TECHNIQUES FOR IMPROVING SOFTWARE DEVELOPMENT PROCESSES BY MINING SOFTWARE REPOSITORIES

by

Tejinder Dhaliwal

A thesis submitted to the Department of Electrical and Computer Engineering

In conformity with the requirements for

the degree of Master of Applied Science

Queen's University

Kingston, Ontario, Canada

(September, 2012)

Abstract

Software repositories such as source code repositories and bug repositories record information about the software development process. By analyzing the rich data available in software repositories, we can uncover interesting information. This information can be leveraged to guide software developers, or to automate software development activities. In this thesis we investigate two activities of the development process: *selective code integration* and *grouping of field crash-reports*, and use the information available in software repositories to improve each of the two activities.

Selective code integration is a process which allows developers to select a particular feature from an existing component and to integrate into another software system. Selective code integration is particularly useful when a software system is implemented by integrating reusable components. To integrate a feature selectively, it is essential to know what part of the source code implements the feature. However, if this information is not available precisely, selective integration may fail, which increases software development time.

Crash reports collected from end user's environment are useful to uncover the bugs that were not found during software validation. However, the number of crash-reports collected from end users is often too large to handle. To reduce the amount of data for the analysis, similar crash-reports are grouped together. Nevertheless, if the crash reports are not well organized, it significantly increases the fix time for bugs.

In this thesis, we propose solutions to counter the above stated two problems by leveraging the information available in software repositories. First we propose an approach to identify the source code related to a feature, which avoids failures in selective code integration. Second we propose an approach to group field crash reports correctly, which reduces the bug fix time. For each of our approach we present a case study and measure the benefits of the respective approach.

Co-Authorship

The case study in Chapter 3 of the thesis is based on a paper [1] co-authored with my supervisor, Dr. Ying Zou, Dr. Ahmed E. Hassan from School of Computing, and Dr. Foutse Khomh, a post-doctoral fellow in our lab. The case study in Chapter 4 of this thesis is based on a published paper [2] co-authored with Dr. Ying Zou and Dr. Foutse Khomh. I was the primary author of both the papers.

In the two studies described in this thesis, Dr. Ying Zou and Dr. Ahmed E. Hassan supervised all of the research related to the work. Dr. Foutse Khomh also participated in the discussions related to both studies, and provided feedback and suggestions to improve the work.

[1] T. Dhaliwal, F. Khomh, Y. Zou, and A. E. Hassan “Recovering Commit Dependencies for Selective Code Integration in Software Product Lines," accepted for publication in *Proc. 28th IEEE International Conference on Software Maintenance (ICSM), 2012*.

[2] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM)*, Williamsburg, VA, USA, pp. 333-342, 2011.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Ying Zou for her wisdom, guidance and encouragement that she provided while supervising my thesis research.

Secondly, I am thankful to all the members of the Software Reengineering Research Group and the Software Analysis and Intelligence Lab. In particular, I would like to acknowledge Dr. Foutse Khomh, Dr. Ahmed E. Hassan, Dr. Bram Adams, and Bipin Upadhyaya for their advice and assistance in producing this thesis.

I am grateful to the Indy team from Research in Motion Ltd. for their support and cooperation for the case study in chapter 3. I am also grateful to Mr. Chris Hofmann from Mozilla foundation for his valuable suggestions and support for the case study in chapter 4.

I would like to thank my committee members Dr. Alireza Bakhshai, Dr. Bob Tennent, Dr. Thomas R. Dean, and Dr. Ying Zou for their valuable feedback on this thesis. Finally, I am grateful to my friends and family for all their encouragement and support throughout my career.

Table of Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

The Software development process translates user's needs into a software product [1]. The process involves software planning, software implementation, software validation and software maintenance. In a nutshell, software planning articulates user's needs into requirements; software implementation realizes the specified requirements; software validation checks whether the software satisfies the specified requirements; and software maintenance is the post release improvements made to the software.

Software repositories such as source code repositories, bug repositories and other archives, record information about the development process. By analyzing and cross-linking the information available in these repositories, we can capture useful information about the development process [2] . For example, a system can identify non-structural dependencies among source code files, by mining the association among these files in the history of a source code repository. In another example a system can identify duplicate bugs, by learning the association patterns between prior duplicate bugs in the history of a bug repository.

In this research we mine software repositories, and leverage the information to assist software developers in two development activities: (a) *selective code integration* and (b) *grouping of field crash-reports*. Both of the activities are tedious in nature and require manual effort from developers. We attempt to learn developers' actions from the information logged in the repositories, and propose an approach to assist them in each of the two activities. Additionally for each of our approaches we present a case study. The first case study on an enterprise software system shows that our approach to improve selective code integration can reduce integration

failures by up to 94%. The second case study on an open-source software system shows that our approach to group the field crash-reports can reduce bug fix time by more than 5%.

In this chapter we provide detailed introduction of the two development activities (i.e., *selective code integration* and *grouping of field crash-reports*) and form the research statement for this thesis.

1.1 Background

1.1.1 Selective Code Integration in Software Product Lines

The following concepts are helpful in developing a fundamental understanding of source code integration:

1. ***Version Control System:*** A version control system (VCS) maintains source code of a software system in a well-organized manner [3]. A code change submitted to a VCS is called as a *commit*. Modern VCS use a change oriented model to manage the source code [4]. In the change oriented model, the source code is maintained as an ordered set of commits. VCS also tracks other information for each commit, for example, the developer information, a description and the submission time of a commit.
2. ***Branching:*** A branch is an instance of source code, forked from another branch. VCS supports the branching of source code to maintain separate instances of the code. For example, a separate branch of source code is maintained for each version. The organization of source code branches is called as the branching model, and the main branch is called as the trunk.
3. ***Functional Change:*** A functional change is a new feature or a bug fix added to a software system. A functional change includes one or more commits submitted by one or more developers. A request for a functional change is called as a *change request* (CR),

and tracked in a CR Tracking System. Often the commits in a VCS are linked to the corresponding CR, and these links help the developers to map a functional change to its implementation.

In software system with multiple branches, often a functional change made into one branch needs to be propagated into another branch. The propagation of a functional change, from one code branch into another, is called the integration of the change.

Software Product Line: Software vendor offers different variants of a software system to support a diverse user base. For example Microsoft Windows[1] support six variants of Windows 7 operating system. Each variant of a software system is a separate product and all variants together form a software product family. The software products in a family are largely similar, and are often implemented by integrating reusable components from a shared set. Such a development model, where similar software products are implemented by reusing common components, is called as a Software Product Line [5]. Software Product Lines decrease the implementation cost and improve the maintainability of the software products [6].

Software product lines generally use a main line branching model [7] . In the main line branching model, the main branch (i.e., trunk) maintains the code common to all the products in the family, and each product has a separate branch (i.e., product branch) to maintain the product specific code. New functional changes for the common part of the source code are implemented in development branches and merged into the trunk. Ideally the trunk should contain only the code common to all products, and the product specific code should be added only into the product branches. However, product specific changes are often added to the common part of the code [8]. In such scenarios, the changes need to be selectively integrated into the product branches.

[1] http://windows.microsoft.com

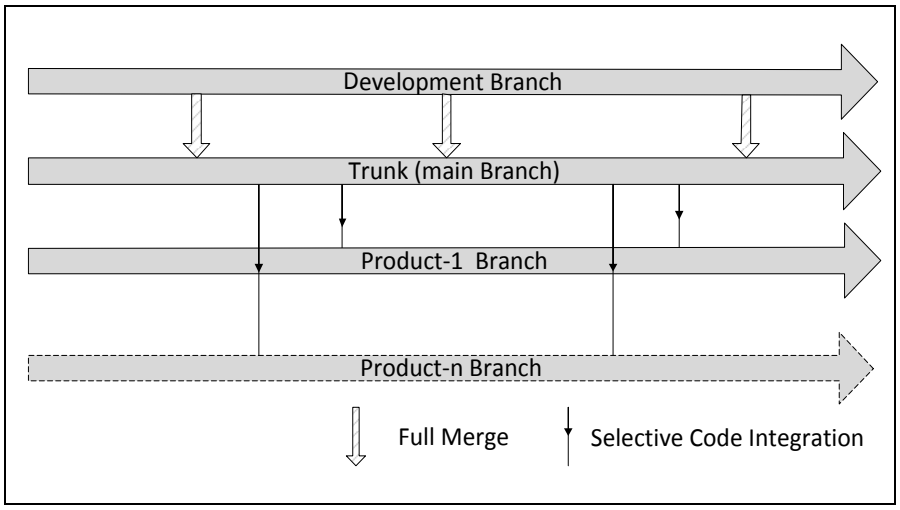

Figure 1-1: Branching Model for a Software Product Line

Selective Code Integration: Selective code integration is a process where a functional change is selectively propagated from one branch into another. In contrast to ***full merge***, where all the commits made into a branch are propagated into another, only the commits implementing a particular functional change are propagated. Figure 1-1 shows the main line branching model for a software product line, and the type of integration between different branches of the product line.

1.1.2 Grouping of Field Crash Reports

Software validation helps to uncover the bugs in a software system. However, because of the unpredictable user behavior and the unforeseen usage patterns, not all the bugs can be uncovered before the software is released to the user [9]. The bugs undetected in software validation can cause a software system to crash in the field (i.e., unexpected termination in a user environment). Many software systems support a built-in automatic crash reporting tool. When a field crash occurs, the tool collects a crash-report from user environment and sends it to the software vendor.

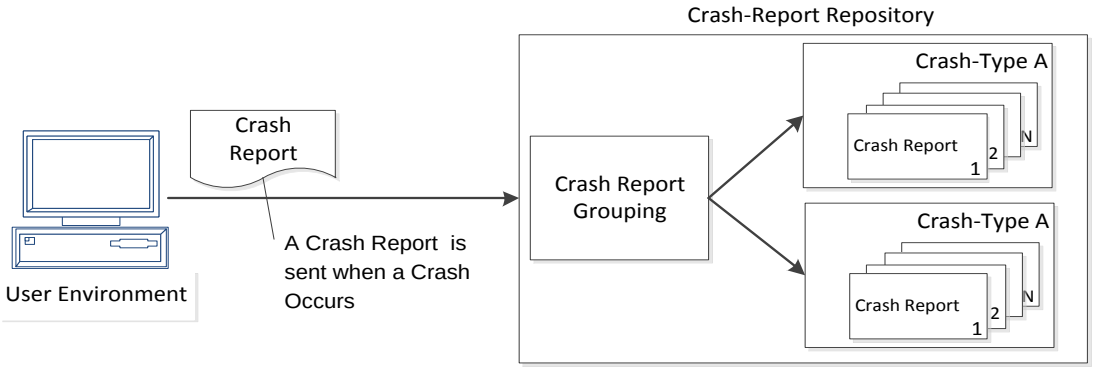

Figure 1-2: Overview of Field Crash Reporting

A crash-report usually contains the stack trace of the failing thread and other runtime information. A stack trace is an ordered set of frames, where each frame refers to a method signature. The crash-reports can help the software developers to diagnose and fix the root cause of the crashes. The automatic collection of crash-reports in Mozilla Firefox[2] improved the reliability of Mozilla Firefox by 40% from November 2009 to March 2010 [10]. Microsoft was able to fix 29% of all Windows XP bugs due to this automatic collection and analysis of crash-reports [11]. Figure 1-2 gives an overview of the field crash reporting process.

The built-in crash reporting tools often collect a large number of crash-reports. For example, Firefox receives 2.5 million crash-reports every day [10]. However, due to reoccurrence of the same bugs, many of the crash-reports are redundant. To reduce the number of crash-reports to handle, similar crash-reports are identified and grouped together as ***crash-types***. By organizing the crash reports as crash-types, developers can identify the crash-types causing majority of field crashes. Thus, the grouping of crash-reports helps developers to organize the crash-reports and prioritize the crash-types.

[2] http://www.mozilla.org/firefox

1.2 Problems

Selective code integration and grouping of field crash-reports are two important activities in the software development process. In this section we explain the existing problems in each of the two activities.

1.2.1 Identifying Commit Dependencies in Selective Code Integration

In selective code integration developers need to select and integrate the commits implementing a functional change. When implementing a functional change, developers links the commits with the corresponding change request, and these links helps to facilitate the selective code integration. However, the commits implementing a functional change are dependent on other commits existing in the branch, and these required dependencies might be missed in selective code integration.

It is a challenging task to identify the commit dependencies, since the dependencies among the commits are not always explicit. The following scenarios should be considered carefully by development teams, when integrating a functional change selectively:

1. A functional change is implemented by multiple commits. Each commit needs to be linked with the change, so that the change can be integrated as a whole. If any of the commits is not linked with the change, the integration will fail.
2. The implementation of a functional change C_1 modifies a function and the implementation of another functional change C_2 has a dependency on the modified function. When integrating the change C_2, the commits implementing the change C_1 should also be integrated.

When commit dependencies are missed during the selective code integration, the integration fails. This failure is characterized by either a compilation or a runtime error. If the integration

fails, a developer needs to debug and add the missing dependencies. Our observation shows that if the integration of a functional change fails in the first attempt, the total integration time of the functional change increases by 530%

1.2.2 Identifying similar Crash-reports in Crash-Report Grouping

The grouping of field crash-reports helps developers to sort a large volume of crash reports, but the existing grouping approaches have limitations in identifying the similar crash reports. For example, Mozilla groups the crash-reports by using the top method signature in the failing stack trace (i.e., the last invoked method in the failing thread) [12]. However, such a grouping approach is not accurate in the following cases:

1. ***Similar Crash-Reports triggered by multiple bugs:*** If multiple bugs have the same top method signature in their failing stack traces, the crash-reports triggered by these bugs are designated to the same crash-type.
2. ***Non identical Crash-Reports triggered by single bug:*** Since the crash-reports triggered by the same bug might not be identical, the crash-reports triggered by the same bug are designated to different crash-types.

When crash-reports caused by multiple bugs are grouped together, the developer can miss a few of the bugs as the developers cannot inspect each crash-report in the group. We observe that if the crash-reports caused by multiple bugs are grouped together it takes on average 65% more time to fix the bugs, in contrast to when the crash-reports caused by each bug are grouped separately. The observation indicates that if each crash-type contains the crash-reports triggered by only one bug, it is easier for a developer to fix the bugs.

1.3 Research Statement

The two problems we discussed in this chapter, increase the manual effort required in the software development process, and in turn the development time of a software system is increased. The missing dependencies in selective code integration increase the integration time of a functional change, and an incorrect grouping of crash-reports increases the bug fix time.

In both of the activities when an error occurs, the developers need to diagnose a large amount of data and fix the error. However, the errors occurred and the corrective actions taken by the developers are tracked in software repositories. For example, if selective integration of a functional change fails, a developer debugs the source code and links the missing dependencies. A careful inspection of historical data can help to extract the association patterns among the dependent commits. Similarly a close inspection of crash-types, can uncover the characteristics of correctly grouped crash-types. The patterns learnt from the historical data, can be translated into algorithms. These algorithms are can replace the developers' actions, or can assist the developers to perform their actions more efficiently.

The goal of this research is to develop automated techniques to assist the developers, within the scope of the two development activities we discussed. The research demonstrates applications of Mining Software Repositories to improve software development process. Our study provides the following solutions for each of the two challenges:

- **An automated approach to identify commits dependencies for selective code integration**.

  In large software systems, developers generally have a limited knowledge of the complete system [13] and usually the developer responsible for the integration of a functional change is not the one who implemented the change. Therefore, it is important

to assist developers in determining commit dependencies. We propose an approach to identify the commit dependencies and group the dependent commits together, that should be integrated together. Our approach determines dependencies among the commits by analyzing structural and logical dependencies among source code elements, and the history of developers' working collaborations.

- **An automated technique to identify similar field crash-reports.**

  Existing techniques [10] group the crash reports based on a comparison of the stack traces, but the results are not always correct. Another technique [11] groups the crash reports based on a large number of grouping heuristics created by developers over the time. However, such grouping heuristics are not always available. A correct grouping of crash-reports can save the developers' time to fix the bugs. Therefore it is important to assist the developers to group the crash reports. To ensure that the crash-reports triggered by different bugs are grouped separately, we propose a two level grouping approach, where a crash-type is further divided based on a detailed comparison of the stack traces in the crash-reports. We capture the threshold values of stack traces similarity from the correctly grouped crash reports in history, and strive for the same level of stack traces similarity within a subgroup. The crash-reports are large in number, and a detailed comparison of stack traces could be an intensive computation, therefore we also optimize our approach to handle a large number of crash-reports.

1.4 Organization

The rest of this thesis is organized as follows:

- Chapter 2 outlines related work in the areas of selective code integration and identification of similar crash-reports.

- Chapter 3 presents our approach to identify commit dependencies in selective code integration, with a case study.
- Chapter 4 presents our approach to identify similar crash-reports in field crash reporting, with a case study.
- Chapter 5 summarizes and concludes the thesis and discusses future work.

Chapter 2

Literature Review

In this chapter we discuss previous work in the areas of the selective code integration in software product lines, identifying the commit dependencies, and grouping of field crash-reports. Additionally, we give an overview of the test and modeling techniques used in the case studies in Chapter 3 and Chapter 4.

2.1 Selective Code Integration

2.1.1 Integration Approaches in Software Product Lines

Software product lines exploit the commonality of software products by reusing the source code and adding features selectively to each product variant [14]. Researchers have suggested different approaches to reuse the software components.

Kang *et al.*[15] suggest a feature oriented model for software product lines. They explain how the common features can be developed together and selectively integrated into software products of a family, using the feature oriented development approach.

Capretz *et al.*[16] provide COTS (Components off the shelf) model for software product lines. In COTS model they suggest to develop independent software pieces that can be used as plug and play components, and the software products can be assembled by reusing these components. In their research they show that the plug and play components will not always exactly match the requirement of a software product, therefore the components require adaptations to integrate in a software product.

Groher *et al.*[17] provide an aspect-oriented and model-driven approach for software product lines. Their approach suggests developing core components for a domain, and a software

product for that domain can be created by automatic assembling of the core components. Their research shows that if the features are not well separated in the core components, the developers needs to manually assemble the core components in software products.

Though each model explained by the researchers can be distinguished from the others, but all of these models agree that in the software product lines developers need to manually select features from a shared pool of features and integrate into software products.

2.1.2 Risk of Integration Failure in Selective Code Integration

At the code level a feature is a functional change, and integrating a feature selectively implies selecting the commits related to the feature from one code branch and integrating into another branch. The modern version control system acknowledges the need of selective code integration and provides built-in support for it. For example GIT[3] provides the command *cherry-pick*, Mercurial[4] supports the command *graft* and Perforce[5] support the command *selective integrate*. These commands completely rely on developer knowledge about the completeness of a functional change. A developer needs to manually select all the commits related to a functional change, to integrate the change.

Selective code integration is applicable to any development scenario, where a functional change needs to be integrated selectively. Walrad *et al.*[18] explain that selective code integration is also applicable in parallel development of multiple versions, and in post release maintenance of multiple versions. Their paper also acknowledges the risk of integration failure in selective code integration.

---

[3] http://inux.die.net/man/1/git-cherry-pick
[4] http://www.selenic.com/mercurial/hg.1.html#graft
[5] http://www.perforce.com/

Sullivan [19] explains the need and the risk of selective code integration in detail. He explains that because each commit has a set of dependencies and all dependent commits need to be integrated as a whole, the selective code integration is a vulnerable process. The existing literature relies on the developers manual efforts to mitigate the risk of integration failure, whereas we propose an automated approach to assist the developers in selective code integration.

2.1.3 Recovering Commit Dependencies by Mining Version Repositories

The problem of recovering commit dependencies falls into the category of identifying source code dependencies. Previous research by Lindvall *et al.*[13] has identified that in large software systems developers generally have a limited knowledge of the complete system, and the logical dependencies in the source code are not always explicit [20]. Therefore we propose an approach to mine the version control history and identify the dependencies among the commits.

Gall *et al.*[20] were the first to use version control histories to identify the dependencies among source code modules. They look into the change patterns among source code modules and reveal the hidden dependencies among the modules. In comparison to their work, we identify the dependencies among the fine grained commits instead of the large source code modules.

Zimmerman *et al.*[21] developed a tool called ROSE that uses association rules on the history of the code changes submitted by the developers. They propose an impact analysis technique to predict the potential source code changes related to a given change, using the version histories. Annie Ying [22] developed a similar approach to identify source code dependencies that uses association rules mining on version control system. She also evaluated the usefulness of the results, considering a recommendation to be more valuable or “surprising” if it could not be determined by program analysis. In contrast, we find the dependencies among the code changes

already added by the developers. We apply similar association rules, but our focus is to determine the correlation among existing code changes, instead of predicting possible code changes.

Apart from mining the version histories, researchers used the alternative sources to determine the dependencies among the code changes. Chen *et al.*[23] uses the textual similarity of the log messages of a version control system and Atkins [24] used the textual similarity of the source code modules to identify the dependencies. HIPKAT [25] is a tool that looks into a large varied amount of information to help the developers in analyzing the impact of a code change. HIPKAT uses the source code files, the mail archives and the online documentation and helps the developers to investigate the impact of a code change.

Several other techniques [26], [27], [28] have been proposed by the researchers to slice the large software systems into modules using the history of source code fragments modified together. These techniques are helpful in impact analysis of a code change, and help to identify the dependencies at a granularity level of the software module, but none of these techniques can be applied to the find the dependencies at a granularity level of the commits.

2.2 Field Crash Report Grouping

In recent years, many large-scale software systems have started collecting the field crash reports and researchers have analyzed the crash reports. The researchers studied the construction of crash reporting systems [11], the replication of crashes [29], the cause of crashes [30] and the prediction of crash-prone modules [31]. Our work focuses on grouping of the duplicate crash report by comparing the stack traces similarities. In our case study we analyzed the Soccoro[6] crash report system used by Mozilla.

[6] https://wiki.mozilla.org/Socorro:Overview

2.2.1 Windows Error Reporting

WER[7] (Windows Error Reporting) is a system developed by Microsoft for handling field crash-reports. WER predates other crash-reporting tools and has a very large user base compared to Soccoro, since it is used with all Microsoft applications, e.g., Windows, Internet Explorer and Microsoft Office. Glerum *et al.*[11] discuss the internals of the WER system. WER performs a progressive data collection of field crashes, whenever a crash occurs on the user's side, only a crash label is sent to the server. Developers need to configure the server if they wish to receive detailed crash-reports for a crash label. WER groups the crash-reports using a bucketing algorithm. The bucketing algorithm uses multiple heuristics specific to the application supported by WER and manually updated by developers. In contrast, the approach suggested in this thesis does not require any intervention from developers.

2.2.2 Identification of Duplicate Crash Report Using Stack Trace

Researchers have investigated the usage of failing stack traces to identify duplicate crash-reports. Dang *et al.*[32] propose a re-bucketing approach for the WER system. Our research predates their work. Their approach also uses the stack trace similarity. They propose the PDM (Position Dependent Model) to compare the stack traces, which gives more weight to the frames closer to the top of the stack trace. They compute the distance between stack traces using the PDM model and apply hierarchal grouping to bind the duplicate crash reports.

Modani *et al.*[33] also propose a technique to identify the duplicate crash reports by comparing the stack traces, but their technique is limited to crash-reports caused by known bugs. They use two similarity matrices, top-K indexing and inverted indexing. The top-K indexing

[7] http://msdn.microsoft.com/gg487440.aspx

compares the top-K frame of a stack trace and the inverted indexing compares the number of common functions in the stack traces.

Liu and Han [34] propose a technique to identify duplicate stack traces using the fault localization methods. Their approach is based on the intuition that the two failing stack traces are similar if the fault localization methods suggest the same fault location.

Bartz *et al.*[35] also propose a technique for finding similar crash reports based on stack trace similarity. They use a tuned call stack trace and their technique requires tuning eleven parameters to compare the stack traces. Automatic learning of the optimal values of these parameters incurs heavy computational cost, therefore the technique is hard to implement.

Brodie and Champlin [36, 37] use the stack-trace comparison to identify similar bugs. But, their approach makes use of historical data of already known problems. From a collection of different stack-traces of an already known problem, they develop a stack-trace pattern for each problem. Whenever a new problem is reported, it is compared to existing patterns of known problems and if a match is found, support staff can use this knowledge to handle the issue.

In comparison with other approaches to detect duplicate crash reports, we propose a simpler and application independent approach. The grouping techniques proposed by other researchers assumes that all the crash reports are available before grouping, whereas we propose a technique that can group a continuous stream of incoming crash reports. Our work also evaluates the impact of crash report grouping on the defect fix time. Moreover, the crash graphs proposed by Kim *et al.*[38] to identify fixable crashes in advance, can also be applied with our grouping approach. The bucketing algorithm of WER can be easily replaced with our simpler and application independent grouping approach to predict fixable crashes.

2.2.3 Bug Correlation and Localization

Grouping of field crash-reports is similar to bug correlation, as we try to find which two crash-reports are correlated. There has been an extensive research on automatic bug correlation and bug localization.

Schroter *et al.*[39] investigate the use of stack trace for bug fixing through an empirical study of the bugs in Eclipse. They observe that for 60% of crashes, bugs were fixed in one of the method appeared in the failing stack trace. Our study confirms the result and we use this result as base for our grouping algorithm.

Chan and Zou [40] propose the use of visualization for bug correlation and the identification of relation between different crashes. But given the large number of crash-reports (2.5 M crash reports every day), visualization cannot be used to comprehend all the crash-reports. However, when crash-reports are grouped together correctly, the visualization of representative reports from each group can be used to find correlation between different bugs.

Liblit *et al.*[41] analyze predicate patterns in correct and incorrect executions traces and proposed an algorithm to separate the effects of different bugs in order to identify predictors associated with individual bugs. They claim that their algorithm is able to detect a wide variety of both anticipated and unanticipated causes of failure.

However, none of the techniques mentioned in these works can be used to analyze stack trace from crash-reports. These techniques are all dependent on instrumentation, predicates, and coverage reports or successful traces. This needed information is not available in crash-reports.

2.3 Analysis Techniques

In this section, we discuss the test and modeling techniques used in the case studies in Chapter 3 and Chapter 4.

2.3.1 K-Mean Clustering

K-mean is an unsupervised learning algorithm to classify a given data set through a certain number of clusters (assume K clusters) [42]. The algorithm was first developed for numerical data, but it can also be used for categorical and mixed data (i.e., to group the data points with multiple types of attributes) [43].The K-mean algorithm proceeds as the following steps:

1. In the data set, K data points are selected randomly as the centroids, and other data points are clustered with the closest centroids.
2. For each data point the distance is calculated from the data point to each cluster. If the data point is not closest to its own cluster, it is moved into the closest cluster.
3. The above step is repeated until a complete pass through all the data points result in no data point moving from one cluster to another.

The K-mean algorithm is not guaranteed to converge, therefore in practice a bound on the number of iterations is applied to guarantee termination [42]. The K-mean algorithm is sensitive to the selection of initial centroids. Researchers have proposed methods for selecting the initial centroids [44, 45].

2.3.2 Mean Square Contingency Coefficient

Mean Square Contingency Coefficient or the phi (ϕ) coefficient is a measure of association for two binary variables [46].

Table 2-1: Contingency table for two binary variables

	Y=1	***Y=0***	***Total***
X=1	n_{11}	n_{10}	n_{r1}
X=0	n_{01}	n_{00}	n_{r2}
Total	n_{c2}	n_{c1}	n

Table 2-1 is the contingency table for two binary variable X and Y. The contingency table displays the frequency distribution of the variables. From the contingency table, the ϕ coefficient can be computed as in Equation (2-1).Value of ϕ coefficient is bounded between -1.0 and +1.0, and a coefficient with a large absolute value indicates a strong association between two variables.

$$\emptyset(\boldsymbol{X},\boldsymbol{Y}) = \frac{\boldsymbol{n_{11}n_{00}} - \boldsymbol{n_{10}n_{01}}}{\sqrt{\boldsymbol{n_{c1}} * \boldsymbol{n_{c2}} * \boldsymbol{n_{r1}} * \boldsymbol{n_{r2}}}} \qquad \textbf{Equation (2-1)}$$

2.3.3 Wilcoxon rank sum test

The Wilcoxon-rank sum test [47] is a non-parametric test. It is used to assess whether one of two groups of data tends to have larger value than the other. For example if the bugs in a software are classified in two groups. Wilcoxon rank sum test can be used to determine if the life time of the bugs in one category tends to be larger than the life time of the bugs in other category. The test is statistically significant if the p-value of the test is less than 0.01.

2.3.4 Kruskal-Wallis rank sum test

The Kruskal-Wallis test [47] is used to test whether multiple samples are originated from the same distribution or not. For example if the bugs in a software are classified in multiple groups. A null hypothesis assumes that the life time of bugs in each category have the same

median value. If the result of the Kruskal-Wallis is statistically significant, then the null hypothesis can be rejected, i.e., bugs in different categories do not have the same median value. The test is statistically significant if the p-value of the test is less than 0.01.

2.4 Summary

In this chapter, we give an overview of previous work in the area of selective code integration, recovery of source code dependencies, field crash reports and grouping of field crash reports. We also discuss tests and models used in our case studies, including the K-Mean clustering, Contingency coefficient, Levenshtein distance, Wilcoxon rank sum test and Kurskal-Wallis rank sum test.

Chapter 3

An Approach to Identify Commit Dependencies in Selective Integration

In this chapter we present our approach to assists software developers to find the dependencies among commits (i.e., code changes). The approach is useful during selective code integration to mitigate the integration failure caused by missing commit dependencies. First we present an overview of the selective integration process, and next we discuss our approach. Lastly, we present a case study where we applied our approach on a software product family of mobile applications.

3.1 Overview of Selective Integration Process

The branching model for a software product line is determined by many factors, for example the size of software, the number of software in the product family, and development model applied by the software product line. However, in general the branching model for a software product line includes a code branch that contains the source code shared by all the products (i.e. trunk), and separate branches for each product containing the product specific code (i.e. product branches). As the products grow, new features are added to the products. At the development level the features are called functional changes. The functional changes added to the common part of the code are selectively integrated into the product branches, as per the requirements of the products. The selective integration process is facilitated by the following three systems:

1. ***Version control system*** (e.g., Git[8])*:* The version control system maintains the branching model and the source code in the branches.
2. ***Change tracking system*** (e.g., Bugzilla[9]): The change tracking system maintains the requests/tickets for the functional changes in the products. A request can be for a new feature in a product or a bug reported in a released version of a product. The requests are called as Change Requests (CR). This system also helps developers to maintain the dependencies among the CRs. A change request CR_1 is considered dependent on another change request CR_2 if for example, CR_1 is a request to fix a bug in code elements related to CR_2 or CR_1 is an enhancement over CR_2.
3. ***Integration support system:*** An integration support system helps to maintain the mapping between the change requests and the commits. When the developers add new commits, they link the commits with the change requests, and these links are used to identify the commits for the change request during the selective integration. An integration support system can be built into the version control system or it can be a separate system. For example when a developer adds new commit, the version control system allows the developer to add comments with the commit, these comment can be used to annotate the commit with the change request id. In the case study we performed, the organization developed an in-house tool, to maintain the links between the commits and the change requests. The links added by the developers are of two types:
    a. ***Commit to CR Links:*** When developers implement a CR, they link the commits with the CR. A commit can be linked with multiple CRs and multiple commits can be linked with single CR.

[8] http://git-scm.com/
[9] http://www.bugzilla.org/

b. ***Commit to Commit Links:*** Often the implementation of a CR is modified during the development. When developers commit the initial implementation for a CR, they link the commit with the CR. However, the subsequent commits are linked with the previous commit to mark the dependencies. In another scenario, if two commits implementing separate CRs, are dependent on each other, the commits are linked with each other.

The *Integration Support System* also keeps track of the integration state of each CR. A CR can have the following integration states:

a. ***Ready for Integration***: When a CR is implemented and merged into the trunk, the CR is marked as ready for integration into product branches.
b. ***Integrated***: When a CR is successfully integrated into a product branch, the CR is marked as integrated.
c. ***Failed to Integrate***: When the integration of a CR into a product branch fails, the CR is marked as failed to integrate.

A typical selective code integration process involves two types of actors:

a. ***Developer***: A developer commits the code changes for a CR into the development branch and integrates the CR into the trunk. Developers implement features for all the products.
b. ***Integrator***: An integrator belongs to a specific product team. Integrators selectively integrate the CRs from the trunk into their product branches.

3.1.1 Steps of Selective Code Integration

The following actions are performed during the selective integration of a CR:

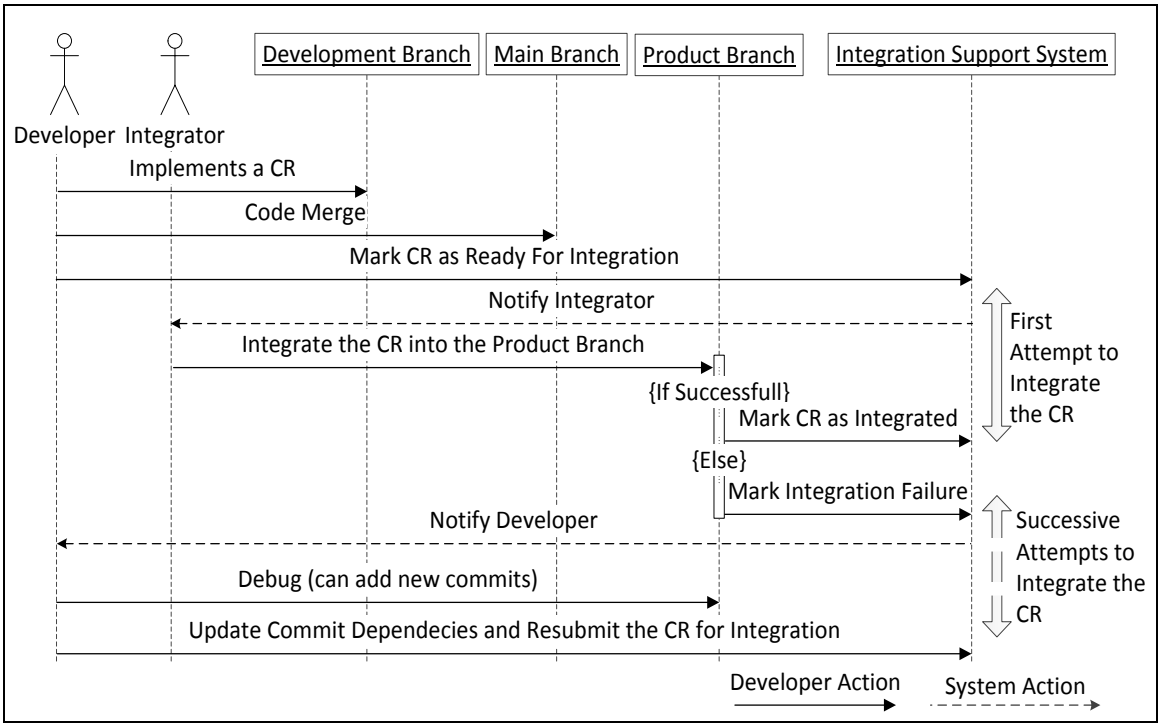

Figure 3-1: Sequence Diagram of Selective Integration of a Change Request

1. A developer implements a CR, commits the code changes into the development branch and links the commits with the CR. A CR can also be implemented by multiple developers.
2. When the CR is implemented and tested in the development branch, it is merged into the trunk and marked as ready for integration into product branches.
3. Once the CR is marked as ready for integration, all product integrators are notified. Each integrator checks if the CR is applicable for his/her product. If applicable, the integrator integrates all the commits linked to the CR into the product branch.
4. When integrating a CR, all commits linked directly or indirectly with the CR are integrated into the product branch. For example, if a commit of a change request CR_1 is linked with commits from a change request CR_2, and the change request CR_1 is

dependent on another change request CR_3 (in the CR tracking system), the integration of CR_1 requires that all commits linked with CR_1, CR_2 and CR_3 be integrated all together.

5. When the integration succeeds, the CR is marked as integrated and the process is complete. Otherwise, the CR is marked with an integration failure notice and the developers who implemented the CR are notified.
6. Developers should debug the source code related to the CR, make the required changes and/or add the missing commit dependencies, and resubmit the CR for integration.

3.2 Commit Grouping

When integrating a CR into a code branch, developers have to incorporate all the commits linked with the CR. However, very often some commits linked with a CR depend on other commits which are not directly linked with the CR. Sometimes, commit dependencies are dynamic, i.e. source codes of two commits are dependent only at runtime. Developers have hard time tracking indirect or dynamic dependencies when integrating CRs, and the missing dependencies cause integration failures. To assist developers during selective integration and prevent failures caused by missing dependencies, we propose two approaches to identify dependencies between commits. The approaches group dependent commits together and link them to the corresponding CRs.

The remainder of this section discusses the details of our identification of commit dependencies and the next section introduces the two grouping approaches.

3.2.1 Identification of Commit Dependencies

A commit is a set of source code changes submitted by a developer into a code branch. A commit contains information such as the list of files affected by the changes, the name of the developer submitting the changes, the date of submission, the description of the changes, and

other information about the code branch in which the changes are submitted. In this work, we represent a commit using a set of three attributes: the list of files modified in the commit, the name of the developer who submitted the commit, and the CR linked with the commit. We select these attributes because they contain essential information about the code elements that have been changed and the author of the commit.

To identify dependencies between commits, we introduce the following four dissimilarity metrics.

1. ***File Dependency Distance (FD):*** The file dependency distance (*FD*) measures the dissimilarity between two commits using source code dependencies between the classes contained in the files modified in the commits.

   If $C_a = \{ F_a, D_a, CR_a \}$ and $C_b = \{ F_b, D_b, CR_b \}$ represents two commits; with $F_a = < f_{a1}, \ldots f_{an} >$ and $F_b = < f_{b1}, \ldots f_{bm} >$ being respectively the lists of modified files in C_a and C_b; D_a and D_b the names of the developers who submitted C_a and C_b respectively; and CR_a and CR_b the CRs linked with C_a and C_b respectively, we define *FD* in Equation (3-1).

$$\boldsymbol{FD(C_a, C_b) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(1 - D_p(f_{ai}, f_{bj}))}{n * m}} \qquad \textbf{Equation (3-1)}$$

   Where $D_p(f_{ai}, f_{bj})$ measures the level of structural dependency between f_{ai} and f_{bj}

   A java file f_a is considered dependent on a java file f_b, if a class defined in f_a instantiates, invokes a function from, extends, or implements a class defined in f_b. To extract dependencies between files in a system, we parse the source code of the system and build a file dependency graph.

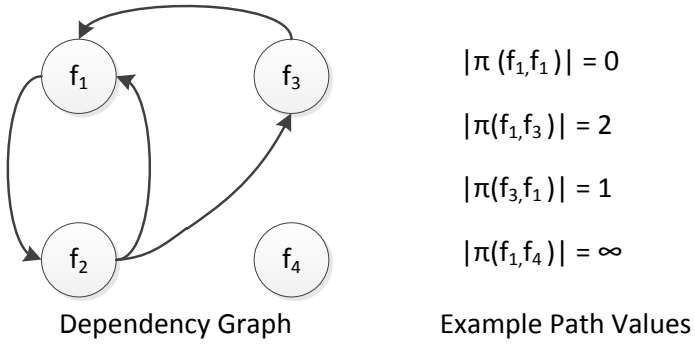

Figure 3-2: Source Dependency Graph

Figure 3-2 shows an example of the file dependency graph. Each node in a file dependency graph represents a source code file. A directed link between two nodes means that one file is dependent on the other file. We use dependency graphs to calculate the level of structural dependency between two files, f_a and f_b, following Equation (3-2).

$$\mathbf{D_p(f_a, f_b)} = \frac{\mathbf{1}}{\mathbf{2}} \left[\frac{\mathbf{1}}{\mathbf{1 + |\pi(f_a, f_b)|}} + \frac{\mathbf{1}}{\mathbf{1 + |\pi(f_b, f_a)|}} \right] \qquad \textbf{Equation (3-2)}$$

Where |π(fa,fb)| is the number of nodes on the shortest path from fa to fb in the dependency graph.

We consider both $|\pi(f_a,f_b)|$ and $|\pi(f_b, f_a)|$ in the definition of D_p (f_a, f_b) because dependency graphs are directed graphs.

2. ***File Association Distance (FA):*** In the definition of *file dependency distance*, the dependency between two files is measured using structural dependencies between the java classes defined in the files. However, two files can be dependent without a direct source code dependency, for example two files that changed together in history are considered logically dependent [48].

We introduce the File Association (*FA*) distance to capture logical dependencies between commits. We use the mean square contingency coefficient (ϕ) [49] to measure the level of logical association between files in commits. The ϕ coefficient is a measure of association between two events, e.g., the modification of two files f_a and f_b. The value of ϕ is computed using Equation (3-3).

$$\emptyset(\boldsymbol{f_a}, \boldsymbol{f_b}) = \frac{\boldsymbol{p_1} * \boldsymbol{p_4} - \boldsymbol{p_2} * \boldsymbol{p_3}}{\sqrt{(\boldsymbol{p_1} + \boldsymbol{p_2}) * (\boldsymbol{p_3} + \boldsymbol{p_4}) * (\boldsymbol{p_2} + \boldsymbol{p_4}) * (\boldsymbol{p_1} + \boldsymbol{p_3})}} \qquad \textbf{Equation (3-3)}$$

Where, p_1 is the number of commits in which both f_a and f_b were modified, p_2 is the number of commits in which f_a was modified but not f_b , p_3 is the number of commits in which f_b was modified but not f_a, and p_4 is the number of commits in which neither f_a nor f_b were modified.

$\phi(f_a,f_b)$ captures the co-occurrence of modifications of f_a and f_b. A value $\phi = 0$ indicates that there is no logical association between f_a and f_b, while a value $\phi = 1$ indicates a perfect association (i.e. f_a and f_b are always modified together).

The *FA* distance of two commits $C_a = \{ F_a, D_a, CR_a \}$ and $C_b = \{ F_b, D_b, CR_b \}$, where $F_a = < f_{a1}, \ldots f_{an} >$ and $F_b = < f_{b1}, \ldots f_{bm} >$, is defined in Equation (3-4).

$$\boldsymbol{FA}(\boldsymbol{C_a}, \boldsymbol{C_b}) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} (\mathbf{1} - \boldsymbol{\phi}(\boldsymbol{f_{ai}}, \boldsymbol{f_{bj}}))}{\boldsymbol{n} * \boldsymbol{m}} \qquad \textbf{Equation (3-4)}$$

3. ***Developer Dissimilarity Distance (DD):*** Often, dependent commits are submitted by developers who were collaborating on CRs in the past. We introduce the Developer Dissimilarity Distance (*DD*) to capture the working relation between two developers submitting related commits. Similar to *FA*, we use the ϕ coefficient to measure the

working relation between two developers. We define the ϕ coefficient for two developers D_a and D_b following Equation (3-5).

$$\boldsymbol{\phi(D_a, D_b) = \frac{p_1 * p_4 - p_2 * p_3}{\sqrt{(p_1 + p_2) * (p_3 + p_4) * (p_2 + p_4) * (p_1 + p_3)}}} \quad \textbf{Equation (3-5)}$$

Where p_1 is the number of CRs implemented by both D_a and D_b, p_2 is the number of CRs in which D_a contributed but not D_b, p_3 is the number of CRs in which D_b contributed but not D_a, and p_4 is the number of CRs in which neither D_a nor D_b contributed.

The *DD* distance between two commits $C_a = \{ F_a, D_a, CR_a \}$ and $C_b = \{ F_b, D_b, CR_b \}$, submitted respectively by developers D_a and D_b is defined in Equation (3-6).

$$\boldsymbol{DS(C_a, C_b) = 1 - \phi(D_a, D_b)} \quad \textbf{Equation (3-6)}$$

4. ***Change Request Dependency Distance (CRD)****:* The Change Request Dependency Distance (*CRD*) captures the dissimilarity between two commits based on the CRs implemented by the commits. When two commits are linked with dependent CRs (as marked in the CR Tracking System), the *CRD* value is 0, otherwise the *CRD* value is 1. Equation (3-7) gives a formalization of the *CRD* between two commits, C_a and C_b.

$$\boldsymbol{CRD\ (C_a, C_b) = \begin{cases} 0 & if\ C_a and\ C_b are\ linked\ with\ dependent\ CR \\ 1 & Otherwise \end{cases}} \quad \textbf{Equation (3-7)}$$

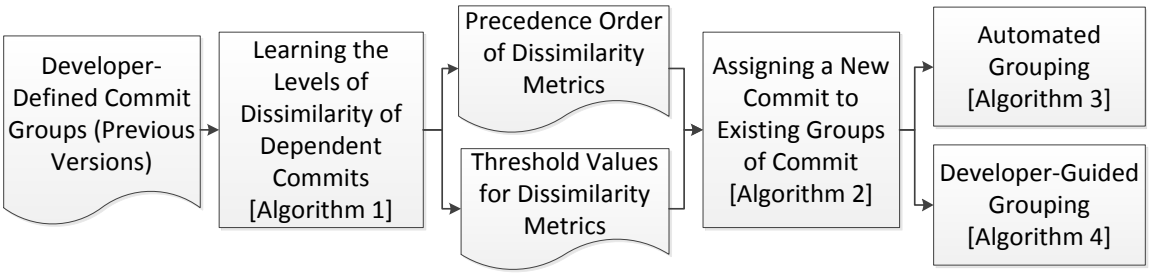

Figure 3-3: Overview of Grouping Approach

3.3 Commit Grouping Approach

Figure 3-3 gives an overview of our process to identify commit dependencies, and link dependent commits to the corresponding CR. The process is composed of three parts. The first part consists in learning levels of dissimilarity among dependent commits using historical commit data and the metrics from Section 3.2.1. The second part describes a process to assign a commit to existing groups of dependent commits. The third part defines the grouping algorithms for each of our two approaches. The algorithms are based on the dissimilarity levels learned in the first part, and use the commit assignment process from the second part. Following in this section we discuss each part in detail.

3.3.1 Learning the Levels of Dissimilarity among Dependent Commits

We learn the levels of dissimilarity among dependent and independent commits using commit data from previous versions. Algorithm 3-1 presents the steps of our learning process.

Algorithm 3-1: ***Learn_Historical_Data***

/ Learning the Dissimilarity Levels from Historical Data */*

input : Commits of Previous Version *C[ ]*

output: Dissimilarity metrics *D[ ]*, in order of precedence

output: *Max_Threshold* and *Min_Threshold* values for each dissimilarity metric

1. Organize the commits into *developer defined commit groups* ;
2. *D[ ] = [FD, FA, DD, CBD]* ;
3. **for each** dissimilarity metric *d* in *D[ ]*
4. {
5. **for each** Commit C_i in *C[ ]*
6. {
7. a_i = average dissimilarity of C_i with its own group;
8. *bi* = ***min*** (average dissimilarity of C_i with other groups);
9. $S_d(C_i) = (b_i - a_i) / max(b_i, a_i);$
10. }
11. $Max\_Threshold_d$ = ***mean*** $(b_i \forall C_i : C[ ])$;
12. $Min\_Threshold_d$ = ***mean*** $(a_i \forall C_i : C[ ])$;
13. S_d = ***mean*** $(S_d(C_i) \forall C_i : C[ ])$;
14. }
15. **sort** dissimilarity metrics in *D[ ]* in descending order of their S_d value ;

- Using the links (added manually by developers) between commits and CRs of prior versions, i.e., commit to CR links and the CR dependency links, we organize the commits

into groups. We call these groups *developer defined groups*. To obtain *developer defined groups*, we assign commits linked together (by developers) in the same group, and commits with no link to separate groups. [Line 1]

- D[ ] is the list of dissimilarity metrics defined in Section 3.2.1 [Line 2]
- Using *developer defined groups* of prior versions, we compute the *Max_Threshold* value, the *Min_Threshold* value and the Silhouette value of each dissimilarity metric. [Line 3-13]
- We order dissimilarity metrics by their Silhouette values.[Line 15]

In summary, in our *developer defined groups*, each group represents a set of dependent commits, that were integrated together in order to integrate the CRs linked to the commits in the group. During the integration process of commits contained in *developer defined groups*, the links between the commits have been verified and corrected if necessary by the integrators. Therefore, the *developer defined groups* are correct and complete. In this study we use *developer defined groups* as our gold standard to learn dissimilarity levels (on previous versions of the system) and compute the performance of our grouping approaches (on future versions of the system).

To compute the *Silhouette* value [50] of dissimilarity metric *d*, on *developer defined groups*, we proceed as follows: given a commit C_i, we calculate the Silhouette $S_d(C_i)$ following Equation (3-8).

$$\boldsymbol{S_d(C_i) = \frac{b_i - a_i}{max(b_i, a_i)}} \qquad \textbf{Equation (3-8)}$$

Where a_i is the average dissimilarity between C_i and other commits from the same group; b_i is the minimum of average dissimilarity values between C_i and all the other commit groups.

The silhouette value of a dissimilarity metric *d* is the mean of the silhouette values $S_d(C_i)$ of each commits C_i in *C[ ]*. The silhouette value measures that how close are the commits in the same groups in comparison to how far are the commits in separate groups. The range of silhouette values is from -1 to 1. A silhouette value of 0 represents a random grouping, while a silhouette value of 1 represents a perfect grouping. A positive silhouette value for a dissimilarity metric indicates that the dissimilarity metric can be applied to identify dependent commits, and between two dissimilarity metric, a metric with the higher silhouette value is the better identifier of dependent commits in comparison to the other metric.

3.3.2 Assigning a Commit to Existing Groups of Commit

To assign a commit *C* into existing groups of commits *G[ ]*, we follow the steps described in Algorithm 3-2.

- For the commit *C*, we select the groups $G_{sel}[\ ]$, such that for all dissimilarity metrics, the dissimilarity between a selected group and the commit *C*, is lower than the *Max_Threshold* value of the respective dissimilarity metric. [Line 2]
- Among the selected groups in $G_{sel}[\ ]$, we find the group G_{min} that has the minimum dissimilarity with *C*, for the dissimilarity metric *d* with the highest ranking. If the dissimilarity between *C* and G_{min} is lower than the *Min_Threshold* value for the dissimilarity metric *d*, we assign the commit *C* to the group G_{min} and the algorithm terminates. Otherwise, we select *d* as the next dissimilarity metric in the order of precedence and repeat the comparison. [Line 3-11]
- If the commit is not assigned to any group, a new group is created for the commit. [Line 12-13]

Algorithm 3-2: **Assign_Groups**

/ Add a Commit in existing groups of commits */*

input : Commit *C*

input : List of existing groups *G[ ]*

1. *D[ ]* = dissimilarity metrics in order of precedence ;
2. $G_{sel}[\,] = \{ G_i \in G[\,] \mid (d_j(C,G_i) <= Max\_Threshold_{dj}), \forall\ d_j : D[\,]\}$;
3. **for each** dissimilarity metric *d* in *D[ ]*
4. {
5. G_{min} = the group in $G_{sel}[\,]$ with minimum value of $d(C,G_i)$;
6. **if** ($d(C,G_{min}) <= Min\_Threshold_d$)
7. {
8. assign *C* to G_{min};
9. exit;
10. }
11. }
12. assign C to a new group G_n ;
13. add G_n to *G[ ];*

3.3.3 Grouping Dependent Commits

Using Algorithm 3-1 and Algorithm 3-2, we propose two commit grouping approaches, which can identify dependencies between commits and form groups of dependent commits that should be integrated together. First approach is automated and the second approach is user guided. Algorithm 3-3 and Algorithm 3-4 present the details of the two approaches.

3.3.3.1 Automated Grouping Approach

When an integrator integrates a CR, he searches for all the commits related to the CR that are available in the trunk. The trunk contains commits linked with many CRs and the links between the commits are not always explicit. In this scenario, the automated grouping approach can help the integrator to recover missing dependencies among commits. The approach is presented in Algorithm 3-3. The approach is based on the K-mean grouping algorithm [42]. The K-mean algorithm moves the elements in the groups iteratively, until a stopping criterion is met. The K-mean algorithm is not guaranteed to converge, therefore in practice a bound on the number of iterations is applied to guarantee termination [42]. For our automated grouping approach, we fixed a maximum of 1000 iterations in case the algorithm does not terminate normally. However, we have observed through experiments that the grouping results of Algorithm 3-3 remain similar when the maximum number of iterations is chosen between 100 and 1000.

- For each commit C_i in *C[ ]*, we assign the commit C_i to a group in *G[ ]* using Algorithm 3-2. Initially *G[ ]* is empty. When a new commit is assigned, either a new group is created for the commit or if a group of similar commits has been created, the commit is assigned to the group of similar commits. [Line 1-5]
- Once all the commits are assigned to a group in *G[ ]*, we reassign the commits among the groups in *G[ ]* by using Algorithm 3-2. During the reassignment of the commits in *G[ ]*, a commit can move from one group to another, therefore the groups can change. If no commit is moved from one group to another group during the reassignment, the algorithm terminates, otherwise we reassign the commits again. To guarantee a convergence, the algorithm is terminated after a maximum number of iterations (i.e., 1000 iterations). [Line 6-18]

Algorithm 3-3: **Automated_Grouping**

```
/* An automatic approach to group the given commits*/
input : Number of Iterations L
input : Commits to group C[ ]
1.  Commit Groups G[ ];
2.  for each dissimilarity metric Ci in C[ ]
3.  {
4.      Assign_Group(Ci,G[ ] );
5.  }
6.  int iterations = 0;
7.  while ( iterations <= L )
8.  {
9.      for each dissimilarity metric Ci in C[ ]
10.     {
11.         Assign_Group(Ci,G[ ] );
12.     }
13.     If no commit moved from one group to another
14.     {
15.         break;
16.     }
17.     iteration++;
18. }
```

3.3.3.2 User Guided Grouping Approach

When a developer adds a new commit, he links the commit with a CR. However, a commit for a CR can be dependent on some other commit(s) from another CR. In this scenario, we recommend the user guided approach. The approach is presented in Algorithm 3-4.

Algorithm 3-4: **User_Guided_Grouping**

/ Add a Commit in existing groups of commits */*

input : Newly added commit C_n

input : Existing group of commits *G[ ]*

1. Assign_Group**(**C_n,*G[ ]* **);**
2. A developer verifies the grouping of C_n. If incorrect, the developer manually assigns C_n to the correct group;

- When a new commit is added, the commit is assigned to a group by using Algorithm 3-2. [Line 1]
- After the new commit is assigned to a group, a developer verifies if the commit is assigned correctly. If not, the developer manually assigns the commit to the correct group. [Line 2]

3.4 Case Study Design

To show the benefits of using our grouping approach, we performed a case study using a large scale enterprise software product family. The goal of this case study is two-fold: (1) validate our dissimilarity metrics; (2) evaluate the effectiveness of our proposed grouping algorithms. We formulate the following three research questions to evaluate our approach:

RQ1: Can the metrics FD, FA, DD and CRD, be used to group dependent commits?

RQ2: How efficient are our proposed grouping approaches?

RQ3: What is the impact of using our approach on a selective integration process?

3.4.1 Data Collection

In this case study we analyze a product family of enterprise mobile applications. Each product in the product family supports a different hand set device. The product family is written in java and follows a main line branching development model. We studied three versions of the product family. Because of enterprise policies, we cannot disclose the descriptive statistics of the data and the exact version numbers of the products. We name the three versions we studied as *V1, V2* and *V3*.

3.4.2 Data Processing

Figure 3-4 gives an overview of our data processing. We extract CRs from the CR Tracking System, and for each CR of either version *V1, V2* and *V3*, we retrieve the states (i.e., Ready for Integration, Integrated, and Failed to integrate) of the CR and the list of commits linked to the CR in the Integration Support System. We download the source code of *V1, V2* and *V3* from the Version Control System and extract dependencies between files. Using this data we apply our grouping algorithms to automatically group dependent commits and assess the effectiveness of our grouping approach. The remainder of this section elaborates on each of these data processing steps.

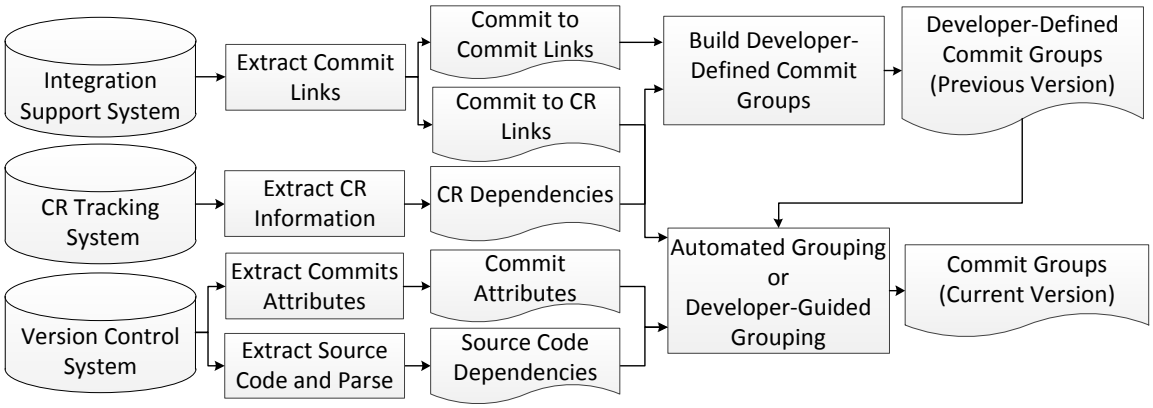

Figure 3-4: Data Collection and Commit Grouping

1. ***Mining CR Tracking System:*** We retrieved the CRs of versions *V1*, *V2* and *V3*, and the dependencies among the CRs for each version using the data collection API provided by the CR Tracking System.
2. ***Mining the Version Control System:*** On the source code of each downloaded version, we extract source code dependencies between every pair of files using our home made parser. This parser first lists the java class names defined in the source files and then for each source file identifies classes defined in other source files and referred in the source file. Using the data collection API provided by the Version Control System, we retrieve commit logs. We parse these commit logs to extract the following attributes for each commit: the list of file modified in the commit and the name of the developer who added the commit.
3. ***Mining the Integration Support System:*** The Internal Integration system used by the enterprise was developed internally. The system is integrated with the Version Control System to allow developers to link commits to related CRs. Using the API provided by the Integration Support System, we retrieved all the commits to CR links, the commit to commit links, and the history of the states of all the CRs from versions *V1*, *V2* and *V3*.

3.5 Case Study Results

This section presents and discusses the results of our three research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion of our findings.

3.5.1 RQ1: Can the metrics *FD, FA, DD* and *CRD,* be used to group dependent commits?

1. ***Motivation:*** In Section 3.2.1 we proposed four metrics to measure dissimilarities between commits. The effectiveness of the grouping approach presented in this paper is dependent on the ability of the proposed metrics to identify dependent commits. In this research question, we evaluate the effectiveness of each of our four metrics in identifying dependent commits.
2. ***Approach:*** We evaluate each of our metrics using the *developer defined groups* of versions *V1, V2* and *V3*, defined in Section 3.2.1. Using the *developer defined groups*, we compute dissimilarity values among the dependent commits and among the not dependent commits for each metric. We expect that two commits from the same group will have a low dissimilarity value, while commits from separate groups will have a high dissimilarity value. To measure the ability of our metrics to identify commit dependencies, we compute silhouette values for each metric following the Equation (3-8) from Section 3.2.1. A positive silhouette value means that the metric can successfully identify two linked commits. The higher is the silhouette value, the stronger is the ability of the metric to identify linked commits.
3. ***Finding:*** The *CRD* metric is the strongest at identifying linked commits. Table 3-1 shows the Silhouette values of the four dissimilarity metrics measured on commits from the three versions we studied. *CRD* has the highest Silhouette and the values are close to 1.

Table 3-1: Silhouette values of the proposed Dissimilarity Metrics

Dissimilarity Measure	***Version***		
	V1	***V2***	***V3***
CR Distance (CRD)	0.94	0.96	0.96
File Association Distance (FA)	0.76	0.79	0.67
Developer Dissimilarity Distance (DD)	0.63	0.67	0.57
File Dependency Distance (FD)	0.46	0.47	0.49

This result was expected since one can expect commits linked together to belong to the same CR or to dependent CRs. The File Association (*FA)* metric has the next highest values, indicating that files from dependent commits are frequently changed together. The next in the precedence order is Developer Dissimilarity (*DD*), indicating that dependent commits are often added by developers with a history of joint work on CRs. The File Dependency (*FD*) metric is reported to be the least effective of our four metrics, with an average silhouette value of 0.47. However, the average *FD* dissimilarity between two dependent commits is 0.24 and the average *FD* dissimilarity between two commits with no link is 0.47; indicating that files modified in dependent commits have slightly stronger source code dependencies compared to files modified in independent commits (i.e., commits with no link). Nevertheless, *FD* remains the least effective of our four dissimilarity metrics.

3.5.2 RQ2: How efficient are our proposed grouping approaches?

1. ***Motivation:*** In Section 3.3.3, we defined two approaches to group dependent commits, in order to assist a developer during a selective integration process. The prime motivation for automating the grouping of commits is to save development time and resources.

Therefore, in this research question we evaluate how correctly and how efficiently the proposed approaches can group dependent commits.

2. ***Approach:*** To evaluate the efficiency of our proposed automatic and user guided approaches, we first use *developer defined groups* from version *V1* to learn dissimilarity levels necessary to calibrate the grouping algorithms. We then apply the two grouping algorithms on the commits of *V2*. Second, we calibrate the two grouping algorithms using developer defined groups from version *V1* and *V2*, and we apply the grouping algorithms on the commits of version *V3*.

   To assess the correctness of the groups created by our two algorithms, we use *developer defined groups* of versions *V2* and *V3* as our gold standard. We compare how well the commit dependency links recovered by our grouping approaches (i.e., the groups created by Algorithm 3-3 and Algorithm 3-4) match the links established in the Integration Support System by the developers of *V2* and *V3* (i.e., the *developer defined groups*).

   We compute the precision and the recall of the automated grouping approach (i.e., Algorithm 3-3) using respectively Equation (3-9) and Equation (3-10). The precision value measures the fraction of retrieved commit dependency links that are correct, while the recall value measures the fraction of correct commit dependency links that are retrieved.

$$\boldsymbol{Precision} = \frac{|\{\boldsymbol{correct\ link}\} \cap \{\boldsymbol{retrieved\ link}\}|}{|\{\boldsymbol{retrieved\ link}\}|} \qquad \textbf{Equation (3-9)}$$

$$\boldsymbol{Recall} = \frac{|\{\boldsymbol{correct\ link}\} \cap \{\boldsymbol{retrieved\ link}\}|}{|\{\boldsymbol{correct\ link}\}|} \qquad \textbf{Equation (3-10)}$$

In the case of our user guided approach (i.e., Algorithm 3-4), we use developer defined groups to simulate feedbacks from developers. Specifically, we proceed as follow: after adding a commit, we compare the grouping result of our algorithm with the groups in developer defined groups. If the commit is not assigned to the correct group of dependent commits, we override the result of our algorithm and assign the commit to the correct group suggested by developer defined groups. Because our user guided approach modifies its grouping results to match with the developer defined groups, the precision and recall values at the end of the user guided grouping process are 100%.

To assess the effectiveness of Algorithm 3-3 in assigning commits to their appropriate groups, we use the accuracy metric defined in Equation (3-11). The accuracy measures the fraction of commits that are assigned correctly (i.e., that didn't needed to be corrected using developers' feedback).

$$\boldsymbol{Accuracy} = \frac{|\{\boldsymbol{commits\ assigned\ correctly}\}|}{|\{\boldsymbol{all\ commits}\}|} \qquad \textbf{Equation (3-11)}$$

3. ***Finding:*** Table 3-2 shows the precision and recall values for the automated grouping approach and the accuracy values for the user-guided approach for versions *V2* and *V3*. When applying the grouping approaches to the commits of version *V2* (respectively version *V3*), we used historical information from version *V1* (respectively versions *V1* and *V2*). The results for version *V3* are better than the results for version *V2*, indicating that a longer history can help improve the effectiveness of Algorithm 3-3 and Algorithm 3-4.

Table 3-2: Precision, Recall and Accuracy of the Group Approach

	Grouping Approach	***Version***	
		V2	***V3***
Precision	Automated Grouping	92%	95%
Recall	Automated Grouping	79%	82%
Accuracy	User Guided	96%	98%

4. ***Complexity of the grouping approaches:*** Our two proposed approaches use Algorithm 3-2 to assign a commit to a group. This algorithm compares a given commit with all the existing groups. The maximum number of commits groups can be the same as the number of commits, therefore the complexity of the algorithm is *O(n)*, where *n* is the number of commits to be grouped. In user-guided approach when developer adds a new commit, the Algorithm 3-2 is executed to find the group of dependent commits, therefore the complexity of the user guided approach to assign a commit to the group of dependent commits remains *O(n)*.

In the automated grouping approach, Algorithm 3-2 is applied for each existing commit. If any commit moves from one group to another, the process is repeated. Therefore the overall complexity of the automated approach is $O(l.n^2)$, where *l* is the number of iterations. We have limited the number of iterations to 1000, which restricts the complexity of the automated approach to $O(n^2)$. Moreover, the automated grouping approach is based on the K-mean algorithm and the number of iterations of the K-mean algorithm depends on the selection of the initial element of each group [10]. In our automated grouping approach, the initial groups are created by Algorithm 3-2, which ensures that only the independent commits are selected as initial element of a group.

3.5.3 RQ3: What is the impact of using our approach on a selective integration process?

1. ***Motivation:*** On average, after a CR is added to the trunk, it takes 6 days to integrate the CR to a production branch. But, when the integration of a CR fails, it takes 32 days to reintegrate the CR to the same production branch. Integration failures are often caused by missing commit dependencies or incorrect implementations. When a CR fails to integrate, the CR is returned to a developer. The developer debugs/corrects the implementation of the CR and re-submits the CR for integration. On average, an integration failure increases the release time of a CR by 28 days. We want to evaluate if our proposed grouping approaches can reduce integration failures caused by missing dependencies between commits.
2. ***Approach:*** To resolve an integration failure caused by missing dependencies, developers link some existing commits with the failing CR. Sometimes, a new commit is created and linked to the CR. The CR is then resubmitted for integration. We assess the effectiveness of our grouping approaches in preventing integration failures by computing a Failure Reduction Rate (FRR). The FRR measures the fraction of missing dependencies between commits (responsible of integration failures) that are successfully recovered by our grouping approaches. To compute FFR we proceed as follows: first we identify all CRs that ever failed during an integration (i.e., CRs that ever had the state Failed to Integrate); second, we select among the obtained CRs, those for which the failure was caused by missing dependencies; third we compute for each selected CR (cr), the set M_{cr} of missing dependencies responsible for the failure. For each of our grouping approaches, we compute FRR following Equation (3-12).

$$FRR = \frac{|\{cr \in F \mid M_{cr} \subset R\}|}{|F|}$$ **Equation (3-12)**

Where F is the set of CRs that failed during an integration because of a missing dependency; and R is the set of commit dependencies retrieved by our proposed grouping approach.

3. ***Finding:*** In our data set in versions V2 and V3, 1.7% of the CRs failed during an integration process, because of missing commit dependencies. This proportion represents many hundreds of CRs failed to integrate in the two versions. The, FRR of our automated grouping (respectively user guided) approach is 76% (respectively 94%). Using our two grouping approaches, integration failures caused by missing dependencies can be avoided in up to 94% of the cases.

   1.7% represents a small portion of the total number of CRs implemented in a version, but a single CR integration failure can delay the release of a version. Therefore, by reducing the integration failures, our approach helps to limit the delay (caused by missing dependencies) in release time of a version.

3.6 Threats to Validity

This section discusses the threats to validity of the case study we discussed in this chapter, following the guidelines for case study research [51].

Construct validity threats concern the measurement errors in an experiment. We collected the data from the CR tracking system, the Version Control System and the integration support system using the API's provided by each system respectively. We computed the file dependencies by parsing the source files. Our parser captures the static dependencies among source files, however the dynamic dependencies among source files are not captured.

Internal validity threats concern our selection of subject systems, tools, and analysis method. Although we study multiple versions of a large enterprise software system, some of the findings might still be specific to the development and maintenance process of our selected software system.

Conclusion validity threats concern the relation between the treatment and the outcome. In evaluation of the user-guided approach, we assume that if a commit is assigned incorrectly, developer will manually assign the commit to a group of dependent commits. However, if the developer doesn't reassign an incorrectly assigned commit, the subsequent commits can be assigned to other groups than the groups we observed in this experiment.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details of our approach to replicate our study. However, the data used for the case study is proprietary of Research in Motion Ltd., and was available only after signing a non-disclosure agreement with the organization. The data may be available to scholars interested in replicating the study, if the scholars approach the organization for a similar agreement.

External validity threats concern the possibility to generalize our results. We have conducted the study using three versions of a large enterprise mobile software application. Gaining access to such industrial data is hard. Nevertheless, further validation on a larger set of software applications is desirable, considering applications from different domains, as well as several applications from the same domain.

3.7 Summary

Selective code integration is an integral part of product line development. However, the selective code integration remains a risky process in which developers can miss commit dependencies. Missing commit dependencies cause integration failures which delay significantly the delivery of CRs. We propose four dissimilarity metrics that can be applied to retrieve dependencies between commits. Based on the dissimilarity levels learned from previous versions of a software system, we propose two approaches to group the dependent commits: a user guided approach, and an automated approach. The user-guided approach can be applied when a developer adds a new commit, while the automated approach can be applied when an integrator integrates a CR. Evaluations of our approaches on data derived from a product line of mobile software applications shows that the user-guided grouping approach and the automated grouping approach can reduce integration failures by 94% and 76% respectively.

Chapter 4

An approach to identify similar Field Crash Reports

In this chapter we present our approach to assist the software developers to identify similar field crash reports and group them together correctly. First we present an overview of a crash report system, next we discuss our approach to group the field crash reports. Lastly, we present a case study where we apply our approach on Mozilla crash reports and validate the approach.

4.1 Overview of Crash Report System

A Crash Report system for a software system includes three entities:

1. ***Client Side Crash Reporter*** (e.g., Breakpad[10]): The client side crash reporter is embedded in the software. It collects the information when the software crashes and sends it to the software vendor.
2. ***Crash Report Server*** (e.g., Socorro[11]): Crash report server collects the crash reports and keep them well organized. The function of report server includes sampling the crash reports, and identifying and grouping similar crash reports. Each group of similar crash reports is called as a *crash-type*.
3. ***Bug Tracking system*** (e.g., Bugzilla[12]): Bug tracking system maintains a bug report for each filed bug. A bug report contains detailed information about a bug, such as the bug open date, the last modification date, and the bug status. Developers file the bugs in the bug tracking system, for the *crash-types* formed in the crash report server.

[10] https://wiki.mozilla.org/Breakpad
[11] https://wiki.mozilla.org/Socorro
[12] https://bugzilla.mozilla.org

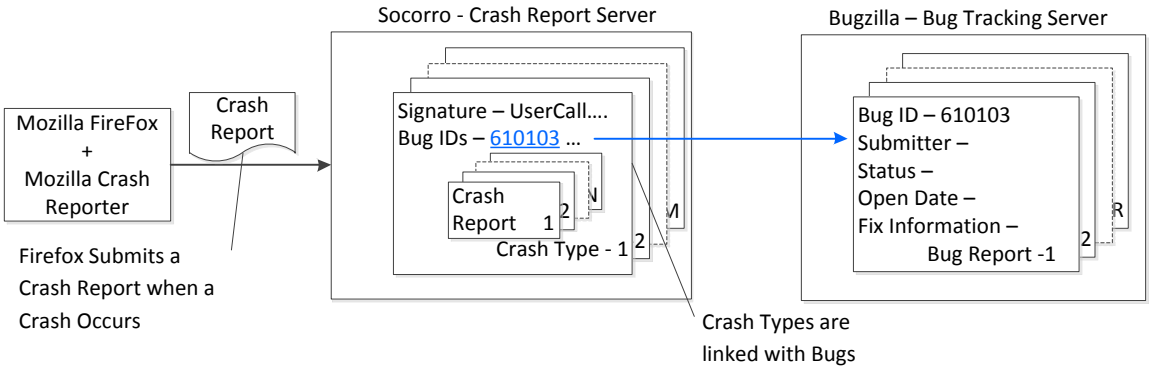

Figure 4-1: Mozilla Crash Report System

Mozilla Firefox supports collection of field crash-reports. Figure 4-1 gives an overview of the Mozilla crash report system. Firefox receives 2.5 million crash-reports every day. When Firefox is terminated unexpectedly, the Mozilla Cash Reporter sends a detailed crash-report to the Socorro crash report server. Figure 4-2 shows a sample crash-report for Mozilla Firefox.

Crash Report – e1c1267874640-94324-32423

Crash Time - OCT 24, 2010 11:20:53
Firefox Install Time – SEP 22, 2010 10:20:15
System Uptime – 1125 seconds
Version- 3.6.13
OS – Windows NT 6.1 2600
CPU – x86
User Comment –
Stack Trace –

Frame	Module	Signature	Source
0		@0x654789	
1	User32.dll	UserCallWinProcCheckWow	
2	User32.dll	DispatchMethod	
3	User32.dll	DispatchMessage	
4	XUI.dll	ProcessNextNativeEvent	Src/win/nsAppShell.cpp:179
5	XUI.dll	nsShell::OnProcess	Src/win/nsShell.cpp:77
6	Nspr4.dll	mozilla::Pump	Ipc/glue/MessagePump.cpp:134
7	XUI.dll	MessageLoop:Run	Ipc/glue/MessageLoop.c:784

Each Crash Report is assigned a unique ID

User Environment Information

All crash Reports with top signature as "*UserCallWinCheckWow*" are grouped together

Not all frames have Source Information

Figure 4-2: A sample Crash-report for Mozilla Firefox

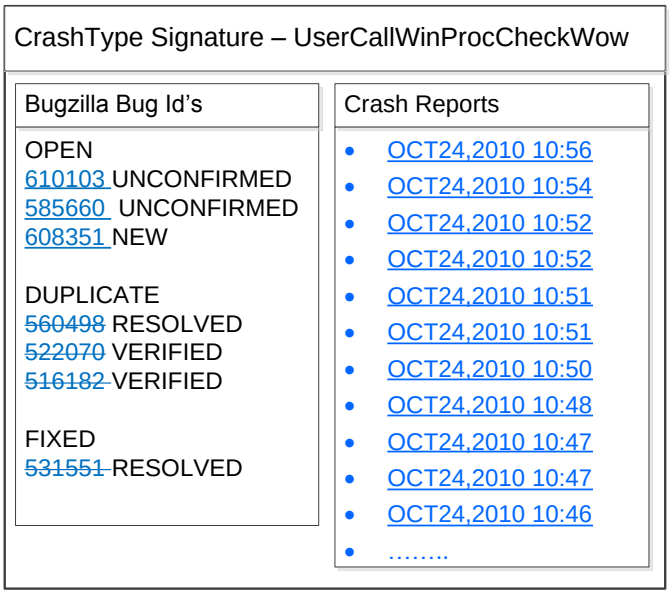

Figure 4-3: A sample crash-type from Socorro

The crash-report includes the stack trace of the failing thread and other information about the user environment, such as operating system, Firefox version, install time, and a list of plug-ins installed. Mozilla uses Socorro crash report server for organizing the crash reports. To organize the large number of crash-reports, Socorro groups the similar crash-reports as a crash-type. The intention is to organize the crash-reports caused by the same bug in one group. Socorro groups the crash-reports based on the top method signature of the stack trace. However, the subsequent frames in the stack trace might be different for different crash-reports in a crash-type. Figure 4-3 shows a crash-type from Socorro. For each crash-type, Socorro server provides a list of the crash-reports of the crash-type and a set of bugs filed for the crash-type.

Mozilla uses Bugzilla to track bugs and maintains a bug report for each filed bug. Along with the detailed information of the filed bugs, Bugzilla also contains the information about the bug-fixes for the fixed bugs. When a developer fixes a bug, he often submits a patch to Bugzilla. The patch includes source code changes and other configuration file changes. Once approved, the

patch code is integrated into the source code of Firefox. Patches can be used to identify where a bug is fixed. For the top crash-types, Firefox developers file bugs in Bugzilla and link them to the corresponding crash-type in Socorro server. Multiple bugs can be filed for a single crash-type and multiple crash-types can be associated with the same bug. As shown in Figure 4-3 the crash-type *UserCallWinProcCheckWow* is linked with seven bug reports from Bugzilla. Web interfaces for the Socorro server and Bugzilla are integrated, developers can navigate from a crash-type summary in the Socorro server to the bugs filed for the crash-type in Bugzilla.

4.2 Grouping Approach for Crash Reports

In the current approach the Socorro server compares only the top method signature in the failing stack traces from the crash-reports. As the crash-reports caused by different bugs can have the same top method signature in the failing stack trace, using this approach a crash-type might contain the crash-reports caused by multiple bugs. To group the crash-reports triggered by different bugs separately, we propose to enhance the existing crash-report grouping approach of Socorro. We suggest a detailed comparison of stack traces to group the crash reports. The existing approach is performance efficient as it compares only the top method signature of the stack traces. But a detailed comparison of the stack traces will cause following two issues:

1. ***Performance of grouping approach***: In the top ranked crash-types (i.e., the most frequently occurring crash-types), the number of crash-reports is very large. As a consequence, the detailed comparison of a large number of stack traces could be computation intensive. Therefore, the grouping approach must be performance efficient.
2. ***Fragile Groups***: Collection of field crash-reports and assigning the crash-reports to appropriate crash-types is a continuous process. If a grouping approach is applied to all the crash-reports collected for a crash-type, the organization of existing groups might be

changed each time a new crash-report is added into a crash-type. However, it is critical to maintain the groups over time. In particular, stable groups allow developers to analyze the crash-reports within a subgroup, file bugs for each group and refer back to the group. Given these two constrains, we suggest a grouping approach that groups the crash-reports efficiently without breaking the existing crash-types. The remainder of this section discusses the measures used in our approach and introduces the grouping approach.

4.2.1 Levenshtein Distance: A metric of stack trace dissimilarity

We use the Levenshtein distance [52] to evaluate the similarity between stack traces. The Levenshtein distance is used for comparing two sequences. It measures the amount of differences between the sequences. For example, the Levenshtein distance between "SUNDAY" and "SATURDAY" is three, since we need to insert letters 'A' and 'T' and replace the letter 'N' with 'R' to convert the former string to the later. As stack traces are sequences of frames, the Levenshtein distance between two stack traces is the number of changes needed to transform one stack trace into another, where a change can be inserting a frame, deleting a frame or replacing a frame. A change can be inserting a frame, deleting a frame or replacing a frame. We evaluate the Levenshtein distances between two stack traces, by comparing the top 10 frames of the stack traces, where the top frame refers to last method call in the stack. We limit the comparison of stack traces to the top 10 frames of each stack trace, since previous study [39] found that bugs are in general fixed in one of the top 10 frames from the failing stack trace.

4.2.2 Trace Diversity

Trace Diversity is the average dissimilarity (Levenshtein distance) among the stack traces in a group. The average value of Levenshtein distance for a group of crash-reports indicates the

diversity among all the stack traces of the group. We refer it as the *Trace Diversity* of the group of crash-reports.

A crash-type is a group of crash-reports, therefore we can compute the *Trace Diversity* of a crash-type. We measure the Trace Diversity for existing crash-type and determine the threshold value, i.e., the maximum value of the trace diversity for crash-types where the crash-reports are triggered by the same bug. We suggest that each group must have a *Trace Diversity* value less than the threshold value, such that we can ensure that all crash-reports in a subgroup are triggered by the same bug.

4.2.3 Representative Stack Trace

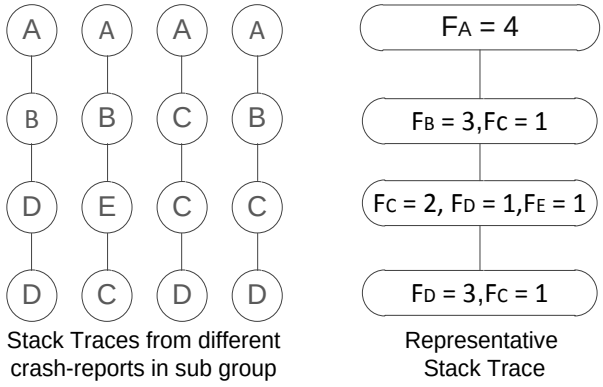

Figure 4-4: Representative stack trace for a group

A *representative stack trace* is a sequence presenting the number of appearance of the modules in each of the top 10 frames of the stack traces. More specifically, the i^{th} frame of a representative stack trace presents the number of appearance of each module that appears in the i^{th} frame of any stack trace from the group. Figure 4-4 shows an example group with four crash-reports. In this example, three crash-reports have the module *B* in the second frame of their stack trace and one crash-report has the module *C* in the second frame of its stack trace. Therefore, the second frame of the representative stack trace has a value "$F_B = 3, F_C = 1$". F_B in the second frame

denotes the number of appearance for module *B* in the second frame of the stack traces from the group.

The Levenshtein distance measures the amount of difference between two sequences. More specifically, Levenshtein algorithm incrementally combines the distances of individual nodes to compute the difference. If both sequences are of the same type, for any pair of nodes from each sequence, the distance is 0 if the nodes are the same; and the distance is 1 if the nodes are different. In our work, we compare the stack trace of a crash-report with the representative trace, which are not of the same type, because the stack trace is a sequence of frames and the representative trace is a sequence of set of frequencies. For a representative trace *R* and a stack trace *S*, the difference between any pair of nodes r and s, selected from *R* and *S* respectively, is defined in Equation (4-1). For example, the distance between a stack trace frame containing module *B*, and the second frame of the representative stack trace shown in Figure 4-4 (i.e. $F_B = 3$, $F_C = 1$) would be 1-3/4 = 0.25.

$$\boldsymbol{dist(r,s) = 1 - \frac{Freq(r,s)}{RC}} \qquad \textbf{Equation (4-1)}$$

Where Freq(r,s) is the frequency value of the module M in r, where M is the module appearing in s ; and RC is the total number of crash-reports in a subgroup.

The representative trace gives higher preference to more frequent frames, therefore only the new crash-reports containing the stack traces with frames similar to the frequent frames in a subgroup are added to the subgroup.

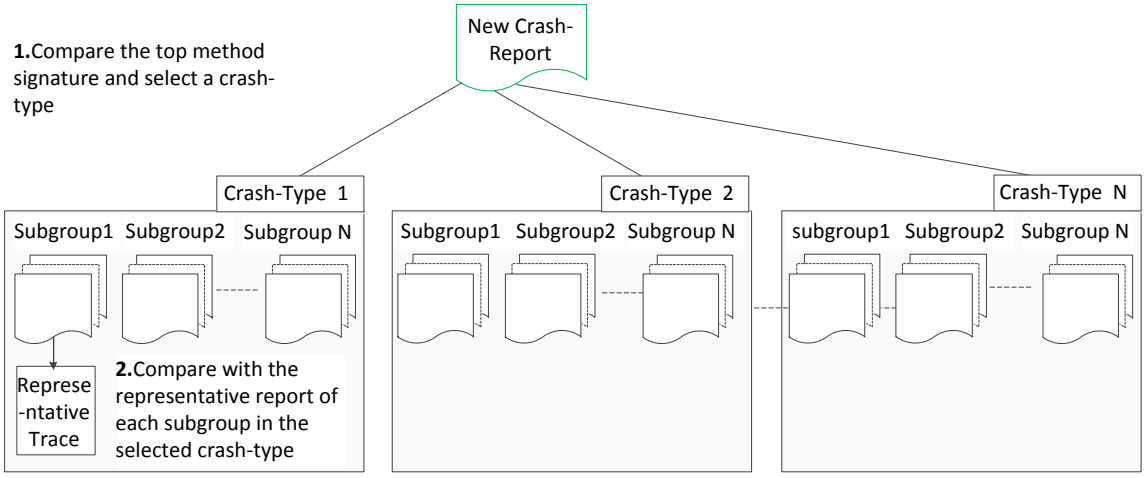

Figure 4-5: Two Level grouping of field crash reports

4.2.4 Grouping Approach

In our grouping approach we suggest a two level grouping of the crash-reports. The first level of grouping leverages the existing approach used by Socorro. It clusters the crash-reports based on the top method signature of the stack traces to form crash-types. Furthermore, we use a detailed comparison of stack traces to divide the crash-reports in a crash-type into subgroups. The subgroups within a crash-type create the second level grouping. Each subgroup is intended for developers to analyze and file bugs, instead of using the crash-types. Our approach subdivides crash-reports that have greater dissimilarities among stack traces within a crash-type. If the first level crash-types contain very similar crash-reports, such crash-types remain intact. Figure 4-5 shows the structure of the two- level grouping of crash-reports.

To improve the performance of the detailed comparison and maintain the structure of the already formed subgroups within a crash-type, we assign a representative trace for each sub-group. When a crash-report is received at the central repository, it is assigned to a crash-type based on the top method signature. In the selected crash-type, the new crash-report is compared with the

existing subgroups. To compare a crash-report with a subgroup, it is not compared with every report in the subgroup. Instead, the stack trace of the new report is compared with the representative trace of the subgroup. The new report is added to the subgroup with the minimum Levenshtein distance between the stack trace of the new crash-report and the representative trace of the subgroup. However, the Levenshtein distance value must be less than the threshold value; otherwise a new subgroup is created for the crash-report.

4.3 Case Study Design

To show the benefits of using our grouping approach, we performed a case study using the crash reports from ten beta releases of Mozilla Firefox. In this study, we examine the usefulness of stack traces for bug fixing activities and evaluate the current grouping approach used in Firefox. To validate and evaluate the proposed two-level grouping approach, we formulate following research questions:

RQ1: Can the stack traces in crash-reports help locate bugs?

Our first research question evaluates the usefulness of the stack traces in the crash-reports for bug fixing activities. We want to verify if it is useful to analyze the stack traces in crash-reports to identify the bugs.

RQ2: Does a crash-report grouping approach have an impact on bug fixing?

Our second question investigates the impacts of a crash-report grouping approach on the bug fixing time. Furthermore, we compare the trace diversity of existing crash-report groups and investigate the correlation between trace diversity and number of bugs linked with the group.

RQ3: Does a detailed comparison of stack trace help improve the grouping?

The third question evaluates our proposed grouping approach. This question verifies if the approach can group the crash-reports triggered by different bugs separately.

4.3.1 Data Collection

We sample crash-reports from ten beta releases of Firefox, ranging from Firefox-4.0b1 to Firefox-4.0b10. The beta releases are used for field testing. We download the summaries of the available crash-types and select the crash-types for which at least one bug is filed. For each selected crash-type we download 100 crash-reports (randomly sampled). We download all the available crash-reports for the crash-types which have less than 100 crash-reports. We parse the sampled crash-reports and extract the failing stack traces. Table 4-1 reports the descriptive statistics of our dataset.

Table 4-1: Descriptive Statistics of the Data Size

The number of crash-types with at least one bug filed	1,329
The total number of crash-reports sampled	82,156
The total number of bugs linked to crash-types	1,733
The number of fixed bugs	519
The number of duplicated bugs	253
The number of open bugs	961
The number of fixed bugs with a patch	231

For all the bugs filed for the crash-types in our data set, we retrieve the bug reports from the Bugzilla. If a patch is submitted for the bug, the bug report includes the patch. For every patch found in a bug report, we perform a syntactical analysis to retrieve information about what changes are made to fix the bug. We map this information on source code change locations to the stack trace in the crash-reports. Moreover, for each fixed bug we compute the bug fixing time, i.e., the difference between the bug open time and the last modification time. In the case of a bug resolved as DUPLICATE, if the original bug is filed for the same crash-type, we ignore the

duplicate bug. If the original bug is filed for some other crash-type, we link the original bug to where the duplicate bug was linked.

4.3.2 Data Analysis

RQ1: Can the stack traces in crash-reports help locate bugs?

In our first research question we investigate if stack traces contained in crash-reports can help developer to locate the bugs. We map the modules changed for bug fixing to the stack traces of the crash-reports. If the faulty module appears in any of the stack traces from the crash-type for which the bug is filed, we call it a *bug fixed in the linked stack trace*. If the faulty module appears in a stack trace from other crash-type, i.e. a crash-type not linked with the bug, we call it a *bug fixed in other stack traces*. If a bug is fixed in a module that has never appeared in a failing stack traces from any crash-type, we call it a *bug fixed elsewhere*. We compute the bug fixing time for the bugs and test the following null hypothesis:

H_{01}: the lifetime of a bug is the same for the bugs fixed in the linked stack traces, the bugs fixed in other stack traces and the bugs fixed elsewhere.

We use the Kruskal-Wallis rank sum test to investigate if the distribution of fixing times is the same for the bugs fixed in the linked stack traces, the bugs fixed in other stack traces and the bugs fixed elsewhere. The Kruskal-Wallis rank sum test is a non-parametric method for testing the equality of the population medians among different groups. It is an extension of the Wilcoxon rank sum test to 3 or more groups.

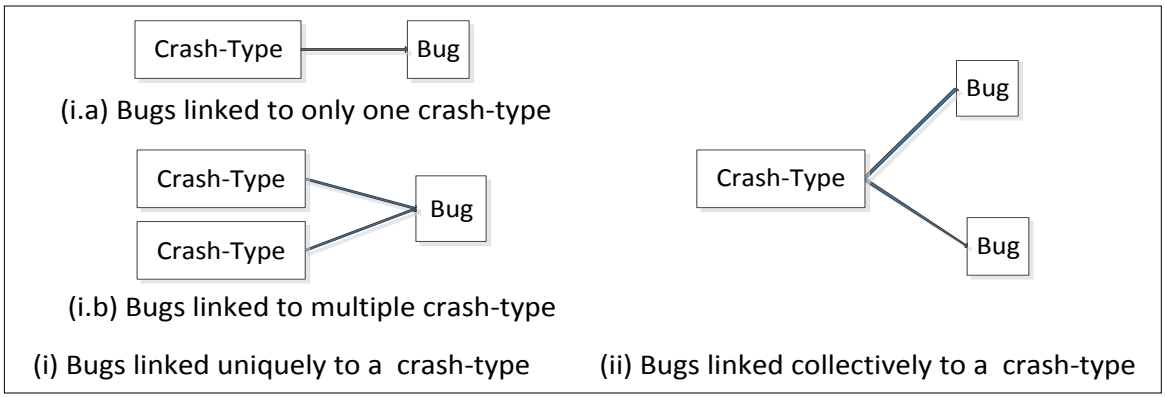

Figure 4-6: Categories of bugs based on the number of bugs linked to the crash-type

RQ2: Does a crash-report grouping approach have an impact on bug fixing?

In the second research question we investigate if the grouping of crash-reports has an impact on the bug fixing time. First we categorize the bugs by checking if the crash-reports triggered by a bug are grouped separately or if the crash-reports triggered by multiple bugs are grouped together. When the bugs are uniquely linked with one or more crash-types, it indicates that the crash-reports triggered by the bug are grouped separately. If multiple bugs are collectively linked with a crash-type, it indicates that the crash-reports triggered by multiple bugs are grouped together. Figure 4-6 presents the categories we defined for the bugs filed for the crash-types. We subdivide the bugs for which the crash reports are grouped separately, by checking if the crash-reports triggered by a bug are grouped together in a single crash-type, or split in multiple crash-types. We compare the fixing time for the categories and test the following two null hypotheses:

H^1_{02}: *the lifetime of a bug is the same for the bugs for which crash-reports triggered by multiple bugs are grouped together and for the bugs for which the crash-reports triggered by every individual bug are grouped separately.*

H^2_{02}: *the lifetime of a bug is the same for bugs for which the crash-reports are grouped in a single crash-type or crash-reports are split in multiple groups.*

We use the Wilcoxon rank sum test [47] to accept or reject H^1_{02} and H^2_{02}. The Wilcoxon rank sum test is a non-parametric statistical test used for assessing whether two independent distributions have equally large values. For example, we compute the Wilcoxon rank sum test to compare the distribution of the fixing time for the bugs linked to multiple crash–types and the bugs linked to a single crash-type.

Furthermore, we analyze the trace diversity of the crash-types, as discussed in Section 3. We analyze the relation between the trace diversity of crash-types and the number of bugs linked with the crash-types.

RQ3: Does a detailed comparison of stack trace help improve the grouping?

The third research question evaluates the two-level grouping approach presented in Section 4.2.4. We use the silhouette validation technique to evaluate the two-level grouping algorithm. The silhouette validation is a technique to measure the goodness of a grouping approach. Using silhouette validation, we compare the dissimilarity of a crash-report with other crash-reports from the same subgroup and the similarity of the crash-report with other subgroups in the same crash-type. For a crash-report i the silhouette value S(i) is defined in Equation (4-2).

$$\boldsymbol{S(i) = \frac{b(i) - a(i)}{max(b(i), a(i))}} \qquad \textbf{Equation (4-2)}$$

Where a(i) is average dissimilarity of crash-report i to all other crash-reports in the same subgroup and b(i) is the minimum of average dissimilarity of the crash-report i to the crash-reports in other subgroups in the same crash-type.

We compute the similarity (or dissimilarity) of two crash-reports by comparing the top ten frames of the stack traces from the crash-report, as discussed in Section 4.2.4. The average

silhouette value for all crash-reports is the silhouette value for the crash-type. The silhouette value has a range from -1 to 1, where the value -1 implies misclassified and a value close to 1 implies well clustered.

Furthermore, we assess the effectiveness of the two-level grouping approach to group the crash-reports triggered by the same bug. We select the crash-types for which at least one bug is fixed and the bug has the patch information. We apply our proposed grouping approach to the selected crash-types, and build subgroups. For each bug, we identify modules that are changed to fix the bug. We map this information to the stack traces contained in crash-reports from a subgroup. If a bug fix location appears in the stack trace of any of the crash-report from the subgroup, we link the bug with the subgroup. As a result, a subgroup can be linked to a single bug, to multiple bugs, or to no bug. It is desirable to have a subgroup linked with a single bug, since it suggests that crash-reports in the subgroup are triggered by the same bug.

We compute the accuracy of our grouping algorithm as defined in Equation (4-3). The accuracy metric assesses the ability of the approach to group the crash-reports triggered by the same bug. When a subgroup is linked to multiple bugs, it's likely that the crash-reports in the subgroup are triggered by multiple bugs. If the crash-reports in the subgroups are not linked with any bugs, such crash-reports are triggered by a bug which is not identified and not filed by the developers.

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{N(s)} + \boldsymbol{N(z)}}{\boldsymbol{N(s)} + \boldsymbol{N(z)} + \boldsymbol{N(m)}} \qquad \textbf{Equation (4-3)}$$

Where N(s) is the number of crash-reports in the subgroups linked to a single bug, N(z) is the number of crash-reports in the subgroups linked to no bug, and N(m) is the number of crash-reports in the subgroups linked to multiple bugs.

We compute the precision of a subgroup as defined in Equation (4-4). The precision of a subgroup measures the percentage of crash-reports in the subgroup triggered by the bug linked to the subgroup. If the faulty module, where the bug is fixed, appears in the stack trace of a crash-report, we assume that the crash-report is caused by the same bug.

$$\boldsymbol{Precision} = \frac{\boldsymbol{N(t)}}{\boldsymbol{N(t) + N(f)}} \qquad \textbf{Equation (4-4)}$$

Where N(t) is the number of crash-reports in a subgroup for which the faulty module appears in the stack trace; and N(f) is the number of crash-reports in a subgroup for which the faulty module does not appear in the stack trace.

4.4 Case Study Results

In this section, we present the results of our case study on the research questions and discuss our findings.

4.4.1 RQ1: Can stack traces in crash-reports help to locate bugs?

This research question investigates the use of the crash-reports for bug fixing. More specifically, we aim to assess if stack traces contained in the crash-reports are useful to fix bugs. We analyze all fixed bugs which have available patches, and extract the corresponding patches to identify modules that are changed to fix the bug. The patches are available for 231 bugs and can be mapped to source code. We map the bug fix locations to the stack traces of the crash-reports, as described in Section 4.3.1. We compute the percentages of bugs belonging to each of the following three categories: (1) bugs that are fixed in the linked stack traces; (2) bugs that are fixed in other stack traces; and (3) bugs that are fixed elsewhere. On average, 57% of bugs are fixed in the linked stack traces; 23% of the bugs are fixed in other stack traces; and the remaining

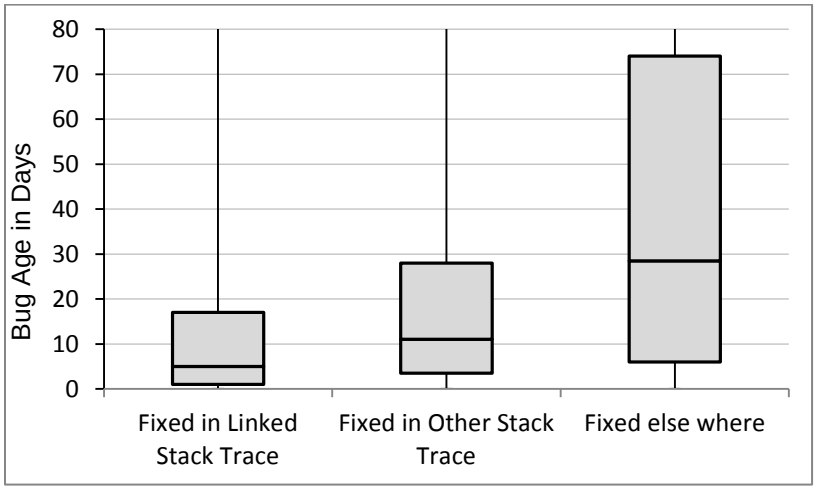

Figure 4-7: Boxplots comparing lifetimes of the bugs, based on bug fix locations

20% of bugs are fixed elsewhere. Figure 4-7 presents the boxplot of the lifetime of bugs in the three categories.

As shown in Figure 4-7, the bugs fixed in the linked stack traces are fixed quicker than the bugs classified in the other two categories. The mean and median values of the time to fix bugs in the linked stack traces category are 19 days and 5 days, respectively. The mean and median values are 23 days and 11 days, respectively for bugs fixed in other stack traces and 48 days and 29 days, respectively for the bugs that are fixed elsewhere.

We perform the Kruskal-Wallis rank sum test on the lifetimes of bugs from the three categories and obtain a statistically significant result (i.e., p-value is less than 0.01). Therefore, we reject hypotheses H_{01}. We can conclude that the lifetime of a bug is significantly shorter when the faulty module appears in the stack traces of the crash-reports. The lifetime of a bug can be further reduced when the stack traces containing the faulty modules are correctly linked to the bug.

It indicates that bugs fixed in the linked stack traces take shorter time to get fixed, since developers can locate the bugs easily by analyzing the failing stack traces of the linked crash-

reports. However, it is surprising that bugs fixed in other stack traces take shorter time than bugs fixed elsewhere. Since we sample only 100 crash-reports for each crash-type, the shorter bug fixing time observed for the bugs fixed in other stack traces indicates that there may be other crash-reports in the crash-types with stack traces containing the faulty module. Overall, our results suggest that in general for 57% to 80% of the bugs, stack traces in the crash-reports can help to locate the bugs. We answer positively our research question that the stack traces in crash-reports can help the localization and correction of bugs. Moreover, crash-reports triggered by the bug can be identified by analyzing the stack traces in the crash-reports.

4.4.2 RQ2: Does the grouping of crash-reports impacts bug fixing?

We observe in our data set that for some of the crash-type multiple bugs are filed and some bugs are linked to multiple crash-types. We assume if crashes triggered by multiple bugs are grouped together, this creates ambiguity for developers to analyze the crash-type. We also assume if a crash-type contains the reports triggered by a single bug, developers can fix bugs more efficiently. To verify our assumptions and answer the research question, we compare the bug fixing times for different bug categories based on bug crash-type relations.

4.4.2.1 Crash-types Linked to Multiple Bugs

Our data set contains 519 fixed bugs. 74% of the fixed bugs are uniquely linked to the corresponding crash-types. The remaining 26% of bugs are linked to the crash-types where other bugs are also linked to the same crash-type, it indicates that crash-reports triggered by different bugs are grouped together.

We compare the fixing time for the bugs which are uniquely linked to one or more crash-types (i.e., crash-report for each bugs are grouped separately) with the bugs which are collectively

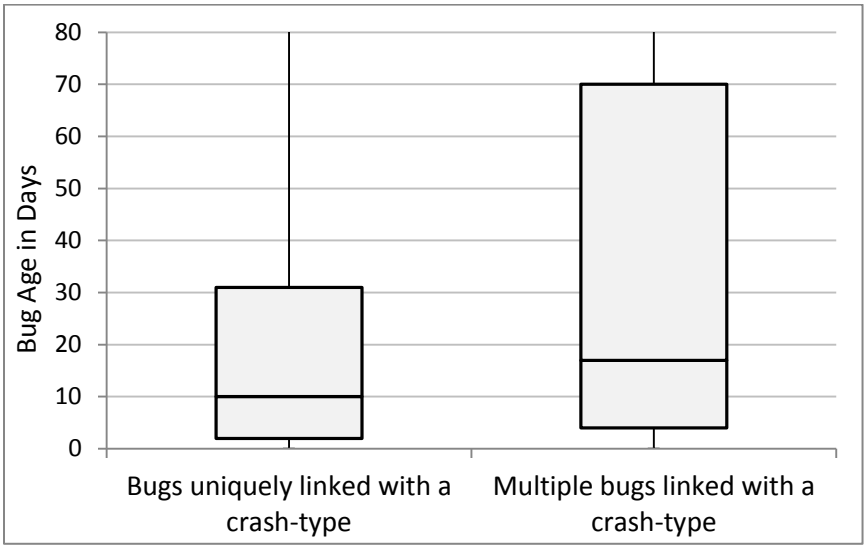

Figure 4-8: Comparing lifetimes of Bugs uniquely/collectively linked to a crash-type

linked to the same crash-type (i.e., crash-reports for each bug is mixed with crash-reports caused by other bugs).

Figure 4-8 shows the boxplot of bug fixing times for both cases. The mean and median values of the time to fix bugs uniquely linked with one or more crash-types are 26 days and 10 days, respectively. The mean and median values are 43 days and 17 days, respectively for bugs collectively linked to the same crash-type. If the crash-reports triggered by a bug are grouped separately, the bug takes on average 17 days lesser to be fixed than fixing the bugs for which crash-reports are grouped together. This finding validates our assumption that it is difficult to locate and fix the bug when crash-reports triggered by different bugs are grouped together. We perform a Wilcoxon rank sum to verify the statistically significance of this result and obtain a p-value of 0.04. Therefore, we reject H^1_{02}.

In summary, when multiple bugs are collectively linked to the same crash-type, it takes a longer time to have the bugs fixed than fixing the bugs that are uniquely linked to one or more

crash-type. We answer our research question positively: The grouping of crash-reports has an impact on bug fixing time.

4.4.2.2 Bugs Linked to Multiple Crash-types

In our data set, 384 fixed bugs are uniquely linked to one or multiple crash-types; 40% of the bugs are uniquely linked to multiple crash-types and the remaining 60% of bugs are uniquely linked to a single crash-type.

Figure 4-9 shows the boxplot of bug fixing times for both cases. The mean and median values of the time to fix bugs linked with single crash-types are 28 days and 13 days, respectively. The mean and median values are 22 days and 10 days, respectively for bugs linked to multiple crash-types. We observe that the bugs linked to multiple crash-types take on average 6 days less to be fixed than fixing the bugs linked to a single crash-type. It hints that the bug is assigned a high priority when a bug is linked with multiple crash-types. Moreover, when bugs are linked to multiple crash-types, the crash-types provide rich information on different scenarios of the bug occurrences. Thus, it helps developers better understand the issues.

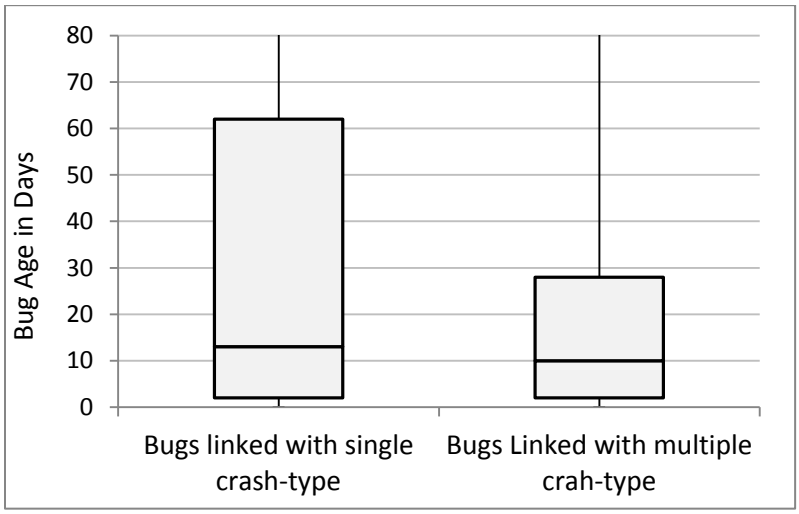

Figure 4-9: Comparing life times of bugs linked with single/multiple crash-types

However, the Wilcoxon rank sum test reveals no statistically significant difference between the lifetimes of the bugs linked to multiple crash-types and the lifetimes of the bugs linked to a single crash-type (i.e., p-value equal to 0.08).

4.4.2.3 Trace Diversity of Existing Crash-types

We analyze the current grouping approach of crash-reports from Socorro to understand the diversity of the stack traces contained in the crash-reports of a crash-type. As aforementioned, the existing approach groups crash-reports based on the top method signature of the failing stack trace. The stack traces are not identical for all the crash-reports of a crash-type. We quantify the diversity of the stack trace in crash-reports from a crash-type using the trace diversity as discussed in Section 3. We categorize the crash-types based on the number of bugs filed for each crash-type. For each category, we compute the average trace diversity of the crash-types. Table 4-2 lists the detailed results for the categories.

As shown in Table 4-2, if a single bug is filed for a crash-type, the crash-type has relatively lower trace diversity than the crash-types which has multiple bugs filed. We statistically verify the result as the Spearman's rank correlation value between the trace diversity values and the number of bugs linked to the crash-type is 0.95 (i.e., p-value equal to 4.96e-05). The result shows that higher trace diversity indicates that crash-reports in a crash-type are triggered by multiple bugs. The effectiveness of crash-report grouping approach can be improved by controlling the magnitude of the trace diversity value for grouping crash-reports together.

Table 4-2: The Average Trace Diversity Values for All Crash-Types

Number of bugs linked to each crash-type	Average Trace Diversity
1	4.82
2	5.81
3	5.88
4	6.67
5	8.22

4.4.3 RQ3: Does a detailed comparison of stack trace help improve the grouping?

We perform a case study to assess the effectiveness of the two-level grouping approach presented in Section 3. We select the 231 bugs from our data set which have patches available. The 231 bugs are linked to 277 crash-types which consist of 18,498 crash-reports. We apply the two-level grouping approach to regroup the crash-reports. We set the trace diversity threshold value to 5, since in the result of second research question we observe that the crash-types for which a single bug is filed have a trace diversity value close to 5. Table 4-3 lists the descriptive statistics of the data set used for our evaluation. Table 4-4 shows the result for evaluating the two-level grouping approach.

The average trace diversity of the subgroups created by using the two-level grouping approach is low, i.e., 3.8. We measure the goodness of our grouping by computing silhouette values. The average silhouette value for each crash-type is 0.81. A high value (i.e., 0.81) suggests a good clustering of crash-types.

For the subgroups, we compute the accuracy as using Equation (4-3). As shown in Table 4-4, the accuracy of the two-level grouping approach is 0.88. It shows that 88% of the newly created subgroups are linked to only one bug or no bug.

Table 4-3: Descriptive Statistics of Evaluation Data Set

	# of crash-type	**# of crash-reports**	**# of Bugs Linked**	**fix time (days)**	**Avg. *TD***
All crash-types	277	18498	231	***6540***	6.5
Crash-types linked to a single bug	225	14244	204	5212	4.6
Crash-types linked to multiple bugs	52	4254	27	1328	14.7

Avg. TD – Average Trace Diversity, Est. fix time – Estimated Bug Fixing Time

Table 4-4: Descriptive Statistics of Result

	# of subgroups	**# of crash-reports**	**# of Bugs Linked**	**Est. fix time (days)**	**Avg. TD**
All subgroups	941	18498	231	***6193***	3.8
subgroups linked to a single bug	512	10812	220	5720	3.6
subgroups linked to zero bug	297	5547	0	0	3.9
subgroups linked to multiple bugs	132	2139	11	473	4.3

Avg. TD – Average Trace Diversity, Est. fix time – Estimated Bug Fixing Time

We compute the precision for the 512 subgroups, which are linked to a single bug, using Equation (4-4). The average precision of the subgroups is 0.98, meaning that on average 98% of crash-reports in each subgroup are triggered by the same bug, which is linked to the subgroup.

Despite 88% of accuracy and 98% precision, one can question that the number of subgroups created are 3 times more than the number of crash-types. But our approach maintains the existing crash-types, so at the first level, the number of groups is the same as currently in Socorro. However, when developers analyze a crash-type, the subgroups provide more detailed information. If two subgroups are related to different bugs, the subgroups improve the bug fixing process by separating the crash-reports caused by each bug. Even if two subgroups are caused by

the same bug, both subgroups represent significantly different stack traces. As discussed in Section 4.4.2.2 when a developer selects one crash-report from each subgroup, the selected crash-reports provide better information instead of two randomly selected reports. The 512 accurately created subgroups are linked to 220 bugs, i.e., on average 2.3 subgroups are created for each bug.

4.4.4 Measuring the benefits of proposed grouping approach

To further assess the benefit of our proposed grouping approach, we compare the estimated bug fixing time when crash-types are divided in subgroups using the two-level grouping approach and the actual bug fixing time that we compute from the bug reports.

We hypothesize a scenario where developers were using the sub grouping since the first beta release. We estimate the bug fixing time for crash-type where multiple bugs are linked. We assume that in hypothesized scenario the fixing time for these bugs will be the same as the mean fixing time for the bugs which are uniquely mapped with a crash. The estimated fixing time is compared with the actual time to fix these bugs. Figure 4-10 shows the hypothesized scenario, for two crash-types. Bug 618118 and 619004 are linked to one crash-type, bug 614979 and 618899 are linked to another crash-type. For each crash-type other open bugs also exist. We hypothesize that if crash-reports caused by each bug are well organized in separate groups, the bug fixing time will be less.

The collective time for fixing all the 231 bugs is 6540 days. As discussed in Section 5.B, on average the bug fixing time for a bug uniquely linked with a crash-type is 26 days; and the bug fixing time for the bugs collectively linked with a crash- type is 43 days. Using these average values of bug fixing time, we estimate the collective bug fixing time for the 231 bugs when developers use the proposed two-level grouping approach.

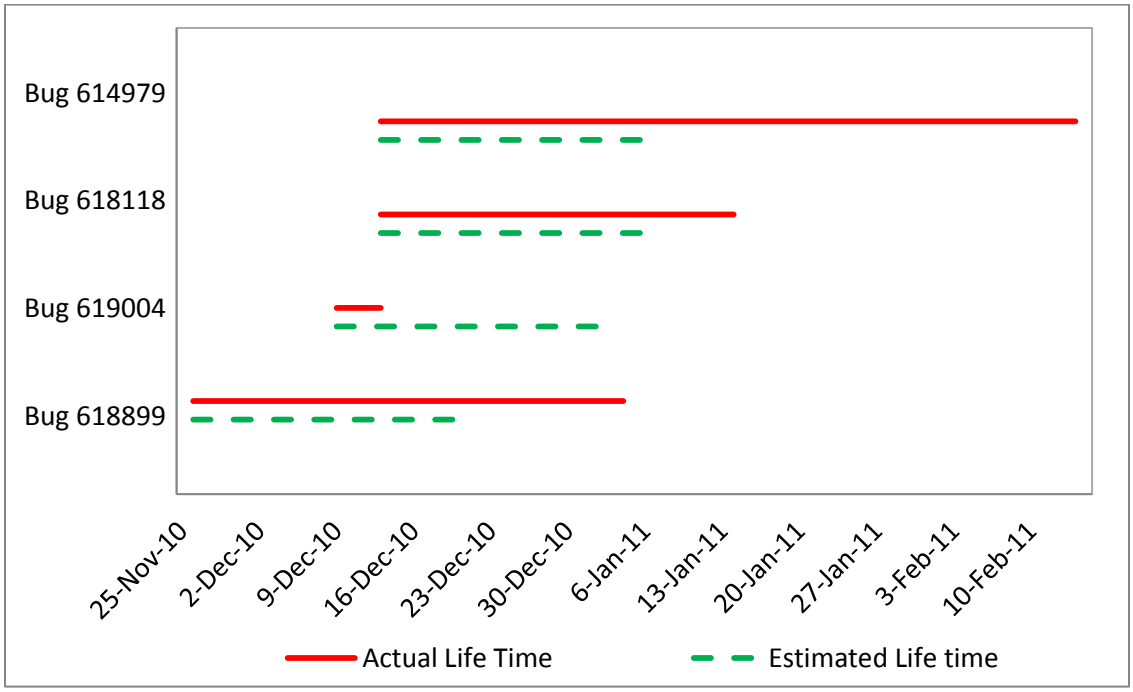

Figure 4-10: Comparing the Actual life time of bugs with the Estimated Life time

The estimated collective time to fix the 231 bugs is 6193 days (i.e., 220*26 + 11*43), in comparison to the actual time of 6540 days. We can conclude that the two-level grouping approach can reduce the bug fixing time by 5.3%.

4.5 Threats to Validity

This section discusses the threats to validity of the case study we discussed in this chapter, following the guidelines for case study research [51].

Construct validity threats concern the relation between theory and observation. In this study, the construct validity threats are mainly due to measurement errors. We extract stack trace and bugs information by parsing the html and xml files and map the bug fix location to the stack traces by applying string matching. The techniques we use are similar to the techniques used by previous studies.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. We used non-parametric tests that do not require making assumptions about the data set distribution.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. Moreover, both the Socorro crash server and Bugzilla are available publicly, to obtain the same data for the same releases of Firefox.

External validity threats concern the possibility to generalize our results. Nevertheless, our study is limited to 10 releases of Firefox, further studies with different systems and different automatic crash-reporting systems are desirable to make our findings more generic.

4.6 Summary

It becomes the norm to embed automatic collection of crash-reports in software systems. However, limited studies investigated the use of the collected crash-reports by developers in their maintenance activities. In this work, we studied the use of field crash-reports during the beta testing Firefox version 4. We summarize the key findings of our study as follows:

1. We analyze the use of failing stack traces in crash-reports by developers when performing bug fixing activities and find that 80% of bugs are fixed in modules appearing in failing stack traces of crash-reports. Therefore the stack traces in crash-report could be used to identify the crash-report triggered by the same bug.
2. We investigate the crash-report grouping approach used by Mozilla. We observe that in average it takes 17 days longer to fix the bugs when crash-reports triggered by multiple bugs are grouped together in comparison to fixing the bugs for which the crash-reports are grouped separately.

3. We identify the limitation of the current grouping approach and propose a Trace Diversity metric which could help improve the efficiency of groupings. The result shows that if the trace diversity of a crash-type is greater than 5, the crash-type is likely to contain crash-reports triggered by multiple bugs.
4. We suggest a detailed comparison of stack traces to group the crash-reports. This limits the trace diversity of a crash-report group and it is easier for developers to locate and fix bugs. Our grouping approach limits the trace diversity of a subgroup to less than 5 and 88% of the subgroups contain crash-reports triggered by a single bug. This improvement to the existing Mozilla crash reporting system can help to reduce the bug fixing time by more than 5%.

Chapter 5

Conclusion and Future Work

Identifying commit dependencies in selective code integration and identifying similar crash reports are two important steps in the software development process. For each of the two challenges we use the information available in software repositories and present an automated approach to assist the software developers and software maintainers. In this chapter, we outline the contributions of this thesis and discuss future work.

5.1 Thesis Contributions

This thesis makes the following contributions:

1. ***Propose dissimilarity metrics to identify dependencies among commits.*** Based on the commit attributes, we defined four dissimilarity metrics (File Dependency Distance, File Association Distance, Developer Dissimilarity Distance and Functional Change Dependency Distance). We show that these dissimilarity metrics can be applied to identify the dependencies among the commits.
2. ***Define an approach to group the dependent commits.*** To avoid the missing commit dependencies during the selective code integration, we present an approach to group the dependent commits. The approach learns the dissimilarity levels among the dependent commits using the historical data, and assists the developers to identify the commit dependencies for newly added commits. We present two use cases of our approach: in the first use case, a developer can link a newly added commit to the existing dependent commits; and in the second use case, an integrator can identify all the commits related to a functional change. Through an empirical case study we show that our approach to

group the dependent commits can reduce the integration failures caused by missing commit dependencies by up to 94%.

3. ***Investigate the impact of crash report grouping on software maintenance.*** We show that it takes significantly longer time to fix the bugs when crash-reports triggered by multiple bugs are grouped together in comparison to fixing the bugs for which the crash-reports are grouped separately. We also show that when crash reports caused by the same bug but under different use cases are grouped separately, it helps developers to locate the bug more easily in comparison to when all crash reports caused by the same bug but under different use cases are grouped together.
4. ***Define Trace Diversity metric to evaluate the quality of a crash report group.*** We defined the Trace Diversity metric, based on the dissimilarity between the stack traces of the crash reports in a group. The metric can be applied to verify if a crash report group contains the crash reports caused by single bug or the crash reports caused by multiple bugs. The trace diversity metric of a group is a measure of the time required to locate and fix the bugs related to a crash report group.
5. ***Define an approach to group the field crash reports.*** We defined a two level grouping approach to group the crash reports, based on the detailed comparison of the stack traces in the crash reports. The approach groups a continuous incoming stream of crash reports that allows developers to work on the crash report groups while new reports are being added to the groups. We defined the representative stack trace for a group of crash reports. The representative stack trace optimizes the approach to handle a large number of crash reports. Through an empirical case study we show that our approach to group the

field crash reports can reduce the overall bug fix time for bugs causing the field crashes by up to 5%.

5.2 Future Work

The two approaches we defined in this thesis assist the developers in the software development process, but neither of the approach can completely replace the developers for the respective tasks.

For the selective code integration, we present two variants of our approach, one to assist the developer when he submits the code changes and another variant to assist the integrator when he integrates a functional change. We evaluated both variants of our approach individually. We plan to evaluate the benefits of combining both variants of our approach. Also, since our grouping approaches learn the dissimilarity levels among dependent commits from a previous version, they cannot be applied on the first version of a software system. In future work, we plan to explore the possibility of learning dissimilarity levels across projects i.e., if dissimilarity levels learned from one project, can be applied to other projects of the same team. The case study for selective code integration was conducted on a proprietary system. We also plan to conduct a similar study on an open source system, which will help us to generalize the results.

For crash report grouping, we create a representative trace to identify the crash reports caused by the same bug. In a way, the representative stack trace reflects the stack trace pattern of the bug. In the future, we plan to optimize the representative trace to further improve the crash report grouping. The representative trace can also be used for bug correlation and bug localization.

Bibliography

[1] "IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries," *IEEE Std 610*, pp. 1, 1991.

[2] S. Diehl, H. Gall and A. E. Hassan, "Guest editors introduction: special issue on mining software repositories," *Empirical Software Engineering*, vol. 14, pp. 257-261, 2009.

[3] W. W. F. Tichy, "Rcs — a system for version control," *Software, Practice & Experience*, vol. 15, pp. 637-654, 1985.

[4] A. Lie, T. Didriksen, R. Conradi, E. Karlsson, S. Hallsteinsen and P. Holager, "Change oriented versioning," in *Proceedings of the 2nd European Software Engineering (ESEC '89)*, London, UK, pp. 191-202, 1989.

[5] N. N. Ohlsson, "Experiences of Fault Data in a Large Software System," *Failure & Lessons Learned in Information Technology Management*, vol. 2, Cognizant Communication Corporation, pp. 163-171, 1998.

[6] J. Bosch, "Product-line architectures in industry: A case study," in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, pp. 544-554, 1999.

[7] L. Wingred, "How software evolves," in *Practical Perforce, Channeling the Flow of Change in Software Development Collaboration*, Jonathan Gennick, Ed. O'Reilly Media, pp. 176-197, 2005.

[8] J. van Gurp, C. Prehofer and J. Bosch, "Comparing practices for reuse in integration-oriented software product lines and large open source software projects," *Software: Practice and Experience*, vol. 40, John Wiley & Sons Inc., pp. 285-312, 2010.

[9] P. P. Bourque, "The guide to the Software Engineering Body of Knowledge," *IEEE Software*, vol. 16, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 35-44, 1999.

[10] L. Thomsan, "Socorro: Mozilla's Crash Reporting System," http://blog.mozilla.org/webdev/category/soccoro/, Last accessed 2012/07/19.

[11] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle and G. Hunt, "Debugging in the (very) large: Ten years of implementation and experience," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, Montana, USA, pp. 103-116, 2009.

[12] T. Dhaliwal, F. Khomh and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*, Williamsburg, VA, USA, pp. 333-342, 2011.

[13] M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" *J.Syst.Softw.*, vol. 43, pp. 19-27, oct, 1998.

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

[15] K. C. Kang, Jaejoon Lee and P. Donohoe, "Feature-oriented product line engineering," *Software, IEEE*, vol. 19, pp. 58-62, 2002.

[16] L. F. Capretz, F. Ahmed, S. Al-Maati and Z. A. Aghbari, "COTS-based software product line development," *International Journal of Web Information Systems*, vol. 4, Emerald Publishing Ltd., pp. 165-180, 2008.

[17] I. Groher and M. Voelter, "Aspect-Oriented Model-Driven Software Product Line Engineering" *Transactions on aspect-oriented software development VI*, Springer-Verlag, pp. 111-152, 2009.

[18] C. Walrad and D. Strom, "The importance of branching models in SCM," *Computer*, vol. 35, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 31-38, sep, 2002.

[19] B. O'Sullivan, "Making sense of revision-control systems," *Commun ACM*, vol. 52, ACM, New York, NY, USA , pp. 56-62, sep, 2009.

[20] H. Gall, K. Hajek and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the 11th International Conference on Software Maintenance (ICSM '98)*, Bethesda, MD, USA, pp. 190-198, 1998.

[21] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, pp. 429-445, june, 2005.

[22] A. T. T. Ying, G. C. Murphy, R. Ng and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, pp. 574-586, sept., 2004.

[23] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang and A. Michail, "CVSSearch: Searching through source code using CVS comments," in *Proceedings of the IEEE International Conference Software Maintenance (ICSM '01)*, Florence, Italy, pp. 364-374, 2001.

[24] D. L. Atkins, "Version sensitive editing: Change history as a programming tool," in *ECOOP '98: Proceedings of the SCM-8 Symposium on System Configuration Management*, pp. 146-157, 1998.

[25] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, Portland, OR, USA, pp. 408-418, 2003.

[26] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, Ischia, Italy, pp. 45-53, 1998.

[27] A. Mockus and D. M. Weiss, "Globalization by chunking: a quantitative approach," *Software, IEEE*, vol. 18, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 30-37, 2001.

[28] M. Kamkar, "An overview and comparative classification of program slicing techniques," *J. Syst. Software*, vol. 31, Elsevier Science Inc., New York, NY, USA, pp. 197-214, 1995.

[29] S. Artzi, S. Kim and M. D. Ernst, "ReCrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, Paphos, Cypress, pp. 542-565, 2008.

[30] A. Ganapathi, V. Ganapathi and D. Patterson, "Windows XP kernel crash analysis," in *In Proceedings of the 2006 Large Installation System Administration Conference (LISA '06)*, Washington, D.C., USA, pp. 12-22, 2006.

[31] Dongsun Kim, Xinming Wang, Sunghun Kim, A. Zeller, S. C. Cheung and Sooyong Park, "Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts," *IEEE Transactions on Software Engineering*, vol. 37, pp. 430-447, 2011.

[32] Y. Dang, R. Wu, H. Zhang, D. Zhang and P. Nobel, "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity," in *Proceedings of the 2012 International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland, pp. 1084-1093, 2012.

[33] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood and L. Mignet, "Automatically identifying known software problems," in *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, Istanbul, Turkey, pp. 433-441, 2007.

[34] C. Liu and J. Han, "Failure proximity: A fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, Oregon, USA, pp. 46-56, 2006.

[35] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu and G. Loihle, "Finding similar failures using callstack similarity," in *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, San Diego, California, 2008.

[36] M. Brodie, S. Ma, L. Rachevsky and J. Champlin, "Automated Problem Determination Using Call-Stack Matching," *Journal of Network and Systems Management*, vol. 13, Springer, New York, USA, pp. 219-237, 2005.

[37] G. Lohman, J. Champlin and P. Sohn, "Quickly finding known software problems via automated symptom matching," in *Proceedings of the Second International Conference on Automatic Computing*, Turin, Italy, pp. 101-110, 2005.

[38] S. Kim, T. Zimmermann and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, Hong Kong, China, pp. 486-493, 2011.

[39] A. Schroter, N. Bettenburg and R. Premraj, "Do stack traces help developers fix bugs?" in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR '10)*, Cape Town, South Africa, pp. 118-121, 2010.

[40] B. Chan, Y. Zou, A. E. Hassan and A. Sinha, "Visualizing the results of field testing," in *Proceedings of the IEEE 18th International Conference on Program Comprehension (ICPC '10)*, Braga, Minho, Portugal, pp. 114-123, 2010.

[41] B. Liblit, M. Naik, A. X. Zheng, A. Aiken and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, pp. 15-26, 2005.

[42] Christopher D. Manning, Prabhakar Raghavan,Hinrich Schütze, "Flat clustering," in *Introduction to Information Retrieval*, Cambridge University Press, pp. 321-343, 2008.

[43] A. Ahmad and L. Dey, "A k-mean clustering algorithm for mixed numeric and categorical data," *Data Knowl. Eng.*, vol. 63, Elsevier Science Publishers B. V., Amsterdam, Netherlands, pp. 503-527, 11, 2007.

[44] A. Likas, N. Vlassis and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern Recognition*, vol. 36, pp. 451-461, 2, 2003.

[45] P. S. Bradley and U. M. Fayyad, "Refining initial points for K-means clustering," in *Proceedings of the Fifteenth International Conference on Machine Learning*, Madison, WI, USA, pp. 91-99, 1998.

[46] M. E. Hohn, "Binary coefficients: A theoretical and empirical study," *Mathematical Geology*, vol. 8, Springer Netherlands, pp. 137-150, 1976.

[47] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall, 2007.

[48] R. Robbes, D. Pollet and M. Lanza, "Logical coupling based on fine-grained change information," in *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, Antwerp, Belgium, pp. 42-46, 2008.

[49] J. Guilford, "The phi coefficient and chi square as indices of item validity," *Psychometrika*, vol. 6, pp. 11-19, 1941.

[50] P. J. and Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53-65, 1987.

[51] R. K. Yin, *Case Study Research: Design and Methods - Third Edition.* SAGE Publications, 2002.

[52] J. B. Kruskal, "An Overview of Sequence Comparison: Time Warps, String Edits, and Macromolecules," *SIAM Rev*, vol. 25, pp. 201-237, 1983.