to discover representative fault types to be used for fault injection at the source code level (Christmansson and Chillarege, 1996; Duraes and Madeira, 2006). To do so, these studies used the Orthogonal Defect Classification (ODC) to classify the fault types; Christmansson and Chillarege (1996) used 408 defect reports from an IBM operating system and Duraes and Madeira (2006) used 668 faults from twelve open source projects. Despite the vast differences between the case studies, the distribution of the fault types was very similar. Eldh et al. (2007) also conducted a study of software faults aimed at exploring candidates for fault injection. It was based on 362 failures reported during testing and, similarly as (Christmansson and Chillarege, 1996; Duraes and Madeira, 2006), it was focused on coding faults. The results showed that the majority of software faults manifested themselves for the first time at the integration/system level. Other type of software faults (e.g., requirements faults, design faults, integration faults), detection activities, and failure severity were not explored in these works (Christmansson and Chillarege, 1996; Duraes and Madeira, 2006; Eldh et al., 2007).

Leszak et al. (2002) conducted a study of defect (i.e., fault and failure) modification requests (MRs) from a network element of an optical transmission network, which considered the 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 study showed that implementation faults (which likely included coding faults and integration faults) consumed 75% of the total effort. Yu (1998) analyzed the phase when faults were introduced into a switching telecommunication software and reported that nearly half of the faults were related to coding faults and the majority of them could have been prevented.

Lutz and Mikulski (2004) studied safety-critical (i.e. high severity) post-launch anomalies[1] 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, 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). Their explored the association between defect (i.e., fault) type and trigger, and also between target and trigger and although correlation between some attributes was suggested, it was not quantified.

The work by Zheng et al. (2006) was aimed at exploring the value of static code analysis as a fault detection technique. Their work used data extracted from three large-scale industrial software systems developed at Nortel Networks and found that automatic static code analysis had similar fault removal yield as manual inspections, while the fault removal yield of execution-based testing was two to three times higher than that of static code analysis. In addition, the authors mapped the faults identified by the static code analysis and inspections to the ODC fault types (i.e., assignment, checking, function, algorithm, interface, build/package/merge, timing/serialization, and documentation). Other fault types and severity of software failures were not considered (Zheng et al., 2006).

Another group of papers related to our study, although with different goals, explored fault and/or failure characteristics across at least two releases (Biyani and Santhanam, 1998; Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Pighin and Marzona, 2003; Ostrand et al., 2005a,b; Bell et al., 2006; Andersson and Runeson, 2007; Grbac et al., 2013). It should be noted that none of these works considered the types of faults and the activities taking place when the failures were revealed on a finer granularity than pre-release and post-release. Severity, to a limited extent, was considered only in (Ostrand and Weyuker, 2002).

Biyani and Santhanam (1998) explored the relationship between the number of faults in consecutive releases, as well as the relationship between pre-release and post-release fault proneness for four releases of a commercial application. 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.

The first systematically reported study of software faults and failures which empirically investigated several common software engineering beliefs was conducted by Fenton and Ohlsson (2000). The work was based on two releases of an Ericsson telecommunications application and the fact that most fault prone modules pre-release were among the least fault prone post-release was the most surprising result. Recognizing the importance of replicated studies Andersson and Runeson (2007) and Grbac et al. (2013) conducted two conceptual replications of the study by Fenton and Ohlsson (2000). While the majority of the results in the replicated studies were fairly consistent, the results related to the relationship between pre-release and post-release faults were not. That is, Andersson and Runeson (2007) and Grbac et al. (2013)[2] found that a high incidence of pre-release faults implied a high incidence of post-release faults.

Ostrand and Weyuker (2002) 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-release to post-release and from one release to the next for twelve releases of an

[1] Anomaly is defined as anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documents (ISO/IEC/IEEE 24765, 2010).

[2] The hypothesis in the case of the second replication study (Grbac et al., 2013) was tested conditionally using only faults detected during site testing (i.e., information on faults detected during product operation was not available).

inventory control system. Consistent with the findings by Fenton and Ohlsson (2000), 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 (Ostrand and Weyuker, 2002), but only to a limited extent, that is, evidence of the Pareto principle was found when analyzing the system by release and severity. The follow up work by Ostrand et al. (2005a) considered several additional releases of the same system (Ostrand and Weyuker, 2002) and used a negative binomial regression model to predict which files are likely to contain most faults in the next release. Later works by the members of the same research group tested the accuracy and applicability of the model by applying it to more releases of the original case study and two new systems (Ostrand et al., 2005b; Bell et al., 2006).

Another work focused on software evolution through releases, conducted by Pighin and Marzona (2003), used the average fault density at each release to track the fault proneness across 23 releases of a management application and 15 releases of a medical application. The results showed that files with above average fault density in the first release tend to have higher than average fault densities in later releases as well.

In our previous work (Hamill and Goseva-Popstojanova, 2009) we used the same NASA mission used in this paper and the GNU C-compiler GCC to explore the fault localization and major fault types in large scale software applications. The results showed that a large portion of failures required making fixes across multiple files. Additionally, it was identified that requirement faults, coding faults, and data problems were the dominating fault types in the NASA mission. The only attribute considered in our earlier work (Hamill and Goseva-Popstojanova, 2009) was the fault type. However, we did not consider the trends of fault types within individual releases (i.e., pre- and post-release) and across multiple releases. Our follow up work (Hamill and Goseva-Popstojanova, 2013) was focused on empirical characterization of software fixes. (A fix refers to all changes made to correct the fault(s) that caused an individual failure.) In that work the fixes which permanently addressed the root cause(s) of the failure were distinguished from workarounds, which were implemented to circumvent the problem without actually correcting the faults. 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. Neither (Hamill and Goseva-Popstojanova, 2009) nor (Hamill and Goseva-Popstojanova, 2013) considered the detection activities and failure severity, and their association with fault types.

It is obvious from the related work that additional detailed empirical studies are needed to reveal and quantify the associations among faults, failures and other relevant attributes, as well as to identify the characteristics invariant across multiple projects. The work presented in this paper is a step towards that goal.

4 Case study description

In this section we describe the case study, including its plan and the context. It should be noted that certain details are omitted to protect proprietary information.

We start with the plan of the case study, which includes identification of the case (i.e, what is studied), the case study design, the objective, research questions, methods for data collection, and selection strategy (Robson, 2002; Runeson and Höst, 2009).

The *case* in this study is the flight software of a large NASA mission. The design can be classified as *a single-case holistic design* because we treat the mission software as a single-unit of analysis (Yin, 2014). The single-case study is an appropriate design as it fits three single-case rationales discussed by Yin (2014), that is, our case is common for mission critical systems, revelatory, and longitudinal. First, typical for NASA missions this is a long-lived critical software system which undergoes evolutionary development and requires sustained engineering, and therefore can benefit from thorough empirical analysis. Second, our research team had access to the change and corrective action tracking database which contains information on software fault and failure attributes that was either non-existing or inaccessible to previous empirical studies. In that aspect, the case study is worth conducting because the descriptive information can be revelatory (Yin, 2014). Third, since part of our work is focused on studying the mission software across multiple releases this is a longitudinal case.

The *objectives* of our study are (1) to contribute towards better understanding of fault types, detection activities and severity of software failures, including their associations and trends within and across releases, and (2) based on empirical evidence, to assist the developers and verification and validation personnel in improving the product quality in a cost-efficient way. These objectives can be classified as exploratory (i.e., seeking new insights and generating ideas for new research), descriptive (i.e., portraying the current status of studied phenomena) and improving (i.e., aiming to improve certain aspects of the studied phenomena) (Runeson et al., 2012).

The exploratory and descriptive aspects of our research are broken down in the following research questions:

RQ1: Are certain activities more likely to detect certain fault types? Which types of faults lead to post-release failures?
RQ2: Which activities are more likely to reveal safety-critical failures?
RQ3: Are certain fault types more likely to cause safety-critical failures?
RQ4: Does the contribution of dominating fault types change as the software matures across releases?
RQ5: Is there an association between the number of failures reported in release n and release $n+1$ (i.e., across releases)?
RQ6: Is there an association between the number of failures reported pre-release and post-release (i.e., within individual release)?

Based on the empirical findings we complied detailed lessons learned which were regularly communicated to the mission's independent verification and validation team, and we identified several directions for cost effective improvement of the software quality.

The *data collection procedures* are a crucial part of any protocol for conducting case studies (Yin, 2014) because they affect the data quality and thus the outcomes of the case study research. Therefore, we provide details of the process followed by the NASA mission for creating and addressing software change requests. The change tracking system of the NASA mission stores Software Change Requests (SCRs) entered by analysts. SCRs are filed at the component level, per component release. Since we were interested in exploring and characterizing the associations between faults and failures we focused on SCRs entered when non-conformance to a requirement was observed. By definition non-conformance SCRs represent failures. It should be noted that some failures have been observed (for example during testing or operation), while others were only potential failures that have been prevented from happening by detecting faults through verification activities (e.g., analysis, inspection) and then fixing them pre-release. 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 the non-conformance observed, and 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 determined that the problem needs to be fixed, the SCR is assigned to an analyst who records data, including the type of fault that caused the failure (selected from a pre-defined list), and the severity of the failure. A review board verifies the data recorded in SCRs in terms of consistency and correctness. 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. Obviously, the procedures and processes followed by NASA provide assurance of the data validity and accuracy.

The *selection process* addressed the selection of components and the selection of non-conformance SCRs for these components. Based on whether data was available for all consecutive releases, we selected 21 components for the analysis, referred to as components 1 through 21. When it comes to the selection of specific SCRs attributed to these 21 components, to ensure the data quality we removed the SCRs that were missing multiple mandatory fields, SCRs that were withdrawn, 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 the 21 components over a period of almost 10 years.

Next, we focus on the context of our case study (Kitchenham et al., 2002; Petersen and Wohlin, 2009), describing the product, development process and organizations, and the practices and techniques.

The *product* used as a case in this study is an embedded software system, more specifically a flight software of an active NASA mission. Based on the selection criteria described above, our case consists of 21 components, that together contained millions of lines of code in over 8,000 files. Details for each component, including the number of releases, lines of source code (LOC), the number of files, the cumulative number of non-conformance SCRs over all releases, and the number of non-conformance SCRs per file are given in Table 1.

Table 1 Details of the 21 components, which are grouped by the number of releases each has undergone

Component	**# of releases**	**LOC**	**# of files**	**# of non-conf SCRs**	**# of non conf SCRs per file**
1	1	48,910	207	350	1.69
2	2	60,386	200	22	0.11
3	2	78,854	287	8	0.03
4	2	46,657	228	15	0.07
5	2	71,953	269	13	0.05
6	2	92,978	321	21	0.07
7	2	34,938	289	73	0.25
8	2	43,012	270	103	0.38
9	2	21,266	125	19	0.15
10	3	83,134	356	27	0.08
11	3	103,145	444	40	0.09
12	3	55,475	277	12	0.04
13	3	57,800	599	104	0.17
14	3	27,940	280	75	0.27
15	3	38,882	81	84	1.04
16	3	47,541	169	129	0.76
17	4	147,520	587	202	0.34
18	5	174,614	552	239	0.43
19	7	164,419	747	183	0.24
20	7	737,504	1368	735	0.54
21	7	74,618	415	104	0.25
Total	-	2,211,546	8,071	2,558	0.32

Some details of the *development process and organizations* are as follows. This mission is developed following an evolutionary (i.e., iterative) development process by teams at several locations. The number of locations varied from two to four throughout the project lifetime. It should be noted that each component is on its own release schedule. Therefore, the number of releases differed per component, ranging from one release to seven releases. (The components in Table 1 are grouped by the number of releases each has undergone.) Within each release, due to the mission critical nature of the software, the development organizations followed fairly traditional life cycle phases. The fault detection process spanned throughout the life cycle, that is, pre-release (i.e., development and verification & validation, including independent verification & validation) and post-release (i.e., on-orbit). It included different software artifacts (e.g., requirements, design, code, procedural issues).

With respect to *practices and techniques* we briefly summarize those relevant to the data quality. This NASA mission has a software review board which is responsible for determining which SCRs need to be addressed (e.g., there may not be any changes required for duplicate SCRs or SCRs due to operator error), for appointing an analyst(s) to individual SCRs, for ensuring that the proposed solution is appropriate and that changes indeed address the problem reported. The review board follows a well defined process and procedures with respect to tracking SCRs and updates the attribute values as deemed appropriate, which supports the consistency and quality of the data.

Several characteristics of our case study, such as *triangulation*, *analysis techniques*, and *iterative research and reporting process*, are also worth emphasizing.

We followed the principle of *data (source) triangulation*, that is, the data were collected from different sources, which included Software Change Requests (SCR), as well as corrective actions notifications. Whenever possible we used different data sources to confirm the data consistency and accuracy. For example, we extracted the values of the severity attribute from the non-conformance SCR documents in the change tracking system. In addition, we also explored the severity levels assigned to changes made to address each SCR in the corrective action notification documents. The consistency and the quality of the data were confirmed by the fact that there were very few differences between severity levels originally associated with the SCR and those associated with corrective actions. In a few cases when they differed, we used the values associated with the corrective action, which based on the recommendation by the project analysts are considered to be more accurate.

In addition, we combined different types of data collection methods, which is known as *methodological triangulation*. Thus, even though this is an observational study, the work reported in this paper was done in close collaboration with NASA personnel. This allowed us, in addition to extracting data from multiple archival documents described above, to use interviews (both informal, open ended and focused) as data collection method. The domain expertise provided by the project personnel helped us to understand the software in the context of the larger system, and clarify the meaning, usage, and relationships of data fields and values.

In this paper we use several *data analysis methods*. For the association between pairs of different attributes (i.e., *fault types*, *detection activities*, and *severity*) we used methods appropriate for cases when at least one of the variables has categorical values (i.e., is given at a nominal scale). Thus, for RQ1 - RQ3 we used contingency tables (also presented as two dimensional bar graphs), χ^2 test for contingency tables, and contingency coefficient to measure the associations (Seigel and N. J. Castellan, 1988). These analysis methods, to the best of our knowledge, have not been previously used in software engineering research. For the analysis of dominant fault types across releases, the association between the number of failures in subsequent releases, and the association between pre-release and post-release failures we used box plots (RQ4), and scatter plots and descriptive statistics techniques (RQ5 - RQ6). Because the number of components was small, which leads to low statistical power, we did not quantify the associations by using Pearson or Spearman correlation coefficients.

Last but not least, the case study research and reports to the sponsoring agency and those published in the research literature (including this paper) were completed through an *iterative process*. Thus during the course of our work, in collaboration with NASA personnel, we went through multiple cycles of analyzing, presenting, reviewing, and re-analyzing. The provided input and regular feedback contributed to the quality of the collected data and ensured accurate interpretation of the results.

5 Analysis of fault types, detection activities, and failure severity and their associations

It is fairly common in the related work to group the data for files, components, or even missions cumulatively for the purpose of conducting fault and/or failure analysis (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Lutz and Mikulski, 2004; Andersson and Runeson, 2007). Similarly, the analysis presented in this section is based on the SCRs for all components, cumulatively for all releases. Thus, the results presented in this section are based on 2,558 SCRs which were entered throughout the life cycle (e.g., development, testing, and operation) of the 21 components, over a period of almost 10 years.

In our previous work (Hamill and Goseva-Popstojanova, 2009)[3] we explored if certain fault types are more common causes of failures than other fault types. The results presented in Figure 1 showed that *coding faults*, *requirements faults*, and *data problems* are the three most common faults types. Specifically, *coding faults* and *requirements faults* were responsible for 36% and 35% of the total number of failures, followed by *data problems*[4], which were the source of approximately 15% of the total failures. Design faults accounted only for 6% of the total number of failures, which may in part be due to the fact that for this NASA mission the design documents typically did not capture detailed design.

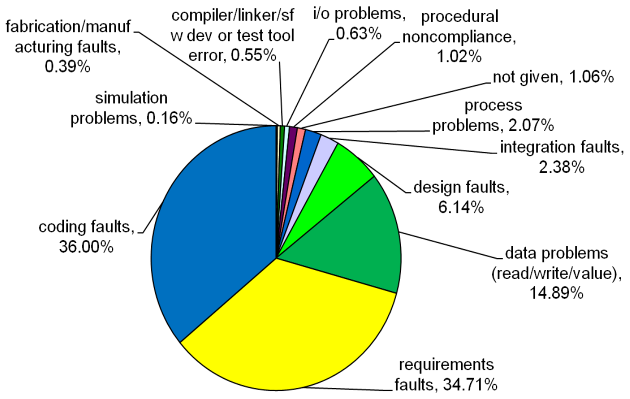

Fig. 1 Distribution of SCRs across different fault types

In this paper we explore the associations among fault type, detection activity, and severity. In particular, in this section we investigate the first three research questions:

RQ1: Are certain activities more likely to detect certain fault types? Which types of faults lead to post-release failures?
RQ2: Which detection activities are more likely to reveal safety-critical failures?
RQ3: Are certain fault types more likely to cause safety-critical failures?

To formalize the observed results, we used statistical tests including testing the statistical significance of the observed phenomena. For this purpose, each attribute (i.e., fault type, detection

[3] It should be noted that the dataset considered in our previous study (Hamill and Goseva-Popstojanova, 2009) was slightly larger than the one in this paper (i.e., 2,858 rather than 2,558 non-conformance SCRs) since further analysis of additional fields (i.e., detection activity and severity) led us to exclude some SCRs from the analysis. Nevertheless, the results and observed trends remain the same, with only insignificant changes in some values. For consistency, all results given in this paper are based on the new sample of 2,558 non-conformance SCRs.

[4] In most of this paper *integration faults* and *data problems* are grouped together in *integration & interface faults*.

activity, and severity) was considered to be a random variable X. Note that the fault type and activity attributes are based on a nominal scale (i.e., have categorical values), while the severity attribute is based on ordinal scale. To test if the results are statistically significant, for each pair of attributes X and Y (e.g., detection activity and fault type, detection activity and severity, and fault type and severity) with n and m categories respectively, we built an 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 approximated with a chi-square distribution with $(m-1)(n-1)$ degrees of freedom. When used for contingency comparisons, χ^2 test is a non-parametric test, since it compares entire distributions rather than parameters of distributions. Note that the power of the χ^2 test used for contingency tables depends not only on the total number of instances, but also on the frequencies in each category. Therefore, the power of the test may be different for different pair of attributes, depending on how rare are the considered categories. In all cases we achieved statistical power in the 70% - 90% range.

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 attributes is categorical (i.e., given on a nominal scale). (The more widely used Pearson and Spearman correlation coefficient cannot be used when the attribute values are on nominal scale.) Unlike the other measures of correlation, the maximum value C_{max} of C depends on the size of the contingency table. By normalizing C with the corresponding C_{max} we ensure that the range will be between 0 and 1, and hence, $C^* = C/C_{max}$ values for different size tables can be compared (Blaikie, 2003). (Further details on the χ^2 test for contingency comparison and contingency coefficient C are given in Appendix A.)

The statistical tests in this section were based on a stratified random sample to ensure that the relative contribution of each component to the overall sample of failures was preserved. (Because some components had much less non-conformance SCRs than others, if the sampling was not stratified the chance to select SCRs from these components would have been small.) In particular, we randomly selected three quarters of the failures from each component, which led to a random sample of $N = 1,918$ failures. This random sample is large enough to allow for statistical power of 70% or higher. For each attribute considered in this paper, we checked to ensure that the trends seen in the random sample accurately represent those seen in the population. It should be noted that the tables and figures presented throughout this section represent the entire sample of 2,558 failures, while the random sample was used for statistical tests.

5.1 RQ1: Are certain activities more likely to detect certain fault types? Which types of faults lead to post-release failures?

First, we study the distribution of failures across different detection activities. As shown in Figure 2, 49% of failures were discovered during *analysis*, 38% were discovered during various types of *testing*, 7% were discovered during *inspections or audits*, and only 3% were discovered *on-orbit.* The fact that only 3% of failures occurred post launch *on-orbit* shows that the verification and validation (V&V) procedures and processes in place were very effective at discovering and fixing faults during development and testing (i.e., pre-release). We also note that the percentage of non-conformance SCRs entered post-release is comparable or even smaller than in other studies based on industrial software applications (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Andersson and Runeson, 2007).

Similarly to other works in this area (Fenton and Ohlsson, 2000; Leszak et al., 2002; Lutz and Mikulski, 2004; Zheng et al., 2006; Andersson and Runeson, 2007; Grbac et al., 2013), the number of SCRs reported per detection activity can be determined from the available data, but the amount of effort spent conducting each activity was not available. Based on the input from the project personnel, it appears that *analysis* and *testing* were the most common verification activities performed on the mission. However, without quantitative values for the amount of effort spent on each activity, claims that one activity does better with respect to observed (potential

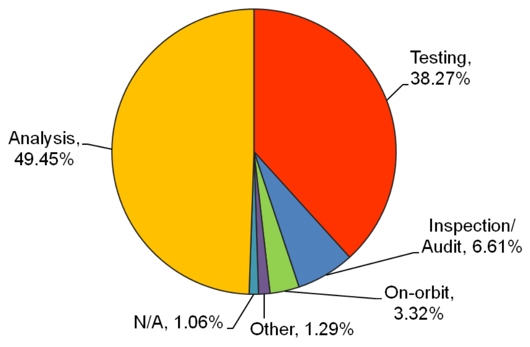

Fig. 2 Percentage of SCRs detected by different activities

Table 2 Frequency counts across detection activities and fault types

	Inspections Audits	Analysis	Testing	Other	On-orbit
Requirement Faults	79	519	267	11	12
Design Faults	15	59	64	8	11
Coding Faults	27	355	462	32	45
Integration & Interface Faults	34	262	126	6	14
Other	14	70	57	6	3
Total	169	1265	976	63	85

or actual) failures than other activity cannot be made. Nevertheless, we can still explore how effective different activities were in detecting different types of faults.

For the purpose of this analysis we used the following categories for activities: *inspection/audit*, *analysis*, *testing* (which includes all types of testing activities), *on-orbit* and *other* (which includes ‘n/a’, and the original ‘other’ category from Figure 2). With respect to fault type, we use the following categories: *requirements faults*, *design faults*, *coding faults*, *integration & interface faults* (which includes ‘integration faults’ and ‘data problems’ from Figure 1), and other (which includes the remaining categories in Figure 1).

A three dimensional plot of the frequency counts for the resulting 5 x 5 contingency table for activities and fault types is shown in Table 2 and Figure 3. Based on the contingency table, we made the following observations with respect to different activities:

– As expected *analysis* and *testing* activities revealed the majority of faults for each fault type. In general, a smaller percentage of total failures was detected by *testing* than by *analysis* (i.e., 38% of failures were detected by *testing* compared to 49% detected by *analysis*). *Testing* activities were more effective than *analysis* in revealing *coding faults* (i.e., 50% of *coding faults* were detected during *testing* while 39% were detected through *analysis*). On the other hand, *analysis* revealed more *requirements faults* than *testing* (i.e., 58% of *requirement faults* were detected through *analysis* while only 30% where detected during *testing*).
– *Inspection/Audit* activities revealed close to 7% of the total number of failures. 47% of these failures were caused by *requirement faults.*
– The majority of *on-orbit* failures (i.e., close to 53%) were caused by *coding faults.*

As it can be seen from Figure 3, the number of on-orbit failures is very small (i.e., 3% of the total number of failures), which when distributed across different fault types leads to small frequencies for all faults types except coding faults. Therefore, we further grouped the detection activities, using the following categories *analysis*, *testing*, and *other* (which includes *inspection*, *on-orbit*, and *other* categories from Figure 3). This way we can use the χ^2 test to compare the distribution of fault types across the two dominant detection activities – analysis and testing – and achieve statistical power of over 70%. Based on the 3 x 5 contingency table we calculated

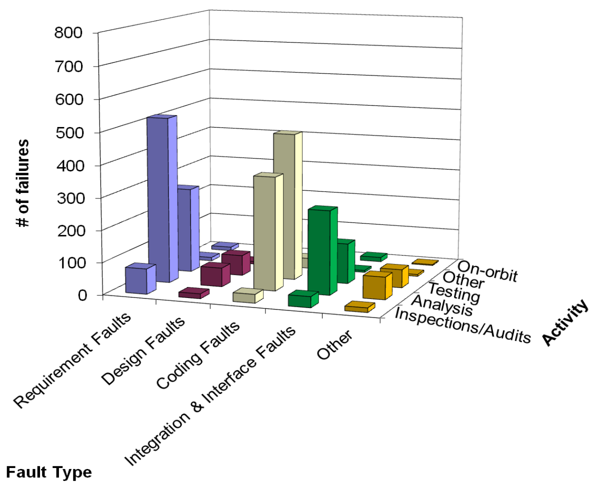

Fig. 3 Frequency counts across detection activities and fault types

$\chi^2 = 91.78$, which has a probability of less than 0.001 under the null hypothesis. This results confirms with statistical significance of $\alpha = 0.05$ that **fault types have different distributions for analysis and testing activities**.

We then calculated the contingency correlation coefficient $C = 0.21$, $C_{max} = 0.85$, and $C^* = 0.25$, which can be considered as a medium degree of correlation (Seigel and N. J. Castellan, 1988). Based on the fact that the probability of obtaining $\chi^2 = 91.78$ is less than α, it follows that the correlation between the activity and fault type attributes is statistically significant. This result, in other words, suggests that **certain activities are more likely to detect failures caused by certain type of faults**, that is, each of the detection activities is necessary as it is responsible for revealing some of the failures caused by each fault type.

Since on-orbit failures are of the highest priority we explored the characteristics of on-orbit failures compared to all other failures by focusing on another meaningful grouping of detection activity categories, that is, one 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). This allows us to explore a different question: "Which types of faults were most common causes of post-release failures?" In other words, we were interested to discover which fault types are most likely to escape the pre-release detection, which obviously has a practical value.

Table 3 Distribution of major fault types for pre-release and post-release SCRs

Fault Type	**% of Development & Testing SCRs**	**% of On-orbit SCRs**
Requirements faults	35.42	14.12
Design faults	5.90	12.94
Coding faults	35.42	52.94
Integration & interface faults		
Data problems	15.20	5.88
Integration faults	2.10	10.59
Other	5.94	3.53

Table 3 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 the *Integration & interface* type of faults consists of *integration faults* and *data problems*, whose values are given in

Table 3 in order to distinguish their contributions.) Based on these results we made the following main observations:

- **Coding faults and requirement faults are major causes of failures observed during development and testing, as well as on-orbit.** However, 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 contributions of failures due to design faults and integration faults are 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* increased *on-orbit* suggests that components may be interacting in unexpected ways and that analysis of component interactions and conducting more integration testing before components are released to fly on on-orbit may be beneficial.

Due to the fact that post-release failures were rare, especially when distributed across five fault type categories, we grouped the fault types into three categories: *requirements and design*, *coding and integration & interface*, and *other*, which allowed us to test the hypothesis related to the distribution of fault types for pre-release and post-release (i.e., on-orbit) failures (see the discussion provided in Appendix A). We calculated $\chi^2 = 6.79$, and since the probability of obtaining $\chi^2 = 6.79$ is less than $\alpha = 0.05$, it follows that **the fault types are not distributed the same way across failures reported pre-release and post-release,** and the results are statistically significant.

5.2 RQ2: Which detection activities are likely to reveal safety-critical failures?

For the purpose of our analysis, we grouped the severity levels used by the mission into three categories: *safety-critical*, *non-critical*, and *unclassified.* Safety-critical failures represent failures which would result in loss of a safety-critical function or loss of a critical mission support capability.

We found that 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 was assigned). Although safety-critical failures were less than 9% of the total number of failures, fixing them has a high priority as they can lead to unacceptable consequences. Thus, characterizing such failures is important.

By considering the detection activity and severity level for each SCR we explored if certain activities were more likely to detect safety-critical failures to help answer research question RQ2. From Figure 4 it is clear that *analysis* detected more *non-critical* and *unclassified* failures (i.e., 51% of each class) than any other activity. Further, it can be seen that slightly more *safety-critical* failures were discovered by *testing* than by *analysis.* Specifically, 40% of the total number of *safety-critical* failures were discovered through some *testing* activity, while 35% were discovered during *analysis. Inspection/Audit* activities revealed approximately 7% of failures in each severity class. Another interesting observation is that a fairly significant portion (i.e., 13%) of the total number of *safety-critical* failures occurred *on-orbit.*

The frequencies of safety-critical failures for all cells of the contingency table were sufficient to allow for power of over 90% for the χ^2 test, which does not require further groupings of the

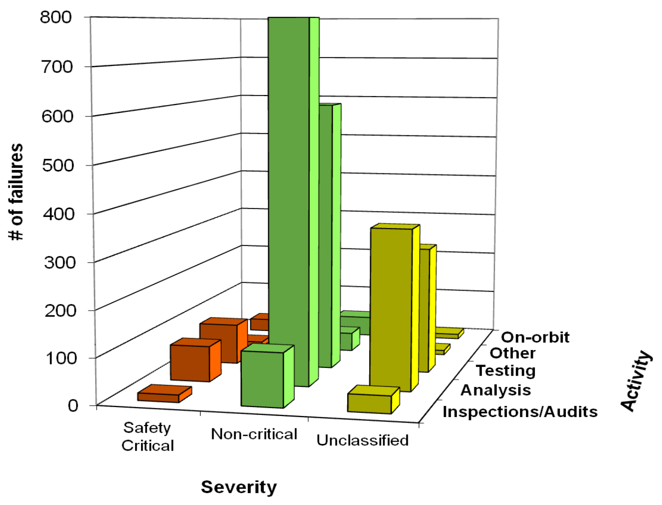

Fig. 4 Frequency counts across detection activities and severity levels

detection activities and severity levels. Since the probability of obtaining the value $\chi^2 = 81.72$ is less than $\alpha = 0.05$, we concluded (with statistical significance of 0.05) that **safety-critical failures were more likely to be revealed by certain activities.** The values of the contingency coefficient is $C = 0.20$ and $C_{max} = 0.85$, which leads to $C^* = 0.24$.

Of course, safety-critical on-orbit failures are the most critical class of failures. Hence, we also explored the distribution of severity levels (i.e., *safety-critical*, *non-critical* and *unclassified*) for failures reported during *pre-release* (i.e., development and testing) and *post-release* (i.e., on-orbit). The results are shown in Table 4. The value $\chi^2 = 61.29$ allowed us to conclude that **the distribution of the severity levels of pre-release failures differed from the distribution of the severity levels of post-release failures.** Further, $C = 0.18$, $C_{max} = 0.76$, and $C^* = 0.23$, which means that the correlation is moderate and statistically significant.

Table 4 Severity levels of pre-release and post-release failures

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

Based on Table 4 and the statistical results we made the following observations:

– **The percentage of safety-critical failures increased on-orbit.** Specifically, 34% of the total number of *on-orbit* failures were *safety-critical*, while less than 8% of the total number of failures reported during *development and testing* were *safety-critical.*
– **Greater emphasis was placed on fixing and documenting on-orbit failures**, which is obvious by the fact that only 13% of *on-orbit* failures were unclassified compared to 28% unclassified failures during *development and testing.*

These are not surprising results because on-orbit failures typically have more significant impact on the mission operation. In addition, it is less likely that the severity of on-orbit failures will remain unclassified, which may also have contributed to the increased percentage of on-orbit safety-critical failures.

5.3 RQ3: Are certain fault types more likely to cause safety-critical failures?

The next logical question is whether some fault types are more likely than others to lead to failures that have a critical impact on the system. The frequency counts of the main fault types in each severity level are shown in Figure 5. It can be seen that for each fault type (excluding *other*) 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*, followed by 24% caused by *requirement faults*, 15% by *integration & interface faults*, and 12% by *design faults*.** The distribution of fault types for safety-critical failures is also shown in Figure 6. It appears that for safety-critical failures the contributions of the coding faults and design faults were larger, while the contribution of the requirements faults was smaller compared to the distribution of fault types across all severity levels shown in Figure 1. Note that *integration & interface faults* in Figure 5 include *data problems* and *integration faults*. Even though the total percentage of *integration & interface faults* did not differ significantly for safety-critical and all failures (i.e., 14.67% vs. 17.27%) the contribution of *data problems* was smaller while the contribution of *integration faults* was larger for safety-critical failures than for all failures.

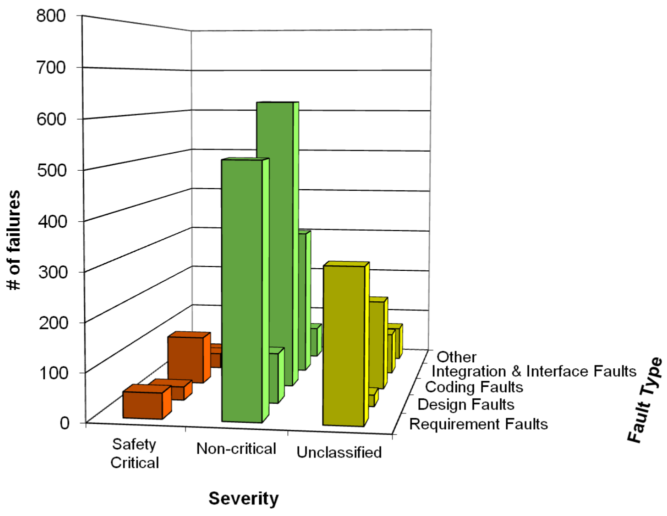

Fig. 5 Frequency counts across fault types and severity levels

As in the previous case, the frequencies of each cell in the contingency table were high enough to ensure more than 90% power of the χ^2 test without need of additional groupings. In this case, $\chi^2 = 88.07$, $C = 0.21$, $C_{max} = 0.85$ and $C^* = 0.25$. It follows that **safety-critical failures were related more often with some fault types than with others**, with 0.05 statistical significance.

When integrating these results with earlier results, it appears that **coding faults were not only responsible for a significant percentage of the total number of failures (i.e., 36%, see Figure 1) and more than 50% of the total number of on-orbit failures (see Table 3), they were responsible for over 45% of the total safety-critical failures (see Figure 5 and Figure 6) 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

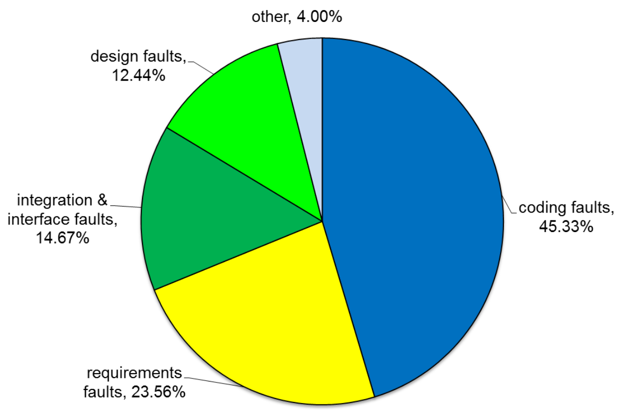

Fig. 6 Distribution of fault types for safety-critical failures

priority failures (i.e., safety-critical failures and on-orbit failures). For example, 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.

6 Analysis of trends across and within releases

Since the NASA mission follows an evolutionary development process with functionality being added between releases, we analyzed fault and failure attributes across releases (i.e., from one release to the next), as well as within individual releases (i.e., from pre-release to post-release). Our goal was to explore the existence of trends that could be useful in planning and managing quality assurance efforts throughout ongoing development and sustained engineering.

For the analysis presented in this section we selected a subset of components, each with two or more releases and at least 70 non-conformance SCRs filed against it. We believe that less failures, when broken down per release, may not show realistic trends, or at least the results will be very sensitive to small variations. Thus, the analysis presented in this section is based on the data from the following eleven components: 7, 8, 13, 14, 15, 16, 17, 18, 19, 20, and 21. This subset actually contains almost 70% of the total non-conformance SCRs (i.e., 2,029 SCRs were reported against the selected eleven components) and as expected it showed strong support for the observations made thus far. Specifically, in this section we explore the following research questions:

RQ4: Does the contribution of dominating fault types change as the software matures across releases?
RQ5: Is there an association between the number of failures reported across releases (i.e., between release n and release $n + 1$)?
RQ6: Is there an association between the number of failures reported pre-release and post-release (i.e., within release)?

The part of our study related to research questions RQ5 and RQ6 can be considered as conceptual replication, which therefore provides new evidence for previously explored phenomena and contributes towards identification of invariant characteristics across multiple projects.

6.1 RQ4: Does the contribution of dominating fault types change as the software matures across releases?

For the analysis in this section we study the failures due to the three major fault types: *requirements faults*, *coding faults*, and *integration & interface faults*, which consist of *integration faults* and *data problems*. This part extends the analysis presented in our previous work (Hamill and Goseva-Popstojanova, 2009) by considering how the contribution of the three dominating fault types changes as the software matures across releases.

Cumulatively for all components, requirements faults, coding faults, and integration & interface faults contributed from 82% to 87% of failures across different releases, which leads to the observation that the three major fault types persist across releases.

Considering each component individually we found that the vast majority of releases of each component have at least 75% of their total failures linked to requirements faults, coding faults, and integration & interface faults. We further explored the trends of the three major fault types across releases using the box plots shown in Figures 7 through 9. We use box plots because they show both the central tendency and dispersion of random variables (in our case the number of failures due to specific fault type for components at a given release). In each graph, for each release, the box contains the 25th to 75th 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. In the majority of cases in Figures 7 - 9, the maximum value shown by the whiskers belonged to component 20, which is not surprising having in mind that component 20 is the largest and among the oldest components in our sample.

The distribution of requirements faults across releases at the component level showed the largest variation (see Figure 7), which means that this type of fault varies more than coding faults and interface & integration faults for different components. On the other side, the distribution of coding faults was the most stable, and even though some components experienced higher number of coding faults, there was a clear central tendency with a median from 9.5 to 16 failures per component due to coding faults (excluding R7 which had median equal to 0). The number of components per each release was small (i.e., eleven components for R1 and R2, nine components for R3, five components for R4, and four components for R5) to allow for reasonable power[5] of the statistical test for central tendency, which therefore is not reported here.

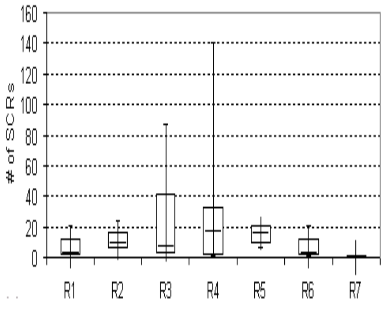

Fig. 7 Number of SCRs due to requirements faults per release

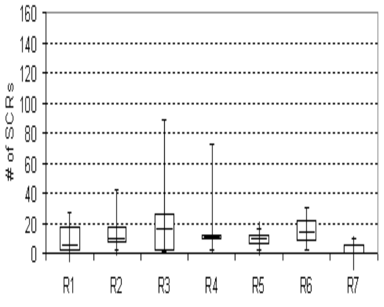

Fig. 8 Number of SCRs due to coding faults per release

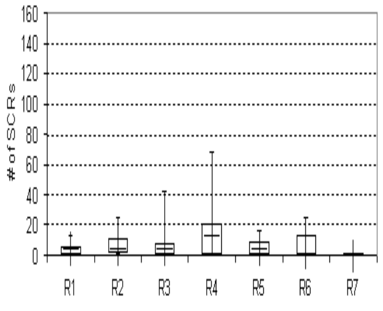

Fig. 9 Number of SCRs due to integration & interface faults (which include integration faults and data problems) per release

[5] The power of a statistical test is defined as the probability to reject the null hypothesis assuming it is false, that is, the probability to be able to identify a pattern in the data, assuming it exists.

6.2 RQ5: Is there an association between the number of failures reported across releases (i.e., between release n and release $n+1$)?

Next we explore if, at component level, the number of failures in release n provides any indication of the number of failures that occurred in the next release $n+1$, as such information could be used to focus resources on certain components in upcoming releases. Since later releases were based on fewer components and had significantly less failures reported, we explored RQ4 for releases R1 and R2, and then for R2 and R3.

Figure 10 shows a scatter plot representing the association 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. Figure 11 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.

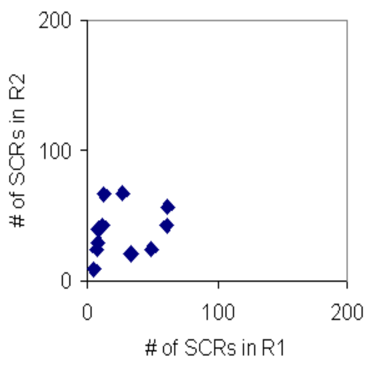

Fig. 10 Scatter plot showing association between the number of SCRs in R1 and R2, for the eleven components which had at least two releases

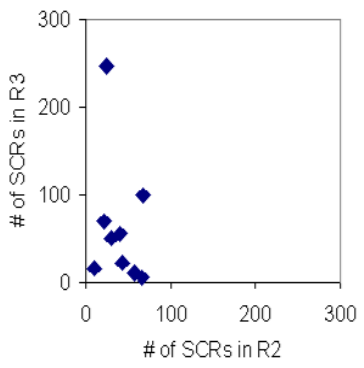

Fig. 11 Scatter plot showing association between the number of SCRs in R2 and R3, for the nine components which had at least three releases

Although the quantification of the correlation and the test of the statistical significance are important aspects of quantitative analysis, due to the small numbers of components (i.e., eleven or nine components) the power of the statistical tests would be very low[6]. Therefore, we do not present the results of the inferential statistics. Rather, based on the scatter plots shown in Figures 10 and 11 we conclude that **at component level it is unlikely that there is either positive or negative linear correlation between the number of failures in two subsequent releases**, that is, it appears that the number of failures in release n and the number of failures in release $n+1$ are unrelated.

Several related papers which studied fault persistence across software releases (Biyani and Santhanam, 1998; Ostrand and Weyuker, 2002; Pighin and Marzona, 2003) reported that software units[7] with high number of faults in release n were among the software units with high number of faults in release $n+1$. These results are not consistent with our observation, which most likely is due to the fact that these projects followed different evolutionary development practices. We believe that in our case study the lack of correlation between the number of failures identified in subsequent releases at the component level is due to the evolutionary development process which led to including new functionality between releases. Even more, components in this case study

[6] If a large effect size (i.e., correlation coefficient $|\rho| = 0.5$ (Cohen, 1988)) is assumed the two-sided test of statistical significance of the Spearman correlation coefficient with statistical power 80% would require 32 components, which is larger than our sample sizes.

[7] Software unit is at different level of granularity – file (Ostrand and Weyuker, 2002; Pighin and Marzona, 2003) or modules (Biyani and Santhanam, 1998).

had unique schedules with respect to the added functionality across releases. Thus, a plausible scenario which may explain the lack of correlation would assume that components with more added functionality would tend to experience more failures in the subsequent release, while components with less or no added functionality may experience less failures due to the fact that some latent failures have been fixed and few changes were introduced. 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. In that aspect, our study is not unique. The studies in the related work (Biyani and Santhanam, 1998; Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Pighin and Marzona, 2003) which explored the relationship between faults (or fault densities) in consecutive releases also did not consider the addition and/or removal of functionality since such data were not available.

We note that the results of different studies should be compared with caution because related papers (Biyani and Santhanam, 1998; Ostrand and Weyuker, 2002; Pighin and Marzona, 2003) considered fault-proneness, while we are studying failure-proneness. (A rigorous analysis should not equate faults and failures because it has been shown that there is no simple one-to-one relationship between faults and failures (Eldh et al., 2007; Hamill and Goseva-Popstojanova, 2009).) Another reason for careful comparison is the different levels of software units' granularity (i.e., component or module versus file level) and the sizes of the corresponding units. For our study the data were not available at file level for the whole mission. The analysis based on the subset for which data were available at the file level (i.e., 404 files) suggested that the observation that failures in subsequent releases are not linearly correlated would likely hold at the file level as well.

6.3 RQ6: Is there an association between the number of failures reported pre-release and post-release (i.e., within individual release)?

In this section we explore the potential association between the number of failures reported pre-release and post-release (i.e., within releases), first cumulatively for all releases and then within individual releases. Two intuitive, but opposite arguments can be provided for the association between pre-release and post-release failures (Fenton and Ohlsson, 2000). Thus, 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 (i.e., post-release). 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.

The scatter plot of the pre-release failures versus post-release failures, cumulatively over all releases for the eleven components considered in this section is shown in Figure 12. The association between pre-release and post-release failures could be quantified using the Pearson correlation coefficient or its non-parametric counterpart the Spearman correlation coefficient. However, as discussed in section 6.2 the small number of components per release, which leads to small power of the test, prevented us from quantifying the association. The scatter plot shown in Figure 12 suggests that **at component level, cumulatively over all releases, the number of post-release failures tend to be positively correlated with the number of pre-release failures.** From a software engineering perspective this means that cumulatively over all releases, components which have more non-conformance SCRs during *development and testing* also tend to have more *on-orbit* failures.[8]

Next, 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

[8] It should be noted that although component 20, which is the largest in size, has the highest number of both pre-release and post-release faults, in general component size in LOC did not appear to be linearly correlated neither with pre-release nor post-release faults, that is, in some cases smaller components had more pre-release failures and/or more post-release failures than larger components. In other words, component size is not the reason behind positive correlation of pre-release failures and post-release failures.

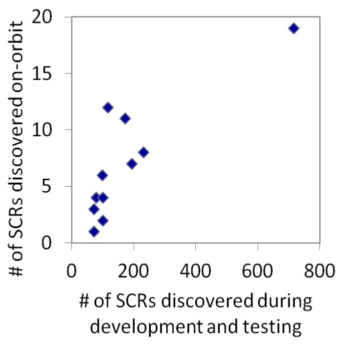

Fig. 12 Scatter plot showing association between pre-release and post-release failures, cumulatively over all releases, for the subset of eleven components

failures were reported in our data set for releases five through seven (R5-R7). The corresponding scatter plots are shown in Figures 13 – 16, where each dot represents a component[9]. These scatter plots also indicate moderate to strong positive correlation, which means that at component level within individual releases, a high incidence of pre-release failures implied a high incidence of post-release failures for most components. In other words, for releases one through four, the components that had a high number of non-conformance SCRs during development and testing tended to experience more on-orbit failures. Due to the low power we do not report the results of the inferential statistical tests here. Instead, we conducted similar descriptive statistical analysis about the persistence of failure proneness as Fenton and Ohlsson (2000); Ostrand and Weyuker (2002); Andersson and Runeson (2007); Grbac et al. (2013). The results are compared in Table 5.

Table 5 Pre-release / post-release persistence of faults incidence (for (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Andersson and Runeson, 2007; Grbac et al., 2013)), that is, of failure incidence (for this study)

Study	Release or Project	% of pre-release faults (failures) in modules with no post-release faults (failures)
(Fenton and Ohlsson, 2000)	Release n	93%
	Release $n+1$	77%
(Ostrand and Weyuker, 2002)	Releases 1-13	72% - 94%
(Andersson and Runeson, 2007)	Project 1	36%
	Project 2	29%
	Project 3	13%
(Grbac et al., 2013)	Release n	26%
	Release $n+1$	10%
	Release $n+2$	3%
	Release $n+3$	25%
	Release $n+4$	2%
This study	Release 1	32%
	Release 2	8%
	Release 3	23%
	Release 4	2%

Thus, Fenton and Ohlsson (2000) reported that almost all of the faults detected in pre-release testing appeared in components that subsequently had no post-release faults. In particular, they observed that 93 percent and 77 percent of the faults in pre-release testing occurred in components that had no post-release faults for the two releases, respectively. (That is, 100% of the post-release

[9] It should be noted that there is no missing data in Figures 13 – 16. Rather, some components have gone through less releases than others. For example, component 8 had only two releases, while component 20 had seven releases.

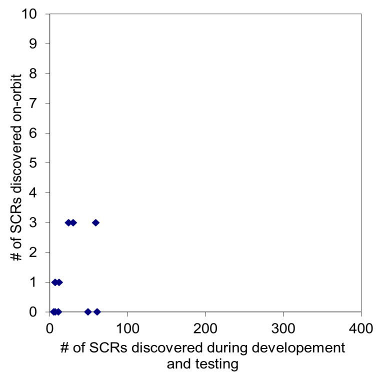

Fig. 13 Scatter plot showing association between the number of SCRs entered during development & testing and the number of SCRs entered on-orbit for release R1, nine components. Components 8 and 21 are not included because they were never loaded on orbit for release R1.

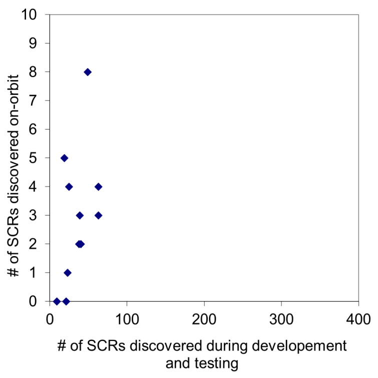

Fig. 14 Scatter plot showing association between the number of SCRs entered during development & testing and the number of SCRs entered on-orbit for release R2, eleven components.

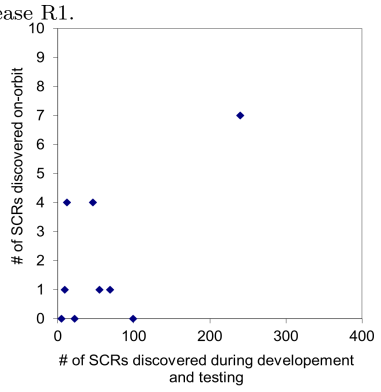

Fig. 15 Scatter plot showing association between the number of SCRs entered during development & testing and the number SCRs entered on-orbit for release R3, nine components.

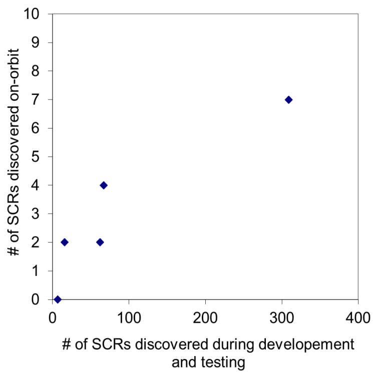

Fig. 16 Scatter plot showing association between the number of SCRs entered during development & testing and the number of SCRs entered on-orbit for release R4, five components.

faults occurred in components that had either 7% in one release or 23% in the other release of the pre-release faults.) These results were inconsistent with the common belief about the persistence of faultiness in components. Ostrand and Weyuker (2002) did similar evaluation of the late-pre-release and post-release faults for a software system with 13 releases and similarly as Fenton and Ohlsson (2000) found that from 72% to 94% of pre-release faults were detected in files with no post-release faults. Basically, the results of these two studies suggested that components (or files in (Ostrand and Weyuker, 2002)) that contained pre-release faults were not the most-likely place where post-release faults were detected.

On a contrary note, Andersson and Runeson (2007) in a replication study of (Fenton and Ohlsson, 2000) reported that only 36 percent, 29 percent, and 13 percent of the faults in pre-release testing occurred in modules that have no subsequent post-release faults for the three projects, respectively. The percentages of faults detected in pre-release testing that occurred in modules with no post-release faults were even lower in the five releases considered in the second replicated study (Grbac et al., 2013). These finding were also supported by two studies based on the open source application Eclipse. Thus, Zimmermann et al. (2007) obtained significant and high correlation coefficients (0.907 for files and 0.921 for packages) meaning that the files/packages that had high number of pre-release faults also had high number of post-release faults. Similar results were obtained in a study performed by Shihab et al. (2010).

Having in mind the fact that the related work (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Andersson and Runeson, 2007; Grbac et al., 2013) analyzed pre-release and post-release faults, while we analyzed pre-release and post-release failures, we are again cautious to make straightforward comparison, but it appears that our results are consistent with the observations made by Andersson and Runeson (2007); Grbac et al. (2013); Zimmermann et al. (2007); Shihab et al. (2010). Thus, our results showed that for releases R1 through R4 components that did not have any post-release failures were not among most failure prone components pre-release, that is, were associated with only 32%, 8%, 23%, and 2% of the pre-release failures, respectively. With respect to the results presented in Table 5 we point out the following:

– **The persistency of fault proneness (or failure proneness in our study) from pre-release to post-release seems to be project specific.** Thus, for some projects components with high incidence of faults (or failures) pre-release continue to experience high incidence of faults (or failures) post-release (Andersson and Runeson, 2007; Grbac et al., 2013; Zimmermann et al., 2007; Shihab et al., 2010) and this paper. For other projects (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002) the opposite was true.
– **With respect to RQ6 these works can be considered as conceptual replication.** Several works considered different products from the telecommunication domain (i.e., switching system (Fenton and Ohlsson, 2000), inventory tracking system (Ostrand and Weyuker, 2002), two consumer products and one internal platform (Andersson and Runeson, 2007), and software system for telecommunication applications (Grbac et al., 2013)); two papers considered the open source product Eclipse (Zimmermann et al., 2007; Shihab et al., 2010); and our study is based on a mission safety-critical system. Interestingly, software units with high incidence of pre-release faults (or failures) are among more fault (or failure) prone post-release (that is, there is a persistence of fault (or failure) proneness) in products from different domains (i.e., telecommunication, open source and safety-critical space software). This fact appears to indicate that the domain is not the major factor behind this phenomenon.
– **Some factors that may influence the persistence of fault (or failure) incidence from pre-release to post-release were not measured in any of these case studies** (Biyani and Santhanam, 1998; Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Andersson and Runeson, 2007; Grbac et al., 2013). For example, pre-release testing efforts or the intensity of field usage, if measured, may help explaining some of the reasons behind the contradictory results.

7 Threats to validity

We discuss the threats to validity grouped in four categories: construct validity, internal validity, conclusion validity, and external validity.

Construct validity is concerned with whether we are testing what we intended to test. One obvious threat to construct validity is related to not having sufficiently defined constructs before they are translated to metrics. The use of inconsistent and/or not sufficiently precise terminology in the area of software quality assurance is a serious threat to validity and often makes meaningful comparisons of results difficult. Therefore, we provided the definitions of the terms, such as faults and failures, which were adapted from ISO/IEC/IEEE 24765 (2010). We avoided terms, such as defects, that lack rigorous definitions and were used inconsistently to refer to different terms.

The data values in each field and the interpretation of each attribute value can certainly influence the results. For the NASA mission, possibly because the change tracking system was not designed with such detailed analysis in mind, some formal definitions of field values were missing. Therefore, we took the time to ensure that we thoroughly understood the data and we were not misinterpreting any values.

While the categories of the fault types (i.e., ‘source’ field) were selected from a predefined list, the categories of detection activity (i.e., ‘discovered by’ field) were based on a free text field. The categories used for detection activity in this paper (see Figure 2) were developed with help from the project team members. The ‘analysis’ category of the activity attribute included only those SCRs that were specifically tagged ‘analysis’ in the ‘discovered by’ field. We manually explored multiple SCRs that were discovered by analysis and discussed the perceived meaning with multiple software engineers to ensure that we were not misclassifying this type of activity. The other categories for the activity attribute were either self explanatory or easily explained and grouped into categories as suggested by the project personnel.

So called mono-operation bias to construct validity is related to under-representation of the cause-construct. Many empirical studies experience lack of some types of data that, if available, may improve the interpretation of the results or help explaining the cause-effect relationships. This is especially true for case studies, which are observational by nature. In our case study, as in the related work, the data related to the effort spent on each detection activity and data related to the details of the added functionality were missing. In these cases we made clear statements of the limitations due to lack of specific data, which should allow future studies to gain access or collect data that would allow exploring some of the interpretations made in this paper and thus further advance the area.

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. In case of the NASA mission, the software review board is responsible for determining which SCRs need to be addressed, for appointing an analyst(s) to individual SCRs, for ensuring that the proposed solution is appropriate and that changes indeed address the problem reported. The review board follows a well defined process and procedures with respect to tracking SCRs and updates the attribute values as deemed appropriate. Clearly, the fact that the data used in this paper were reviewed by members of the review board supports data validity.

Validity of severity data is often questionable based on the subjectivity of the persons assigning the severity level and their interest in the problem being addresses, as it is common practice for development organizations to prioritize fixing faults based on the severity. Therefore, we handled the severity attribute with special care. In addition to the values of the severity attribute in the non-conformance SCR document, which were reviewed by the software review board, we also explored the severity levels assigned to changes made to address each SCR. That way we were able to verify the previously classified severity levels. It should be noted that we observed very few differences between severity levels originally associated with the SCR and those associated with changes. In cases when they differed, based on the recommendation made by the project analysts, we used the values associated with the change, which are considered to be more accurate. In addition, looking at severity levels associated with changes made to address each SCR, we were able to classify the severity levels of 78 previously unclassified SCRs.

To ensure the quality of the other data fields, based on an initial review of the available data in the SCRs, we removed the SCRs which were missing multiple mandatory fields, SCRs which were withdrawn, SCRs tagged as duplicates, and SCRs tagged as operator errors.

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. We took specific care to ensure that the appropriate statistical techniques were selected to test the hypotheses. Thus, we chose the statistical tests based on the measurement scales of the attributes and the validity of the underling assumptions of the tests.

Additionally, in this paper all formal statistical tests are based on random samples. As noted earlier, we checked that trends observed in the entire population hold true for the selected random subsets which we used for conducting statistical tests.

Typical for high-quality operational software, the sample of post-release SCRs was small, in spite the fact that our dataset spans over ten years period. Specifically, only 3% of the non-conformance SCRs were reported on-orbit. Therefore, comparisons of pre-release and post-release non-conformance SCRs were made on vastly different sized samples. This is a threat to validity to almost all studies that focus on analysis of post-release failures. For clarity, throughout the paper the percentages within each class, as well as within the entire sample are noted to avoid misrepresenting the data.

When it comes to the analysis across multiple releases (i.e., section 6) the number of components was small (i.e., at most eleven components), which inevitably leads to low power of the statistical tests. Therefore, we did not quantify the association (using either Pearson or Spearman correlation coefficient) nor carried out a formal hypothesis testing of the null hypotheses that the correlation between failures in subsequent releases, as well as the correlation between pre-release and post-release failures are zero. We made observations based on descriptive statistics and scatter plots, and compared our results to the related work.

Last but not least, the work reported in this paper was done in close collaboration with NASA personnel. In many instances, multiple cycles of analyzing, presenting, reviewing, and re-analyzing occurred. The provided input and regular feedback contributed to the quality of the research results and ensured accurate interpretation of the results.

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 of lines of code in over 8,000 files), developed at several locations, by different teams, with almost ten years of development and operation. Although these facts allow for some degree of external validation, we cannot claim that the results would be valid across all software products, from different domains.

It is widely accepted that generalizations are usually based on multiple empirical studies that have replicated the same phenomenon under different conditions (Andersson and Runeson, 2007; Kitchenham, 2008; Yin, 2014). The external validity of our study is, to some extent, supported by the fact that whenever possible we compared our results with the related work seeking to identify phenomena that are invariant across multiple domains. In that process, we were very careful to ensure that we do not compare 'apples to oranges' (e.g., faults to failures), which is a challenging task having in mind the inconsistent terminology used in this research area. In addition to providing definitions of the used terminology, we chose carefully the groupings of attribute values (e.g., fault types and activities) used in the statistical tests, not only to support the internal validity and the construct validity, but also to allow future related studies to compare their findings to our results in a meaningful and practically useful way.

8 Conclusions

In this paper we presented a case study based on a large NASA mission, specifically focusing on the characteristics of 'high priority' failure classes such as failures caused by dominating fault types, on-orbit failures, and safety-critical failures. For this purpose we analyzed three attributes – fault type, detection activity, and severity – and the extent of their associations. Furthermore, we studied the trends of fault and failure behavior within individual releases (i.e., pre-release and post-release) and as the software evolved across multiple releases.

Our work shows that in spite of the fact that change tracking systems were not developed for the purpose of analyzing and quantifying fault and failure attributes, they are a valuable source of such information. What distinguishes our study from most of the related work are the facts that we were able to tie the faults to the failures that they caused and to identify the detection activity that took place when faults were detected or failures surfaced. Therefore, we were able to study the associations among fault types, detection activities, and failure severity. Given the extensiveness and quality of the collected data, we believe that lessons learned in this case study are of value to a wider software engineering community because performing and publishing empirical studies and subsequently replicating them provides insights that have practical value and contributes towards identifying trends that are invariant across multiple projects and products.

Table 6 Summary of the main results

	Findings	Related studies	Section
RQ1	The distributions of fault types that caused failures were different across different detection activities. Analysis and testing revealed the majority of faults for each fault type. Testing was more effective in revealing coding faults, while analysis was more effective in revealing requirements faults.	Explored for other detection activities and fault types (Zheng et al., 2006). Comparison is not possible.	5.1
	The distribution of fault types differed for failures reported pre-release and failures reported post-release. While pre-release the percentages of SCRs due to requirements faults and coding faults were similar (i.e., around 35%), coding faults were a dominating root cause for post-release failures (close to 53%).	Not explored.	5.1
RQ2	The distribution of safety-critical and non-critical failures differed across activities. Slightly more safety critical failures were discovered by testing than by analysis (i.e., 40% vs. 35%).	Not explored	5.2
	The distribution of safety-critical failures differed pre-release and post-release. As expected, there were significantly more safety critical failures post-release than pre-release (i.e., 34% vs. 8%).	Not explored	5.2
RQ3	The distribution of severity of failures differed with fault type. Cumulatively for all releases 45% of the safety-critical failures were caused by coding faults, followed by 24% caused by requirement faults, 15% by interface & integration faults, and 12% by design faults.	Not explored	5.3
RQ4	The median number of failures caused by coding faults was stable across releases. Cumulatively for all components, requirements faults, coding faults, and integration & interface faults contributed from 82% to 87% of failures across different releases.	Not explored	6.1
RQ5	At component level, there is no correlation between the number of failures in subsequent releases. This is likely due to added functionality.	Results are not consistent (Biyani and Santhanam, 1998; Ostrand and Weyuker, 2002; Pighin and Marzona, 2003).	6.2
RQ6	At component level, cumulatively over all releases and for individual releases R1 - R4, the number of post-release failures is positively correlated with the number of pre-release failures.	Results are consistent with some works (Biyani and Santhanam, 1998; Andersson and Runeson, 2007; Grbac et al., 2013; Zimmermann et al., 2007; Shihab et al., 2010), but not consistent with others (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002).	6.3

Table 6 summarizes the results for all research questions explored in this paper. Findings related to RQ1 - RQ3 were tested using χ^2 test and were statistically significant at 0.05 significance level. Findings related to RQ4 - RQ6 were based on descriptive statistics.

In summary, integrating the results from multiple research questions we make the following observations:

1. Only a few fault types were responsible for the majority of failures pre-release and post-release, and across releases. Analysis and testing activities detected the majority of failures caused by each fault type, with analysis detecting slightly more faults for all fault types except coding faults.
2. The relative contribution of each fault type differed for pre-release and post-release failures. Post-release (i.e., on-orbit) failures were most heavily associated with coding faults.
3. The percentage of safety-critical failures was small overall and their relative contribution increased on-orbit. Similarly to on-orbit failures, safety-critical failures were most commonly caused by coding faults.

4. Components that experienced high number of failures in one release were not necessarily among high failure components in the subsequent release.
5. Components that experienced more failures pre-release were more likely to fail post-release, overall and for each release.

The benefits from the findings presented in this paper to the software development and quality assurance of the NASA mission and (potentially) other long lived, critical systems are as follows. First, detailed results of our analysis, including lessons learned, were regularly communicated to the independent verification and validation team. While some results confirmed the anecdotal knowledge of the team members or provided additional insights based on quantification of observed phenomena, other results of our analysis revealed unknown patterns and unusual dependencies in the data that added value for sustained engineering of the mission.

Second, the software used in NASA missions shares features with many other high integrity domains, such as only partially understood operational environments, timing constraints for correct operations, and a high degree of autonomy. This leads us to believe that mining bug and change tracking systems is helpful for building knowledge base that can be reused on other, similar systems.

Third, the results presented in this paper indicate that, for long lived, critical systems, analysis of software faults and failures provides useful information for maintaining the deployed systems, as well as for improving the efficiency of detecting faults and revealing failures during the incremental development and evolution of the software system across multiple releases.

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

– **Compile and regularly update a list of common coding faults.** Significant percentage of the total number of failures (i.e., 36%) and more that 50% of the post-release failures in our case study were due to coding faults. Even more, coding faults were responsible for close to half of the total number of safety critical failures and more than a half of the safety critical post-release failures. 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 and have potential to significantly improve system's quality in a cost effective way.
– **Integrate requirements engineering throughout the life cycle, including post-release.** Requirements faults were the second most significant overall root cause of software failures in our case study. They were the root cause for around 35% of the pre-release faults (same percentage as coding faults), and were a reason for around 14% of post-release failures, which indicates that the process of requirements discovery, specification, analysis, and verification continues post-release.
– **Carry on interface analysis and integration testing.** Component analysis and unit testing are not enough. More interface analysis and integration testing will help avoiding and detecting data problems (the third most significant root cause overall and for pre-release failures) and integration faults (which caused more than 10% of post-release failures). The fact that the relative contribution of failures caused by *integration faults* increased *on-orbit* suggested that components may be interacting in unexpected ways and that conducting more integration testing may be beneficial.
– **Allocate more verification and validation resources and embed fault-tolerant mechanisms in components that experience high number of pre-release failures.** In our case study the failure prone components pre-release remained among most failure prone components post-release. Due to the mission critical nature of the software and the facts that it operates with a high degree of autonomy in only partially understood environment, preventing and tolerating post-release failures is of crucial importance.

Our last set of conclusions is related to future work. It is evident that current change tracking systems lack data and additional information that may contribute towards better understanding of the observed phenomena and establishing cause-effect relationships. Software engineering research and practice communities should aim to improve the quality and quantity of collected empirical data, in tune with development and verification and validation approaches. Further research may shed additional light to currently conflicting findings and help us distinguish phenomena that are project specific from those that tend to be valid across multiple projects.

Appendix A: Background on χ^2 test for contingency tables

For any two attributes (i.e., random variables X and Y with n and m categories, respectively) we use the χ^2 test to explore if the distributions of failures across the m categories of Y is the same for each of the n samples (i.e., categories) in X. That is, we test the null hypothesis which specifies no association between the two bases of classification. We build an m x n contingency table using the observed frequencies for each pair of categories X_i and Y_j, and then calculate the standard χ^2 statistic which can be approximated by a chi-square distribution with $(m-1)(n-1)$ degrees of freedom. When used for contingency comparisons, the chi-square test is a non-parametric test, since it compares entire distributions rather than parameters of distributions.

Note that the χ^2 test is applicable to data in a contingency table only if the expected frequencies are sufficiently large. Specifically, for contingency tables with degrees of freedom greater than one the 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 (Cochran, 1954). For rare categories that are not of interest, categories can be combined in order to increase the expected frequencies in various cells. However, rare categories that are considered important will necessitate larger sample size, both to be able to run the χ^2 test and to achieve adequate power (Kraemer and Thiemann, 1987).

The power of a statistical test is the probability that the false null hypothesis will be rejected, that is, that the test will recognize an existing pattern in the data. Although an intuitive response may be to choose a power value around 99% to be almost certain of demonstrating significance if the alternative hypothesis is true, that is usually impossible because the number of instances required per category is going to be prohibitive. Generally, statistical power in the 70% - 90% range is considered acceptable (Kraemer and Thiemann, 1987).

Once the χ^2 statistic has been calculated, we determine the probability under the null hypothesis of obtaining a value as large as the calculated χ^2 value. If the probability is equal to or less than the significance level α (in our case $\alpha = 0.05$), 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. Therefore, we also measure the extent of the correlation between the two variables. 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 attributes is categorical (i.e., given on a nominal scale). The contingency coefficient C does not require underlying continuity for the various categories used to measure either one or both attributes. Even more, it has the same value regardless of how the categories are arranged in the rows and columns. The contingency coefficient C is calculated as:

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

where N is the total number of observations and the value of χ^2 statistics is computed based on the contingency table (Seigel and N. J. Castellan, 1988). 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.

It is important to 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), we ensure that the range will be between 0 and 1, and hence, C^* values for different sized tables can be compared (Blaikie, 2003).

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

Acknowledgements

This work was funded in part by the NASA Office of Safety and Mission Assurance, the Software Assurance Research Program under grant managed through NASA IV&V Facility in Fairmont, WV. Goseva-Popstojanova's work was also supported in part by the National Science Foundation grant 0916284 with funds from the American Recovery and Reinvestment Act of 2009 and by the WVU ADVANCE Sponsorship Program funded by the National Science Foundation ADVANCE IT Program award HRD-100797.

We 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, Gary Lovstuen, Tom Macaulay, Debbie Miele, Lisa Montgomery, James Moon, Don Ohi, Chad Pokryzwa, David Pruett, Timothy Plew, Scott Radabaugh, David Soto, and Sarma Susarla.

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 and the funding agencies (NASA and NSF).

References

Andersson, C., Runeson, P., 2007. A replicated quantitative analysis of fault distributions in complex software systems. IEEE Trans. Software Eng. 33 (5), 273–286.

Bell, R., Ostrand, T. J., Weyuker, E. J., 2006. Looking for bugs in all the right places. In: Int'l Symp. Software Testing and Analysis. pp. 61–72.

Biyani, S. H., Santhanam, P., 1998. Exploring defect data from development and customer usage on software modules over multiple releases. In: 9th IEEE Int'l Symp. Software Reliability Engineering. pp. 316–320.

Blaikie, N. W. H., 2003. Analyzing Quantitative Data. SAGE Publication Ltd.

Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., Wong, M., 1992. Orthogonal defect classification – a concept for in-process measurement. IEEE Trans. Software Eng. 18 (11), 943–956.

Christmansson, J., Chillarege, R., 1996. Generation of an error set that emulates software faults based on field data. In: 26th Int'l Symp. Fault Tolerant Computing. pp. 304–313.

Cochran, W. G., 1954. Some methods for strengthening the common tests. Biometrics 10, 417–451.

Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences. Lowrence Erlbaum Associates, Inc, Hillsdale, New Jersey.

Duraes, J. A., Madeira, H. S., 2006. Emulation of software faults: A field data study and a practical approach. IEEE Trans. Software Eng. 32 (11), 849–867.

Eldh, S., Punnekkat, S., Hansson, H., Jonsson, P., 2007. Component testing is not enough – a study of software faults in telecom middleware. In: Testing of Software and Communicting Systems, LNCS 4581. pp. 74–89.

Fenton, N., Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE Trans. Software Eng. 26 (8), 787–814.

Goseva-Popstojanova, K., Hamill, M., Perugupalli, R., 2005. Large empirical case study of architecture-based software reliability. In: 16th IEEE International Symposium on Software Reliability Engineering. pp. 43–52.

Grbac, T. G., Runeson, P., Huljenic, D., 2013. A second replicated quantitative analysis of fault distributions in complex software systems. IEEE Transactions on Software Engineering 39 (4), 462 – 476.

Hamill, M., Goseva-Popstojanova, K., 2009. Common trends in fault and failure data. IEEE Trans. Software Eng. 35 (4), 484–496.

Hamill, M., Goseva-Popstojanova, K., 2013. Exploring the missing link: An empirical study of software fixes. Softw. Test. Verif. Reliab.Published online, DOI: 10.1002/stvr.1518.

ISO/IEC/IEEE 24765, 2010. Systems and Software Engineering Vocabulary.

Kitchenham, B., Apr. 2008. The role of replications in empirical software engineering–a word of warning. Empirical Softw. Engg. 13 (2), 219–221.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., Emam, K. E., Rosenberg, J., Aug. 2002. Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. 28 (8), 721–734.

Kraemer, H. C., Thiemann, S., 1987. How Many Subjects? Statistical Power Analysis in Research. SAGE Publications, Inc.

Leszak, M., Perry, D., Stoll, D., April 2002. Classification and evaluation of defect in a project retrospective. J. Systems and Software 61, 173–187.

Lutz, R. R., Mikulski, I. C., March 2004. Empirical analysis of safety critical anomalies during operation. IEEE Trans. Software Eng. 30 (3), 172–180.

NASA-GB-8719.13, 2004. NASA Software Safety Guidebook.

Ostrand, T. J., Weyuker, E. J., 2002. The distribution of faults in a large industrial software system. In: ACM Int'l Symp. Software Testing and Analysis. pp. 55–64.

Ostrand, T. J., Weyuker, E. J., Bell, R., 2005a. Where the bugs are. In: ACM SIGSOFT Int'l Symp. Software Testing and Analysis. pp. 86–96.

Ostrand, T. J., Weyuker, J., Bell, R., 2005b. Predicting the location and number of faults in large software systems. IEEE Trans. Software Eng. 31 (4), 340–355.

Petersen, K., Wohlin, C., 2009. Context in industrial software engineering research. In: 3rd International Symposium on Empirical Software Engineering and Measurement. ESEM '09. pp. 401–404.

Pighin, M., Marzona, A., 2003. An empirical analysis of fault persistence through software releases. In: Int'l Symp. Empirical Software Engineering. pp. 206–211.

Robson, C., 2002. Real World Research. Blackwell.

Runeson, P., Höst, M., Apr. 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Engg. 14 (2), 131–164.

Runeson, P., Höst, M., Rainer, A., Regnell, B., 2012. Case Study Reserach in Software Engineering: Guidelines and Examples. John Wiley & Sons, Inc., Hoboken, NewJersey.

Seigel, S., N. J. Castellan, J., 1988. Non Parameteric Statistics for the Behavoiral Sciences. McGraw-Hill.

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., Hassan, A. E., 2010. Understanding the impact of code and process metrics on post-release defects: a case study on the Eclipse project. In: 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 4:1–4:10.

Shull, F. J., J. C. Carver, S. V., Juristo, N., 2008. The role of replications in empirical software engineering. Empirical Software Engineering 13 (2), 211–218.

Yin, R. K., 2014. Case Study Research: Design and Methods. SAGE Publication Ltd, Thousand Oaks, CA.

Yu, W. D., 1998. A software fault prevention approach in coding and root cause analysis. Bell Labs Technical J. 3 (2), 3–21.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., Vouk, M. A., 2006. On the value of static analysis for fault detection in software. IEEE Trans. Softw. Eng. 32 (4), 240–253.

Zhivich, M., Cunningham, R. K., 2009. The real cost of software errors. IEEE Security & Privacy 7 (2), 87–90.

Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for Eclipse. In: 3rd International Workshop on Predictor Models in Software Engineering. paper #9, 7 pages.