Common Trends in Software Fault and Failure Data

Maggie Hamill, *Member*, *IEEE*, and Katerina Goševa-Popstojanova, *Senior Member*, *IEEE*

Abstract—The benefits of the analysis of software faults and failures have been widely recognized. However, detailed studies based on empirical data are rare. In this paper, we analyze the fault and failure data from two large, real-world case studies. Specifically, we explore: 1) the localization of faults that lead to individual software failures and 2) the distribution of different types of software faults. Our results show that individual failures are often caused by multiple faults spread throughout the system. This observation is important since it does not support several heuristics and assumptions used in the past. In addition, it clearly indicates that finding and fixing faults that lead to such software failures in large, complex systems are often difficult and challenging tasks despite the advances in software development. Our results also show that requirement faults, coding faults, and data problems are the three most common types of software faults. Furthermore, these results show that contrary to the popular belief, a significant percentage of failures are linked to late life cycle activities. Another important aspect of our work is that we conduct intra- and interproject comparisons, as well as comparisons with the findings from related studies. The consistency of several main trends across software systems in this paper and several related research efforts suggests that these trends are likely to be intrinsic characteristics of software faults and failures rather than project specific.

Index Terms—Software faults and failures, fault location, fault types, software fault distribution, software reliability, empirical studies.

1 Introduction

All software contains faults and undoubtedly some of these faults will result in failures. The consequences of such failures are sometimes unacceptable. Better understanding and quantification of the relationships between software faults and failures are essential to more efficient detection and elimination of faults and prevention of failures, and thus, to improvement of software quality.

Since many terms in this area are often used interchangeably, or simply misused, we provide the definitions of the terms the way they are used in this paper. These definitions are somewhat adapted versions of the IEEE Software Engineering definitions [26].

A *failure* is a departure of the system or system component behavior from its required behavior. On the other hand, a *fault* is an accidental condition, which if encountered, may cause the system or system component to fail to perform as required. Thus, faults represent problems that developers see, while failures represent problems that the users (human or computer) see. Not every fault corresponds to a failure since the conditions under which fault(s) result in a failure may never be met. It should be emphasized that faults can be introduced at any phase of the software life cycle, that is, they can be tied to any software artifact (e.g., requirements, design, and source code).

A few examples of how researchers and practitioners differ in their usage of the terminology are given next.

- Bugs refer to faults in the code.
- Anomalies refer to conditions that deviate from expectations based on requirements specifications, design documents, user documents, standards, etc., or from someone's perceptions or experiences. In other words, an anomaly does not have to refer to a failure; it can, for example, refer to an unexpected behavior in the eyes of the operator.
- Defects sometimes refer only to faults. However, defects often refer collectively to faults and failures, and even anomalies.

Throughout this paper, we strictly use the terms fault and failure.

Although some studies on software faults and failures have been conducted, the lack of publicly available fault and failure data and the lack of published studies conducted on such data have been recognized by many. This is perhaps best stated in [18].

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

In [11], the authors commented that "The (software defect prediction) models are weak because of their inability to cope with the, as yet, unknown relationship between defects (faults) and failures." The relationship between faults and failures is complex because it is possible that a fault may result in many failures and vice versa. Since associating failures with the faults that caused them is a

- *The authors are with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Engineering Science Building, PO Box 6109, Morgantown, WV 26506.*
E-mail: mhamill@mix.wvu.edu, katerina.goseva@mail.wvu.edu.

Manuscript received 20 June 2008; revised 13 Nov. 2008; accepted 31 Dec. 2008; published online 14 Jan. 2009.
Recommended for acceptance by B. Littlewood.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-06-0206.
Digital Object Identifier no. 10.1109/TSE.2009.3.

0098-5589/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

difficult task, simplifying assumptions and heuristics are often used. In our previous work [14], [15], which explored the adequacy, accuracy, scalability, and uncertainty of architecture-based software reliability models based on two large-scale open-source case studies, we noticed that certain assumptions do not appear to hold true and some heuristics are not justified.

Extracting data from modification requests, change documentation and bug reports are becoming more common. According to [22], "bug reports from testing and operations are a rich, underused source of information about requirements." We believe that bug reporting and change tracking systems are valuable sources of information that hold high potential for conducting empirical studies which will benefit both the research and practitioner communities.

In this paper, we systematically investigate and characterize software faults and failures based on data extracted from the change tracking systems of two large-scale, real-world software projects. The broad goal of our work is to contribute to the body of empirical knowledge about faults and failures. In order to do so, we not only work toward our specific research goals, but also compare our results with related studies whenever possible. The two main research questions explored in this paper are as follows:

1. *Are faults that cause individual failures localized, that is, do they belong to the same file, component, top level component, etc.?*
2. *Are some sources of failures (i.e., types of faults) more common than others?*

These research questions are of particular interest from a software engineering perspective. Although some evidence of nonlocalized faults has been observed by others, this phenomenon has not been specifically investigated and quantified. To the best of our knowledge, the only exception is an almost 25-year-old study [2], which reported that most fixes are confined to single, small, well-encapsulated modules. On the other hand, in [17], the author commented that many major software failures have been traced to unanticipated combinations of otherwise minor problems, but did not provide specific data that support this claim. Additionally, thorough investigation of several catastrophic accidents of safety/mission critical systems [16], [20] led to the conclusion that these accidents can be attributed to combinations of faults, system misconfigurations, and procedure violations.

Investigating the sources of failures is interesting because learning why and how the most common types of faults are introduced into the system will allow developers to focus their efforts on preventing and eliminating faults in the most effective ways, at the most effective time (i.e., during design, implementation, integration, etc.). One of the widespread software engineering beliefs, which dates back to some of the earliest empirical studies [2], [4], [9], has been that the majority of faults are inserted during requirements and design activities. Using these studies as evidence, a recent book [10] formulated the following law: "Errors (i.e., faults) are most frequent during the requirements and design activities and are more expensive the later they are removed." Thus, our second research goal is to investigate the distribution of fault types on current large-scale projects, particularly addressing the identification of dominating sources of failures and their consistency with some of the popular software engineering beliefs.

In addition to answering these research questions, by using two case studies and comparing our work with related work, we seek to determine if the observations made in this paper are likely to be intrinsic characteristics of software faults and failures rather than dependent on specific projects. Detailed analysis of software failures across various domains and projects is essential to certifying dependable systems. However, such studies seem to be few and far between [18].

The paper proceeds by reviewing the related work and listing our contributions in Section 2. Then, we describe the case studies, the available data, and the methods used to extract the data in Section 3. The empirical results and the answers to the research questions are presented in Section 4. Finally, we discuss the threats to the validity of the study in Section 5 and summarize our results in Section 6.

2 Related Work and Our Contributions

Studies focused on software faults can be dated back to the mid 1970s. However, as pointed out in [12], "there continues to be a dearth of published data relating to the quality and reliability of realistic commercial software systems."

Most of the previously published work in the area limits the focus to faults and conducts the analysis for different reasons. In [12], based on the data from two releases of a large commercial telecommunications system, the authors investigated four basic software engineering hypotheses:

1. the Pareto principle (i.e., a small number of modules contain the majority of faults);
2. fault persistence through the testing phase and pre and postrelease;
3. the relationship between lines of code and faults;
4. similarities in fault densities within project phases and across projects.

The same four hypotheses were tested in [1] on a telecommunications system which differed in size, system type, and the development method used, leading to similar results. The authors in [1] specified that, when a failure was corrected by modifying n modules, n distinct faults were counted.

An empirical study of a large industrial inventory control system which used faults extracted from modification requests to predict fault-prone files was conducted in [23]. However, since the modification records did not differentiate between changes made to fix faults and those made for other reasons (e.g., adding functionality or enhancements), it was assumed that only modification requests that required changes in one or two files were associated with faults. In [24], the earlier proposed statistical model was applied on 17 releases of the inventory control system and nine releases of a service provisioning system. In both systems, the top 20 percent of the files identified as fault-prone contained 83 percent of the total faults. Neither [23] nor [24] have investigated the fault types.

The analysis of the phase where faults were introduced into the switching telecommunication software containing several millions lines of source code presented in [25] showed that nearly half of the 600 faults were related to coding faults and the majority of them could have been prevented. Consequently, the paper provided a detailed list of countermeasures for reducing these type of faults.

In [7], 408 defect (i.e., fault) reports from an IBM operating system were classified using Orthogonal Defect Classification (ODC) [6]. The purpose was to map software faults to injectable errors. ODC was also used in [8] to classify 668 faults from 12 open-source projects for the purpose of quantifying fault types in order to obtain more accurate representations of actual faults during fault injection. Interestingly, there was a strong consistency in the fault distributions across ODC defect types observed in [8] with the work presented in [7].

An ODC-like technique was also used in [21] to classify safety-critical anomalies observed postlaunch on seven NASA spacecraft systems. Although it was not its primary goal, this work indicated the complex relationships between software faults and failures by recognizing the existence of multiple triggers (i.e., event(s) or condition(s) leading up to the failure) and multiple targets (i.e., fixes). However, since ODC does not allow for multiple triggers or multiple targets, the authors in [21] recorded and analyzed only the most proximate event and only the first fix.

Another study which classified defect modification requests [19] also showed that many failures can only be accurately represented using multidimensional triggers, and therefore, addressed some of the limitations of ODC by developing a new classification scheme which is not orthogonal, but allows for multiple triggers.

In [14], [15], we analyzed the empirical data from two open-source case studies in order to test the assumptions and understand the limitations of architecture-based software reliability models. The results showed that the models provide very accurate estimation of software reliability for a subset of failures which can clearly be attributed to a single component, but also that a significant portion of the failures is traced to more than one component. Exploring the failures which led to fixing faults in multiple components and more detailed analysis of the fault types were out of the scope of those papers.

In this paper, we analyze empirical data related to software faults and failures for an open-source application and a large-scale NASA mission consisting of multiple software applications developed at two different locations. The methods used to store the change data and the information being stored differed between the two projects. Therefore, we had to complete the complex process of mapping failures to faults for each study.

The main contributions and uniqueness of the work presented in this paper consist of the following:

1. We tie failures to faults for two case studies. The lack of detailed studies of faults and failures relationships in part is due to the fact that establishing links between faults and failures is not a trivial task. Since little is known about the fault-failure relationships, simplifying assumptions and heuristics have been used (e.g., [13], [21], [23], [24]) and analysis has often been restricted only to faults (e.g., [7], [8], [19], [25]).
2. We characterize and quantify failures based on the location of the changes made to fix the corresponding faults. Even more, we use a formal statistical hypothesis test to confirm that the distribution of the spread of fixes throughout files is the same for the open-source software application and two different data sets from the NASA mission. Although the related work indicated that faults that lead to individual failures may not be localized [1], [14], [15], [21], to the best of our knowledge, the evidence that multiple faults are often necessary to cause a single failure has never before been quantified, or even systematically explored.
3. We further study the distribution of the sources of failures (i.e., types of faults) for the NASA mission, which has identified the source of the failures in 93 percent of the cases. In particular, we study a set of over 2,800 Software Change Requests created when the system failed to conform to a requirement (i.e., indication of software failures), collected throughout the software life cycle (i.e., development, testing, and postrelease), over a period of almost 10 years. To the best of our knowledge, this is the largest data set considered so far in the published literature. Surprisingly, despite the fact that some subsystems are developed at different locations using different technical methods and software development processes, the results with respect to the dominating sources of failures are very consistent.
4. We specifically address the internal and external validity of the study by making intra and interproject comparisons. By using two case studies and extensively comparing our findings with recently published related studies, we show that several trends observed in this paper are not project specific. Rather, they seem to be intrinsic characteristics of software faults and failures.

According to [3], the main signs of maturing in software engineering experimentation are: the level of sophistication of the goals of an experiment and its relevance to understanding interesting (e.g., practical) things about the field and observing a pattern of knowledge building from a series of experiments. We believe that the results of the research work presented in this paper contribute toward the maturity of software engineering as an empirical discipline.

3 Empirical Exploration of the Fault-Failure Relationships

In this section, we provide the details of the case studies used in this paper, the open-source application GCC, and a large-scale NASA mission, specifically emphasizing the description of the available information and the methods we used to extract the necessary data from the project repositories. In each study, we used some sort of change documents to tie failures to the faults that caused them. It should be noted that the domains, development methods, implementation languages, change information available, and the way the information was stored are very different for the two case studies. Even within the NASA mission, some subsystems are developed at different locations, using varying development and change tracking methods.

3.1 Basic Facts of the Case Studies

The first case study is the C preprocessor part of the C compiler from the GNU Compiler Collection (GCC), which will be referred to as GCC. This is a mature, heavily used application consisting of over 300,000 lines of code, which can be divided based on functionality into 13 well-defined components [14]. Each component contains between 1 and 32 files.

The second case study is flight software from a mature NASA mission, which includes multiple software applications undergoing iterative development, consisting of millions of lines of code in over 8,000 files. The overall system is divided into Computer Software Configurations Items (CSCIs), each containing between 81 and 1,368 files.[1] These CSCIs span an entire application range from command and control; generation, distribution, storage, and management of supporting utilities; and failure detection, isolation, and recovery; to human-computer interfaces and scientific research support. CSCIs are further divided into Top-Level Computer Software Components (TLCSCs), which are then divided into Computer Software Components (CSCs).

3.2 Data Availability and Extraction

Although GCC and the NASA mission both store the data necessary to link failures to the faults that caused them, the information is not readily available. Moreover, the level of detail and type of information stored differed greatly between the two studies, and even within the NASA mission. For each case study, associating failures with the corresponding faults was not trivial since the change data were not stored for this purpose. Therefore, we had to develop unique methods that map the failures to the faults that caused them. We consider both pre and postrelease failures for each case study. Next, we describe the process of extracting and linking the necessary data for each case study.

GCC. Here, we briefly present the experimental setup used to collect failure data for GCC, which was originally presented in [14]. Like many open-source projects, GCC uses Concurrent Version Control (CVS) and maintains the latest version of the code base, as well as the history of previous releases. A regression test suite, test logs, and change logs, maintained by the GCC development team, are available with each version of GCC. Furthermore, the regression test suite comes with test programs, drivers for these programs, and checkers that compare the test output to the expected results. The drivers and the checkers played the role of a test oracle in our study. In our experimental setup, we executed the C proper part of version 3.2.3 on a total of 2,126 test cases from the regression test suite of the version 3.3.3. Out of these 2,126 test cases, 111 failed. Running newer test cases on an older version of the code allowed us to observe more failures. After observing the failures, rather than making any assumptions or using heuristics, we developed accurate methods to identify the faults that led to observed failures.

These methods involved linking textual entries from test logs to textual entries in change logs based on who entered the information in the log and when. However, since the entries in the change logs often did not identify the reason for the change (e.g., fixing a fault, adding functionality, etc.), it was necessary to further investigate the changes. Additional information was obtained through correspondence with the developers and by linking problem report numbers from the logs to entries in the bug tracking database and/or entries in the version control system logs. The details of these methods can be found in [14].

Using our methods, we identified the faults that led to 85 of the 111 failures of the GCC C compiler version 3.2.3. For these 85 failures, we ensured that changes made to the software actually fixed the corresponding faults by rerunning the failed test cases and confirming that they passed after the changes had been made. In addition, we identified and excluded from further analysis seven test cases which failed because they were designed to test functionality in version 3.3.3 that was not available in version 3.2.3. Some of the remaining 19 unresolved failures are due to faults that are either not known or not yet fixed. The lack of consistency (or discipline) in the change tracking process made it difficult to identify all faults associated with all failures.

NASA mission. For the NASA mission, we could not run the software. However, the change tracking system used by the mission allowed us to map some failures detected during development, testing, and operations to the changes made to fix the faults which led to these failures. In particular, we focused on Software Change Requests (SCRs) that were created when the system failed to conform to a requirement. It should be noted that the NASA mission follows a process of creating and addressing SCRs to ensure that the corresponding changes made to software artifacts are indeed fixing the faults. The data necessary to map failures to faults were extracted from two data sets: NASA_set_1 and NASA_set_2.

NASA_set_1 is based on the data entered in the change tracking database used by the entire mission. This database contains numerous tables representing various documents that detail multiple types of change requests for all CSCIs. Since the change tracking system was not intended to be used for this purpose, extracting the data to link failures to faults from NASA_set_1 was not simple. However, with the support of NASA personnel, we wrote a complex query that spanned several database tables and extracted data from three different types of change tracking documents to automatically extract the necessary information that links failures to faults that caused them. As of our last data dump, the change tracking system contained SCRs collected in a time period of almost 10 years, associated with 21 CSCIs cumulatively having over 8,000 files. Over 2,800 SCRs were entered because of nonconformance to a requirement throughout the life cycle, including development, testing, and operation. Based on the data available to us, we were able to link failures to faults that caused them for 356 SCRs associated with 11 CSCIs.

[1] All the numbers given are from the subset of data made available to us and do not necessarily represent the entire project.

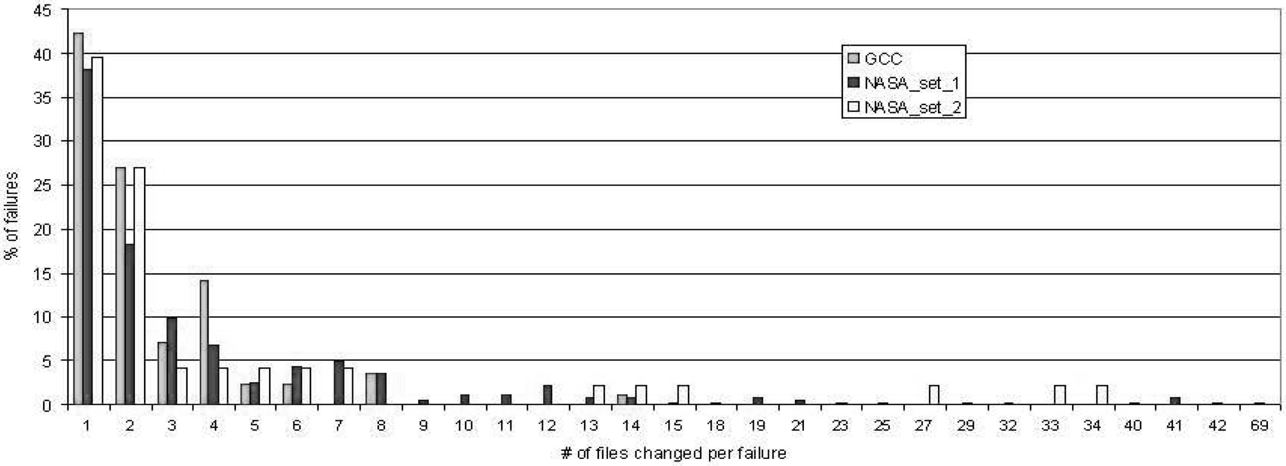

Fig. 1. Fault localization at the file level.

NASA_set_2, which was provided in addition to NASA_set_1, consists of developers' in-house documentation which lists the files that were changed to address the nonconformance SCRs. Therefore, the process of linking failures to faults that caused them was straightforward for NASA_set_2, but data were only available for 48 SCRs associated with four CSCIs. It is important to note that we ensured that the analysis does not include duplicates, that is, data points in NASA_set_1 and NASA_set_2 are mutually exclusive. In other words, 356 SCRs in NASA_set_1 do not include any of the 48 SCRs in NASA_set_2. Even more, 11 CSCIs considered in NASA_set_1 are different from the four CSCIs in NASA_set_2.

4 Empirical Results and Analysis

In this section, we present our results as they pertain to the research questions. In some cases, due to the differences in the available data, the analysis could only be conducted on one study. Whenever possible, we compare our results with the results of similar studies in the field.

4.1 Are Faults that Cause Individual Failures Localized?

By associating observed failures with the faults that caused them, we are able to explore the localization of software faults that led to individual failures. Specifically, we explore the *number* of files changed to prevent the failure from reoccurring and the *spread* of the changes across files. For this analysis, we consider all failures for which the necessary data to map them to the corresponding faults existed. In total, we study the localization of faults for 85 of the 111 (83 percent) failures associated with 13 components of GCC, 356 of the 2,500 (14 percent) nonconformance SCRs associated with 11 CSCIs from NASA_set_1, and 48 of the 281 (17 percent) nonconformance SCRs associated with four CSCIs from NASA_set_2.[2] We analyze the location of all faults at various levels (i.e., file level and component level) for each failure that was successfully mapped. Further, we make both intra- and interproject comparisons.

Fig. 1 shows the percentage of failures mapped to fixes made in $k = 1, 2, \ldots, 69$ files, respectively, for GCC, NASA_set_1, and NASA_set_2 data sets. For example, 42 percent of the GCC failures, 38 percent of the NASA_set_1 failures, and 40 percent of the NASA_set_2 failures were corrected by fixing one file, while 27 percent of the GCC failures, 18 percent of the NASA_set_1 failures, and 27 percent of the NASA_set_2 failures were corrected by fixing two files. It is obvious that the results are remarkably consistent across all three data sets. Thus, 58, 62, and 60 percent of failures map to fixes in more than one file, while 31, 44, and 33 percent of failures map to fixes in more than two files for GCC, NASA_set_1, and NASA_set_2 data sets, respectively.

Based on this evidence, we decided to test statistically the following formal hypothesis related to the localization of faults that led to individual failures:

> *The data from the GCC application and both NASA data sets have the same distribution of the spread of fixes across files.*

By creating a contingency table and using a standard chi-square test (where the test statistics was calculated to be $\chi^2 = 65.00$ with 56 degrees of freedom), we determined that the hypothesis cannot be rejected at the 0.05 significance level. In other words, the formal hypothesis testing proves the consistency of the observation made from Fig. 1: *For one open-source case study and two NASA mission data sets, faults (i.e., fixes made to prevent failures from reoccurring) often are spread across multiple files, that is, are not localized.*

The GCC C compiler contains just over 100 files and it is more comparable in size to individual CSCIs from the NASA mission, each containing from 81 to 1,386 files. Therefore, we next compare the fault localization on a higher level of abstraction. Thus, Fig. 2 presents the percentages of GCC failures and percentages of failures from four CSCIs from the NASA_set_1 mapped to the number of components/TLCSCs changed to fix individual failures. The analysis is limited to these four CSCIs from the NASA_set_1 because TLCSC level data were only available for these CSCIs. Once again, a significant percentage of failures (i.e., 33 percent of GCC failures and between 6 and 36 percent of the failures from the four CSCIs of

[2] The sets of CSCIs from NASA_set_1 and NASA_set_2 are disjoint sets.

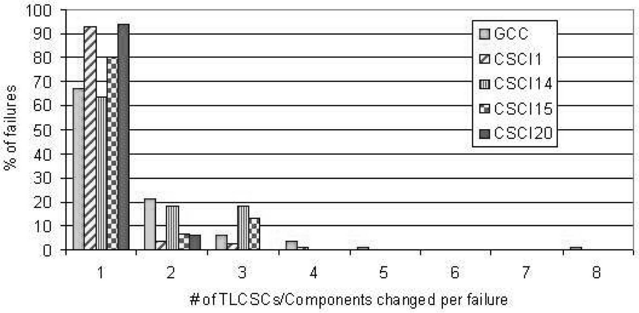

Fig. 2. Fault localization at the component/TLCSC level.

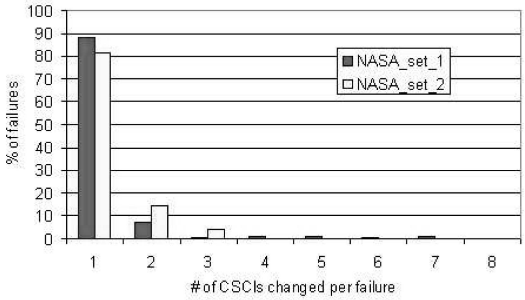

Fig. 3. Fault localization at the CSCI level.

NASA_set_1) is associated with nonlocalized faults, that is, changes were made in multiple components/TLCSCS.

Last, in Fig. 3, we compare failures mapped to fault locations at the CSCI level for the two NASA data sets. Although these data sets store the information for different purposes using different mechanisms and represent CSCIs developed at different locations, the results are still very consistent. In particular, 12 percent of NASA_set_1 failures and 19 percent of NASA_set_2 failures map to changes in more than one CSCI, and 5 and 4 percent, respectively, map to changes in more than two CSCIs.

The following examples illustrate the importance of rigorous empirical studies based on current, large-scale software systems, and the implications of the results presented in this section.

- First, the fact that 58-62 percent of failures in our data sets map to faults in more than one file clearly is not consistent with the only known quantitative result related to fault localization [2], which was published almost 25 years ago. Thus, the study based on an application with approximately 90,000 lines of Fortran code [2] reported that 89 percent of errors were corrected by changing only one module (where 97 percent of modules were very small, with less than 400 lines of code). Explaining these significantly different results is not straightforward due to a lack of details related to the software studied in [2], as well as a lack of other similar studies from that time that would indicate whether fault localization was a common characteristic. Some of the probable reasons may be due to characteristics of the software system studied in [2], such as the much smaller size and complexity (i.e., 90,000 lines of code compared to millions of lines of code in case of the NASA mission) and the large amount of code reused from similar past projects.
- Second, we address the heuristic used in [23] to simplify the process of fault identification from modification records. Thus, in the absence of data that would distinguish the changes made to fix faults from other changes such as, for example, planned improvements or enhancing the functionality, the authors used a heuristics which assumed that only changes made to one or two files were related to fixing faults. This allowed using a simple automatic way to count modification records that may have been related to fixing faults. This heuristic is implicitly based on the assumption that changes made to fix software faults are very much localized, that is, spawn at most two files. Our results (see Fig. 1) show that at least 30 percent of the failures in each data set which are mapped to fixes in more than two files would have been overlooked by a method based on this heuristic.
- The last example is related to the assumption that components fail independently and each component failure leads to a system failure, which is common to most existing architecture-based software reliability models [13]. As can be seen from Fig. 2, our results show that we can be confident that this assumption is valid for 67 percent of GCC C compiler failures and for 63-94 percent of failures associated to four CSCIs from the NASA mission. In other words, 6-37 percent of failures observed in the data sources considered in this paper cannot be handled by existing architecture-based software reliability models.

Although it is not one of our main research goals, next we explore the Pareto principle, one of the popular software engineering beliefs which has been shown to hold true in several previous empirical studies [1], [12], [23]. Specifically, we test the following claim:

Most faults lie in a small proportion of the files.

We investigate the Pareto principle only for the GCC case study. This analysis could not be completed for the NASA study due to the fact that based on the data available to us, a significant number of failures could not be mapped to fixes at the file level.

As illustrated in Fig. 4, which shows the percentage of files versus the percentage of faults, *20 percent of the files contain nearly 80 percent of the faults found in GCC release 3.2.3. Further, 47 percent of the files contain 100 percent of the faults.* This result shows an amazing consistency with the software defect reduction rule presented in [5]: "About 80 percent of the defects (i.e., faults) come from 20 percent of the modules, and about half of the modules are defect (i.e., fault) free." Together with results such as [1], [12], [23], our result provides support for the Pareto principle which indicates that the majority of software faults are located in a small portion of the code.

The quantitative results, including the formal statistical hypothesis testing, presented in this section clearly prove that a significant percentage of software failures are associated with changes that spread across the system,

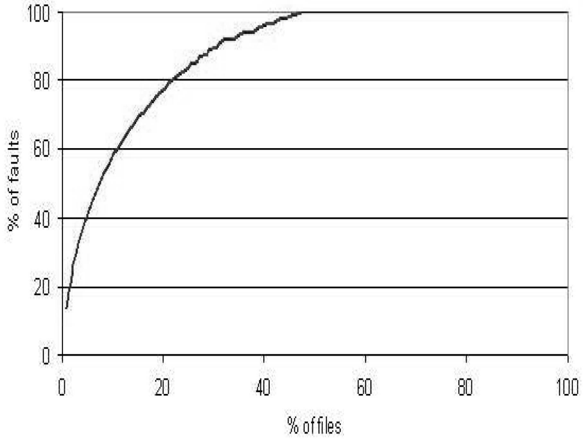

Fig. 4. Pareto diagram showing the percentage of files versus percentage of faults for GCC release 3.2.3.

that is, are due to nonlocalized faults. Failures of this nature are harder to prevent because unexpected combinations of faults are difficult to anticipate and hard to test for. The good news, however, is that, although multiple nonlocalized faults are associated with individual failures, the majority of the faults are still contained in a relatively small part of the system.

4.2 Are Some Sources of Failures (i.e., Types of Faults) More Common than Others?

Our next research question is focused on exploring the distribution of different types of faults and it is aimed at identifying fault types that may dominate as sources of failures. Due to the limited data and inconsistent level of detail describing the changes made to fix faults, we omit GCC from the analysis presented in this section. On the other hand, for the NASA mission, we were able to expand our analysis to 2,858 SCRs associated with 21 CSCIs, entered during development, testing, and operation, and collected over almost 10 years. Each of these SCRs was entered due to nonconformance with requirements and has a *source* field which contains the source of the failure as identified by the analyst addressing the problem reported in the SCR. It should be noted that this set of 2,858 SCRs, annotated with NASA_*, is a superset of the data used in the previous section, that is, it includes all SCRs from the both NASA data sets (NASA_set_1 and NASA_set_2) analyzed in the previous section and additional SCRs that could not be mapped to changes at the file level.

Table 1 presents detailed information for 21 CSCIs considered in this section, ordered by the number of releases they have undergone. As can be seen from this table, CSCIs consist of 81-1,368 files, the number of SCRs entered per CSCI is in the range 9-861, with an average number of nonconformance SCRs per file across the 21 CSCIs in the range 0.03-1.89.

TABLE 1
Distribution of SCRs over 21 CSCIs

CSCI	# of releases	# of files	# of non -conf SCRs	# of non-conf SCRs per file
1	1	207	392	1.89
2	2	200	24	0.12
3	2	287	9	0.03
4	2	228	16	0.07
5	2	269	13	0.05
6	2	321	22	0.07
7	2	289	77	0.27
8	2	270	107	0.40
9	2	125	21	0.17
10	3	356	27	0.08
11	3	444	43	0.10
12	3	277	12	0.04
13	3	599	121	0.20
14	3	280	83	0.30
15	3	81	108	1.33
16	3	169	135	0.80
17	4	587	211	0.36
18	5	552	263	0.48
19	7	747	201	0.27
20	7	1368	861	0.63
21	7	415	112	0.27
Total	-	8,071	2,858	0.35

Fig. 5 shows a scatter plot representing the relationship between the number of files and the number of nonconformance SCRs entered against each CSCI given in Table 1. This scatter plot shows some positive trend which indicates that larger CSCIs tend to have more nonconformance SCRs associated with them. It should be noted that the positive trend does not necessarily establish cause-effect relationship between the two variables. Further study of this relationship is out of the scope of this work.

Our main goal in this section is to explore the sources (or root causes) of failures reported in the 2,858 SCRs entered because of nonconformance to a requirement. Fig. 6 shows these SCRs grouped into 12 categories based on the source of the failure as identified during the fix. The corresponding data values are shown in Table 2. The most common sources of failures are requirements faults and coding faults, each contributing to about 33 percent of the failures. It should be noted that requirements faults include incorrect requirements, changed requirements, and missing requirements. The third largest fault type was related to data problems and it contributed to 14 percent of the failures. Surprisingly, design faults led to less than 6 percent of the failures. Additionally, 4 percent of failures are due to process or procedural issues, 2 percent are due to integration faults, and

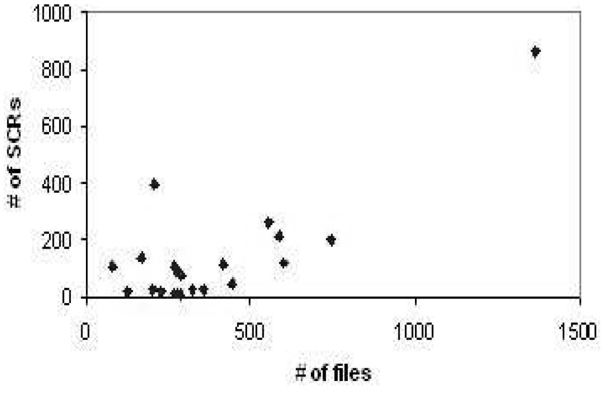

Fig. 5. Scatter plot showing the number of files in each CSCI versus the number of SCRs entered against each CSCIs for all 21 CSCIs from the NASA mission.

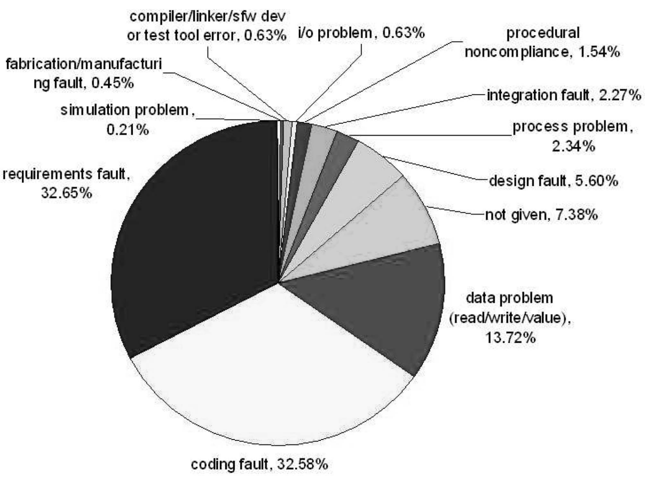

Fig. 6. Pie chart representing the sources of the failures recorded in 2,858 SCRs for 21 CSCIs of the NASA mission.

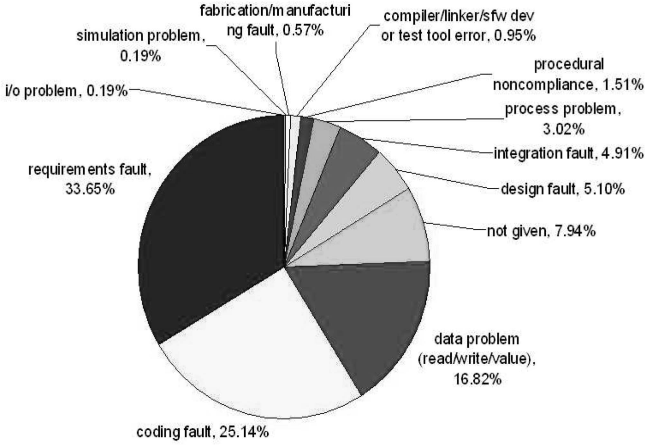

Fig. 7. Pie chart representing the sources of the failures recorded in the 516 SCRs for the seven CSCIs with three releases.

1 percent is due to simulation or testing problems. The source of the failure was not recorded for about 7 percent of the nonconformance SCRs.

The contributions of some of the fault types to the total of 2,858 SCRs in the case of NASA mission, which is a collection of 21 CSCIs with over 8,000 files, differ significantly from the results of some of the earlier empirical studies. For example, the main conclusion of a study done at TRW [4], which was based on 224 faults, was that design faults outweigh coding faults, 64 percent versus 36 percent. Although the percentage of coding faults in our case study is rather consistent (i.e., 33 percent), the design faults contribute significantly less (i.e., only 6 percent). Even more, the faults that originated in the early life cycle (i.e., requirement-related and design faults) together are less than 39 percent, which is significantly less than the 60-70 percent found in [2], [4], [9].

As can be seen from Table 1, the 21 CSCIs considered in this section have undergone different numbers of releases, that is, some CSCIs are much more mature than others. In order to make sure that, by considering all 2,858 SCRs for all 21 CSCIs together, we are not overlooking some phenomena, next we study the distribution of fault types taking into account the number of releases.

TABLE 2
Sources of Failures for 2,858 SCRs for 21 CSCIs of the NASA Mission

Source of Failures	% of SCRs
Requirements fault	32.65
Design fault	5.60
Coding fault	32.58
Data problem	13.72
Integration fault	2.27
I/O problem	0.63
Compilier/linker/sfw dev or test tool error	0.63
Simulation problem	0.21
Procedural non–compliance	1.54
Process problem	2.34
Fabrication/Manufacturing fault	0.45
None identified	7.38

4.3 Comparing the Distribution of Fault Types across CSCIs Grouped by the Number of Releases

According to the developers, CSCIs with exactly three releases (i.e., CSCIs 10-16 in Table 1) show a certain level of stability, so we begin our analysis by focusing on the subset of CSCIs with only three releases. This subset includes seven (33 percent) CSCIs, which are composed of 2,206 (27 percent) files and associated with 516 (19 percent) SCRs. As can be seen in Fig. 7, the distribution of fault types for these seven CSCIs is consistent with the distribution for all 21 CSCIs, with the same three most common types of faults: requirement faults, coding faults, and data problems.

To further explore the consistency of our results, we study the distribution of fault types for six subsets of CSCIs grouped by the number of releases they have undergone. Table 3 provides details for these subsets, including the corresponding percentage of CSCIs, percentage of total files, and percentage of total SCRs associated with the group. The last three columns in Table 3 show that requirements faults, coding faults, and data problems are actually the most common sources of failures for any subset of CSCIs grouped by number of releases. Even more, the corresponding ranges (29-40 percent for requirements faults, 25-59 percent for coding faults, and 4-17 percent for data problems) are very consistent, bearing in mind the wide differences in the number of releases. One of the reasons for this consistency may be due to the fact that NASA mission undergoes an iterative development, with new functionality (i.e., new

TABLE 3
Distribution of the Three Most Common Fault Types across CSCIs Grouped by the Number of Releases

# of rel.	% of CSCIs	% of files	% of SCRs	% Req. faults	% Coding faults	% Data prob.
1	4.76	2.56	13.72	31.12	39.54	13.01
2	38.10	24.64	10.11	39.79	34.60	11.42
3	33.33	27.33	18.51	33.65	25.14	16.82
4	4.76	7.27	7.38	29.38	59.24	4.27
5	4.76	6.84	9.20	31.94	24.71	11.03
7	14.29	31.35	41.08	31.69	30.07	15.42

TABLE 4
Comparison of the Fault Types from NASA Mission with Related Studies

Empirical study	**I** [21] 2004	**II** [25] 1998	**III** [19] 2002	**IV** This study 2009
Description	7 NASA space crafts	Lucent (US) switching system	Lucent (Germany) optical network element	21 CSCIs from a large NASA Mission
Sample Size	199 safety critical post–launch anomalies	600 software faults from several releases	427 pre–release & post–release modification requests	2,858 pre–release & post–release software change requests
% Requirements/ Design	17.1	52.6	46.0	38.2
% Coding	15.6	40.6	40.0	32.6
% Interface/ integration	6.5	6.8	–	16.0
% Procedure/ Process	29.2	–	–	3.9
% Other	27.1	–	14.0	1.9
% Unknown	4.5	–	–	7.4

requirements and code) being added with each release. Surely, the strong consistency of the results between CSCIs grouped by release provides support for the internal validity of this study.

4.4 Comparing the Distribution of Fault Types of NASA Failures with Related Work

In addition to addressing the internal validity of our study by comparing the source of failures across groups of CSCIs with the same number of releases, we compare the source of NASA failures to the classification of faults and failures in the related work. Hence, in this section, we consider the external validity of our results. It should be noted that comparing the empirical results across different studies is a challenging task since the terminology, the level of details provided, and the classification schemes used differ in almost every study.

We chose to compare our results, to the largest possible extent, with the results of several recently published related empirical studies [21], [25], [19]. In what follows, these three studies are labeled Studies I-III, respectively, with our study labeled Study IV. To allow for a meaningful comparison, we have integrated some of the fault types into coarser categories. Grouping our results presented in Table 2 into these categories was done based on our knowledge and support from the NASA personnel. In the case of [21], we used the attribute *Type*, which describes the actual fix that was made, and grouped the categories based on the details provided in [21] and the related paper [22]. Although *Defect-type* classification exists in [19] due to the lack of clear definitions, it was almost impossible to relate different defect types to the ones in the other related studies. Instead, we used the *Phase where defect (i.e., fault) originated* categories which are self-explanatory. Similarly, we used the *Phase where faults were introduced (baseline)* results, combined with the finer grain breakdown of *Major coding faults* to report the results from [25].

The results of this effort are given in Table 4. The first row provides an identification number for the study, the reference, and the year of publication, while the second row describes the system(s) studied. The third row lists the size and the type of the data sample.[3] The remaining rows show the percentage of the sample which is associated with one of the following categories:

- *Requirements/Design faults* include incorrect, changed, or missing requirements and design faults (i.e., faults that result in changing a design artifact, including architecture, high-level, and low-level component design). In [21], the ODC defect-type Function and Algorithm were grouped together and related to requirements and design faults [22].
- *Coding faults* are directly related to the code and may include faults in component implementation, incorrect or missing implementation that is not due to requirements issue(s) and can be fixed without design change(s), values assigned incorrectly or not assigned at all (e.g., ODC assignment/inititalization type), missing or incorrect validation of data or incorrect loop or conditional statements (e.g., ODC checking type).
- *Interface/Integration faults* are traced to interaction and/or integration of components
- *Procedure/Process problems* are caused by noncompliance to procedures, missing procedures, or are related to the process used.
- *Other* category consists of any additional categories given in the related papers that could not be traced to any of the previously listed categories.
- *Unknown* category contains the percentage of the sample points for which data related to the type of fault have not been available in the original study.

[3] The sample size does not reflect the quality of the software. Rather, it depends on the time period spanning the data collection process, the size of the software, the types and criticality of problems included in the analysis, etc. For example, analysis presented in [21] considered 199 postlaunch safety critical anomalies, while in this study, we consider 2,858 software change requests entered throughout development, testing, and operation of 21 CSCIs, including all levels of criticality.

When discussing the results shown in Table 4, we use the term problem to refer collectively to the faults (defects), failures, and anomalies studied. The main observations can be summarized as follows:

- **The percentage of coding faults is significant across all studies. Even more, it is rather close to the percentage of requirements and design faults together.** Specifically, across the four studies, at least 16-41 percent of problems were due to coding faults,[4] while 17-53 percent were due to requirements and design faults. The lower total percentage of requirements, design, and coding faults (32.7 percent) in study I is due to the fact that this study analyzed safety critical postlaunch anomalies, with a large percentage of procedure and process faults (29.2 percent) and large percentage of anomalies for which nothing was fixed (13.6 percent included in the Other category in Table 4).
- **The interactions between components cause problems.** Studies I and II trace around 7 percent of problems to integration and interface issues. In the case of our study IV, 16 percent are due to approximately 2 percent of failures which were explicitly tied to integration issues and almost 14 percent of failures caused by data problems (i.e., problems with the instruction and data repository used by multiple components). Associating data problems with interface/integration was done based on recommendation from the NASA personnel.
- **The remaining categories usually are not major causes of problems and may be domain specific.** For remaining categories, some studies do not report problems. Possible reasons may include: No such problem has been observed, the category was not considered at all, or it is not plausible in the specific domain of the study. Anyway, for all studies, the total percentage of problems associated with these categories is usually small. It is worth mentioning that only studies I and IV reported problems related to *Procedure/Process* category. Although both studies analyze data from NASA projects, the percentage of procedural problems is significantly higher in case of study I. The difference may be due to the fact that study I focuses on safety-critical postlaunch anomalies, while study IV includes all failures observed pre and postrelease, regardless of the criticality level. Perhaps, the difference in the procedure/process problems observed between studies I and IV can provide insight into safety-critical postlaunch anomalies and/or the specifics of the missions studied. Finally, out of 27.1 percent given in the *Other* category for study I, 6 percent were related to timing issues, while 13.6 percent were reported in the *Nothing fixed* category due to variety of reasons, including "false positive" anomalies when software behaved correctly, but the personnel was surprised by its behavior. The 14 percent in the *Other* category for study III are due to testing, load balancing, and not applicable categories.

Although the categories used to compare our work (study IV) with the related work (studies I-III) are broad, we believe that the results are still quite meaningful. Thus, our comparisons show that *the percentage of problems reported due to coding, interface, and integration faults together is approximately the same or even higher than percentage of faults due to early life cycle activities (i.e., requirements and design).*

In [8], the authors actually concluded that "many of the faults that remain in the software after deployment are simple programming errors." As mentioned in Section 2, [7] (considered 408 software faults during field operation of an IBM operating system) and [8] (analyzed 668 prerelease and postrelease software faults from 12 open-source projects) had very similar fault distributions over the main ODC fault types, thus providing even stronger support for our observations. Specifically, in addition to 34.3 and 46.4 percent of faults that certainly are classified as coding faults (i.e., ODC fault-type Assignment and Checking), the vast majority of the faults in the ODC Algorithm type, with 40.1 and 37.7 percent in [7] and [8], respectively, can be related to coding errors based on the details given in these studies. However, several statements given in [7] and [8] indicate that these two papers did not fully consider requirements and design faults, which did not affect their goal—identification of representative software faults for the purpose of fault injection at source code level. We decided not to include [7] and [8] in Table 4 since their results most likely present somewhat skewed view in the context of identification of the major types of software faults.

In summary, the results presented in this section support the idea that a significant portion of software failures is due to faults entered during implementation and integration, which is contrary to the common belief that the majority of faults are entered during requirements, specification, and design activities [2], [4], [9]. The consistency of this observation across multiple projects which span various domains and differ in many aspects (i.e., development methods, coding language, size, development organization, etc.) suggests that this trend most likely is a characteristic of faults and failures themselves and not of the projects studied. Additionally, this consistency provides strong support for the external validity of our study.

5 Threats to Validity

As with all studies, there are limitations and threats to validity that need to be discussed. We consider both the internal and external threats to validity, for both GCC and NASA mission case studies.

5.1 Internal Validity

The threats to internal validity of our study can be split into two groups: 1) threats pertaining to the data available and 2) threats pertaining to the methods used and analysis conducted. In any case when we were unsure of the data available, the meaning of the data, our methods, and/or the meaning of our results, we turned to the developers for additional insights.

[4] The percentages of coding faults in study I may be higher than 16 percent if part of the ODC Algorithm-type faults may have been due to coding errors which do not require changes in the requirements or design.

All of our analysis for each case study is limited to the subset of failures for which the appropriate data were available and may not be representative of the entire data set. As with any case study, we have no control over any bias that may exist in the data due to human subjectivity and/or human errors. The first step in both studies was to identify failures to study. For GCC, we were able to run the regression test suite and observe failures, while for the NASA mission, we assumed that the SCRs entered due to nonconformance with requirements represented failures. This assumption was based on detailed discussions with experienced personnel from the NASA mission. In the case of the NASA mission, we observed that CSCIs with more files tend to have more nonconformance SCRs associated with them. However, a finer grained size metric such as LOC, which was not available to us, would probably allow for more accurate comparison.

When considering our analysis pertaining to fault localization, for GCC out of 111 observed failures, we were able to map 85 failures to the faults that caused these failures. We took special care to exclude duplicates, as well as failures that occurred because some functionality was unavailable in the version used for testing. For the NASA mission, based on the data available to us, only 404 (i.e., 356 from NASA_set_1 and 48 from NASA_set_2) of the 2,858 nonconformance SCRs could be associated with the corresponding faults. Although we observed consistency across the two NASA data sets and at multiple levels of abstraction, one must wonder what the unresolved failures would reveal. However, it should be noted that, compared to related studies, 404 is still a significant number of failures to study.

It is possible that the methods used to map failures to faults could be the reason why only a subset of failures was mapped to faults. However, for GCC, the methods we used to map failures to the changes made to prevent the failures from reoccurring were validated by the developers. Additionally, for the NASA mission, our queries were developed with help from the change tracking system team lead and were well tested. The queries accurately map failures with all changes at the file level, as long as the changes were recorded in the data available to us.

When considering the source of the failure (i.e., fault types), we chose not to include GCC in the analysis because assigning a source to each failure would have been a manual process conducted by the authors of this paper who lack domain knowledge and, thus, would be prone to errors. For the NASA mission, there was a specific field in place with a list of predefined sources whose purpose is to capture the cause of the failure. Although the choice of the source may still be subjective and the field is not mandatory, in this case, the analyst making the choice has specific domain knowledge pertaining to the mission and often to the specific CSCIs.

Additionally, when studying the source of failures, we looked at the cause of failures for all 21 CSCIs as one group and then repeated the analysis for subsets of CSCIs grouped by the number of releases. The fact that our results are consistent, that is, the same top three sources of failures identified for the entire data set are the top three sources of failures (with rather close distributions) for each group, supports the internal validity of the study. However, repeating the analysis based on different grouping may expose additional characteristics.

5.2 External Validity

The external validity of both fault localization and source of the failures (i.e., distribution of types of faults) analysis is shown by the consistency of the observations across the case studies considered in this paper and by comparing with the qualitative and quantitative results from the related work. The fault localization analysis shows strong consistency between open-source and NASA projects, thus allowing us to quantify and statistically test this phenomenon for the first time.

Due to the lack of detailed fault and failure data from GCC, we could not explore the consistency of the source of failures between the open-source GCC project and the NASA mission. However, the fact that we observed consistencies across CSCIs developed by different groups, at different locations, provides some support for the external validity. Additionally, by defining broad categories which cover elements of the finer grained, but different classification schemes used in other related studies, we were able to show a reasonable consistency of a few main observations of our study with several recent related studies which include data from open-source software, large industrial proprietary software, and NASA missions software. More detailed and more rigorous comparison certainly would have been beneficial for advancing the knowledge in the area, but it was not possible due to the fact that different studies are using different classification systems, with some overlapping categories.

6 Summary of the Main Results and Concluding Remarks

Detailed studies on empirical fault and failure data have been rare in the past. Conducting such studies usually is difficult because data are not readily available, often are inconsistent, incomplete, or lacking altogether. In this paper, we have shown that change tracking systems, although they have not been designed and used for this purpose, provide a wealth of information about faults and failures. Using the data extracted from a large open-source application and a very large NASA mission, we addressed two main research questions: 1) *Are faults that cause individual failures localized?* and 2) *Are some sources of failures (i.e., types of faults) more common than others?*. Table 5 summarizes the observations made in this paper with respect to each research question. It should be noted that, whenever possible, we used both case studies to explore the phenomena empirically. When necessary data were not available, we used evidence from related work to show that observed trends apply to multiple studies.

Although a few related studies indicated that failures may often be caused by nonlocalized faults, to the best of our knowledge, this phenomenon has not been quantified or statistically tested before. Proving that a significant percentage of failures in large, complex software products are caused by nonlocalized faults has large software

TABLE 5
Summary of the Main Results

Observations related to each research question	Case Study Evidence	Section
Exploring the localization of faults related to individual failures		
The number of failures associated with faults that spread across multiple files is consistent across the studies. % of failures that map to fixes in two or more files % of failures that map to fixes in three or more files	GCC, NASA_set_1, NASA_set_2 58%, 62%, 60% 31%, 44% ,33%	Section IV-A
A significant number of failures are associated with faults that spread across multiple components. % of failures that map to fixes in two or more components	GCC, NASA_set_1 33%, 6-36%	Section IV-A
The number of failures associated with faults that spread across multiple CSCIs is consistent across two independent NASA data sources. % of failures that map to fixes in two or more CSCIs % of failures that map to fixes in three or more CSCIs	NASA_set_1, NASA_set_2 12%, 19%, 5%, 4%	Section IV-A
The majority of faults are contained in a small percentage of files.	GCC 80% of faults are located in 20% of files	Section IV-A
Exploring common sources of failures		
The number of SCRs entered against a CSCI tends to increase with the number of files in the CSCI.	NASA_* (includes NASA_set_1, NASA_set_2, and additional SCRs)	Section IV-B
Coding faults, requirement faults, and data problems are the three most common sources of failures. % requirements faults % coding faults % data problems	NASA_* 33% 33% 14%	Section IV-B
The most common sources of failures are consistent across subsets of CSCIs grouped by releases.	NASA_*	Section IV-C
The percentage of coding, interface, and integration fault types is close to or even exceeds the percentage of requirements and design faults.	NASA_*, [7], [8], [19], [21], [25]	Section IV-D

engineering implications because these types of failures are usually harder to prevent, as it is likely more difficult to find and fix the faults that caused them. According to [17], many software failures in space missions were caused by the unanticipated combination of simple faults. Learning more about the combination of these simple faults and how and when they cause failures will help prevent failures of this nature in the future. Our analysis pertaining to the number and location of GCC faults confirmed the common belief and the observations made on a few other case studies—the majority of faults are contained in a small portion of the system. Unfortunately, due to the limited number of failures that had the corresponding changes recorded at a file level in the data available to us, we could not explore whether this phenomenon holds true for the NASA data.

On the other hand, for the NASA mission, the sources of failures (i.e., the type of faults) were identified for over 2,800 software change request related to 21 CSCIs with over 8,000 files. To the best of our knowledge, this is the largest software change requests (i.e., problem reports, bug reports) data set analyzed in the published literature. Our analysis of the whole set of 21 CSCIs showed that requirement faults, coding faults, and data problems are the three most common sources of failures. Interestingly, the same observation, with reasonably consistent percentages, was made on each of the subsets consisting of CSCIs that have undergone the same number of releases. In addition, to the largest possible extent, we compared our findings related to the distribution of fault types with several recent empirical studies. Specifically, we showed that, for multiple software systems, the percentage of the coding, interface, and integration faults is close to or even exceeds the percentage of requirements and design faults. Hence, at the current state of the practice of the software development, fault prevention and elimination methods focused on requirements and design activities are useful, but will not eliminate all problems.

Through detailed analysis of empirical data for each of our case studies and comparisons to related recent studies wherever possible, we have taken a step toward revealing the complex relationships between faults and failures. The consistency of several main observations across multiple case studies from various research efforts suggests that the observed trends are likely intrinsic characteristics of software faults and failures rather than project specific. Obviously, standardization of the classification schemes and the way data are stored would facilitate more detailed comparisons and provide additional insights.

The work presented in this paper suggests that there is a lot to learn from conducting detailed and rigorous analysis of software faults and failures. It also shows that some of the popular software engineering beliefs, which seem to be based on older, much smaller case studies, do not hold on current, large-scale software systems. With ever-changing software development technologies and tremendous increase of software size and complexity, it is imperative to revisit some of these beliefs and discover new phenomena. We believe that exposing the characteristics of faults, failures, and the fault-failure relationships is quite beneficial. Future studies that would further explore similar research goals in time can establish empirical bases of the software engineering discipline and benefit both software research and practitioners communities.

ACKNOWLEDGMENTS

The authors would like to thank the developers of both case studies for the vast amount of help they provided. They especially thank the NASA personnel for their invaluable support: Pete Cerna, Randolph Copeland, Susan Creasy, James Dalton, Nick Guerra, John Hinkle, Lisa Montgomery, Chad Pokryzwa, David Pruett, and Timothy Plew. They also thank Robyn Lutz for her feedback on the draft version of this paper. The anonymous *TSE* referees and the associate editor provided valuable comments which have certainly improved the paper. This work was supported in part by NASA OSMA SARP under a grant managed through NASA IV&V Facility in Fairmont, West Virginia, and by the US National Science Foundation under CAREER grant CNS-0447715.

REFERENCES

[1] C. Andersson and P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems," *IEEE Trans. Software Eng.*, vol. 33, no. 5, pp. 273-286, May 2007.

[2] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, vol. 27, no. 1, pp. 41-52, 1984.

[3] V.R. Basili, "The Role of Experimentation in Software Engineering: Past, Current, and Future," *Proc. 18th Int'l Conf. Software Eng.*, pp. 442-449, 1996.

[4] B.W. Boehm, R.K. McClean, and D.B. Urfrig, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software," *IEEE Trans. Software Eng.*, vol. 1, no. 1, pp. 125-133, Mar. 1975.

[5] B. Boehm and V.R. Basili, "Software Defect Reduction Top 10 List," *Computer*, vol. 34, no. 1, pp. 135-137, Jan. 2001.

[6] R. Chillarege, I. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 943-956, Nov. 1992.

[7] J. Christmansson and R. Chillarege, "Generation of an Error Set That Emulates Software Faults Based on Field Data," *Proc. 26th Int'l Symp. Fault-Tolerant Computing*, pp. 304-313, June 1996.

[8] J.A. Duraes and H.S. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Trans. Software Eng.*, vol. 32, no. 11, pp. 849-867, Nov. 2006.

[9] A. Endres, "An Analysis of Errors and Their Causes in System Programs," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 140-149, June 1975.

[10] A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories.* Pearson/Addison-Wesley, 2003.

[11] N.E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.*, vol. 25, no. 5, pp. 675-689, Sept./Oct. 1999.

[12] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797-814, Aug. 2000.

[13] K. Goševa-Popstojanova and K.S. Trivedi, "Architecture Based Approach to Quantitative Assessment of Software Systems," *Performance Evaluation*, vol. 45, nos. 2/3, pp. 179-204, June 2001.

[14] K. Goševa-Popstojanova, M. Hamill, and R. Perugupalli, "Large Empirical Case Study of Architecture-Based Software Reliability," *Proc. 16th IEEE Int'l Symp. Software Reliability Eng.*, pp. 43-52, Nov. 2005.

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

[16] W.S. Greenwell and J.C. Knight, "What Should Aviation Safety Incidents Teach Us?" *Proc. 22nd Int'l Conf. Computer Safety, Reliability and Security*, Sept. 2003.

[17] G. Holzmann, "Conquering Complexity," *Computer*, vol. 40, no. 12, pp. 111-113, Dec. 2007.

[18] D. Jackson, M. Thomas, and L. Millett, *Software for Dependable Systems: Sufficient Evidence?* Nat'l Academies Press, 2007.

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

[20] N. Leveson, *Safeware System Safety and Computers.* Addison-Wesley, 1995.

[21] R.R. Lutz and I.C. Mikulski, "Empirical Analysis of Safety Critical Anomalies during Operation," *IEEE Trans. Software Eng.*, vol. 30, no. 3, pp. 172-180, Mar. 2004.

[22] R.R. Lutz and I.C. Mikulski, "Ongoing Requirements Discovery in High Integrity Systems," *IEEE Software*, vol. 21, no. 2, pp. 19-25, Mar./Apr. 2004.

[23] T.J. Ostrand and E.J. Weyuker, "The Distribution of Faults in a Large Industrial Software System," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 55-64, 2002.

[24] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Where the Bugs Are," *Proc. ACM Int'l Symp. Software Testing and Analysis*, 2004.

[25] W.D. Yu, "A Software Fault Prevention Approach in Coding and Root Cause Analysis," *Bell Labs Technical J.*, vol. 3, no. 2, pp. 3-21, Apr.-June 1998.

[26] www.computer.org/portal/pages/seportal/subpages/sedefinitions.html, 2009.

Maggie Hamill received the BS degree in computer science from James Madison University in 2003 and the MS degree in computer science from West Virginia University in 2006. She is currently working toward the PhD degree at West Virginia University under Dr. Katerina Goševa-Popstojanova. She works at the NASA IV&V facility in Fairmont, West Virginia, in support of her research. Her interests are in empirical software engineering and software reliability. She is a member of the IEEE.

Katerina Goševa-Popstojanova is a Robert C. Byrd associate professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University, Morgantown. Her research interests are in reliability, availability, performance assessment of software and computer systems, and computer security and survivability. She received the US National Science Foundation (NSF) CAREER Award in 2005. She has served as a principal investigator on various NSF, NASA, WVU Research Corporation, NASA West Virginia Space Grant Consortium, CACC, and Motorola funded projects. She served as the program chair of the 18th International Symposium on Software Reliability Engineering and is currently serving as a guest editor of the special issue of the *IEEE Transactions on Software Engineering* on evaluation and improvement of software dependability. She served or is currently serving on program and organizing committees of numerous international conferences and workshops. She is a senior member of the IEEE and a member of the ACM.