MASTER'S THESIS | LUND UNIVERSITY 2015

Fault Localization Automation using Code Coverage Data and Diff Utilities

Jonas Svalin, Axel Hildingson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-22

Fault Localization Automation using Code Coverage Data and Diff Utilities

(Master's Thesis)

Jonas Svalin
jonas.svalin@gmail.com

Axel Hildingson
a.hildingson@gmail.com

June 15, 2015

Master's thesis work carried out at QlikTech International AB, Lund, Sweden.

Supervisors: Lars Andersson, lars.andersson@qlik.com
Emilie Engström, emelie.engstrom@cs.lth.se

Examiner: Per Runeson, per.runeson@cs.lth.se

Abstract

Performing fault localization is one of the most time consuming and expensive aspects of software development today. In large software systems the manual fault localization techniques quickly become unmanageable and as such there is a growing need for automated solutions. In this master's thesis we propose and evaluate a new method for fault localization based on comparing code coverage and diff utility data. The implemented tool is called FLAC which consists of both a Java application that performs the necessary functionality and a Qlik Sense application designed for visualization purposes. The FLAC prototype is restricted to JavaScripts but the method can be applied to all types of software development projects that employ version control and have the ability to gather code coverage data from test executions. Our method is evaluated by applying it to otherwise stable software that has been injected with faults, analyzing efficiency and performance information and also through interviews. Our results indicate that this method can significantly increase fault localization efficiency.

Keywords: Fault Localization, Software Testing, Code Coverage, Version Control

Acknowledgements

We would like to thank Lars Andersson for inspiring this master's thesis. Lars has been essential in guiding the implementation process of FLAC due to his immense knowledge of software development and the operational procedures at Qlik. Furthermore we express our gratitude to Emelie Engström for her help with the methodology of performing a master's thesis. Her extensive feedback in the writing this report has also been crucial. We would also like to thank Tomas Bylander for supporting us in our daily activities at Qlik and our examiner Per Runeson for showing interest in this master's thesis. Lastly we would like to thank all the participants of the survey who took the time to help us evaluate FLAC.

Contents

1 Introduction **7**
1.1 Background . . . 7
1.2 Research Questions . . . 8
1.3 Outline of the Report . . . 8
1.4 Contribution Statement . . . 9

2 Background **11**
2.1 Software Testing . . . 11
2.2 Code Coverage . . . 12
2.3 Version Control . . . 12
2.4 Fault Localization . . . 13
2.5 Application . . . 14

3 Case Description **15**
3.1 Qlik . . . 15
3.2 Software Configuration Management . . . 15
3.3 Automated Testing Framework . . . 16
3.4 Fault Localization Issues . . . 17

4 Related Work **19**

5 Design Methodology **23**
5.1 Investigation of Needs . . . 23
5.2 Assessment of Related Work . . . 24
5.3 Solution Idea . . . 24
5.4 Prototype Development . . . 25

6 FLAC **31**

7 Evaluation Methodology **37**
7.1 Functionality 37
7.2 Efficiency 39
7.3 Usability 40

8 Evaluation Results **41**
8.1 Functionality 41
8.2 Efficiency 42
8.3 Usability 43

9 Discussion **45**
9.1 Discussion of Results 45
9.2 Known Limitations 46
9.3 Future Work 47

10 Conclusion **49**

Bibliography **51**

Appendix A Functionality Evaluation Material **57**

Appendix B Interview Questions **61**

Appendix C Interview Results **65**

Appendix D User Guide **69**

Chapter 1
Introduction

Fault localization is the process of locating the fault responsible for making a software system behave erroneously. This is a process that contrary to testing is still mostly performed in a manual manner at many software companies and is thus an expensive and time-consuming task. The following quote serves as the introduction to a paper written by Collofello et al in 1987 regarding fault localization automation and surprisingly, this fact is still very true today [19] [21] [29].

> *"Software development is a complex and error prone process. Although hardware costs during the last 30 years have consistently dropped, software costs have continued to climb. One of the significant factors in these increased costs is the expense incurred performing software fault (error) localization and repair. As the complexity and size of software systems continues to increase dramatically, emphasis is needed on developing automated methods to help perform fault localization and repair activities."* [18]

In this master's thesis we have worked with a company called Qlik to develop an automated fault localization tool called FLAC. In this chapter, we provide a brief introduction to the issue of fault localization at Qlik and how we intend to solve it.

1.1 Background

Qlik is a software development company operating in the area of Business Intelligence. They produce browser based applications for handling and analysing large amounts of data in order to make informed business decisions. During recent years, Qlik has expanded their software development and as a result of this they have adopted a wide array of automated methods of handling software development in an effort to increase productivity. These include both testing and clustering of test failures, the latter courtesy of a system created at

Qlik in 2014 called NIOCAT [28]. While the test execution is essentially completely automated and the introduction of NIOCAT has helped to automate the process of test analysis, Qlik still have issues with the time consuming and tedious aspect of fault localization.

Since fault localization is such a time consuming part of software development and debugging, Qlik has a great need for an automated solution that could point developers in the right direction when attempting to investigate the fault which causes a test to fail. Several automated fault localization methods have been proposed and implemented in the past (see Chapter 4), but most of these depend on several tests being executed and failing due to the same fault. In this report we aim to investigate a new approach that can be scaled down to analyzing individual test cases if necessary and also make better use of the dynamic nature of software development, contrary to the statistical approaches that are currently offered.

The way we intend to find code deemed suspicious, i.e. code likely to contain the fault, is by monitoring code coverage for the test suite under investigation and comparing the result with recent deliveries to the version control tool.

Theorem 1. *Let S_i denote a given statement in the code and ζ the set of statements executed by a given test case. Furthermore, let β denote the set of statements added or modified in a time period τ. For a test case T_j that succesfully executed before τ but failed after, the statement S_x responsible for the failure is likely to be found in the set $\Phi = \zeta \cap \beta$.*

In other words we look in the intersection of *what has been executed* during the test that failed with *what has changed* since it passed, see Figure 1.1 for a visual representation of this concept. In Qlik's case, up to a thousand files can undergo changes between builds of the system and as such narrowing the search space down to a handful of primary suspects can potentially save a lot of time and effort.

1.2 Research Questions

Based on the issue we are attempting to solve and the solution that has been proposed, we have determined three questions regarding a potential automated fault localization tool (AFLT) that will serve as the basis for the research performed in this master's thesis:

- How can an AFLT be implemented to find potential fault suspects based on code coverage and file change data?
- How can the efficiency of an AFLT be quantified and measured?
- How can the results from an AFLT be displayed in a manner that is intuitive and easy to understand for developers and testers?

1.3 Outline of the Report

This master's thesis report is divided into ten chapters. Chapter 2 gives a background to the concepts necessary to fully grasp the work that has been done and Chapter 3 gives a context to where and why the work was done. Related Work describes other research that

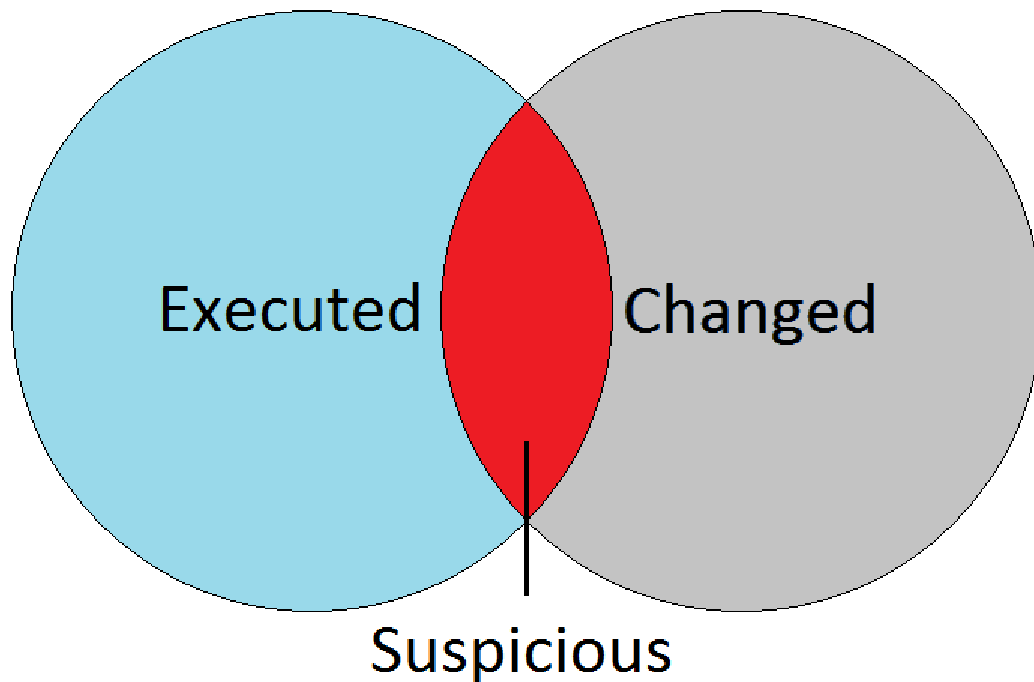


Figure 1.1: The principle behind how FLAC finds suspicious code.

has been done in the field of automated fault localization. Chapter 5 includes a solution idea for a new approach to automated fault localization and details of the process we went through in developing it. Chapter 6 describes the final prototype which was evaluated in Chapter 7. The results of the evaluation is presented in Chapter 8 and the final two chapters presents points of discussion and conclusion respectively.

1.4 Contribution Statement

The steps taken during the process of this master's thesis can be seen in Figure 1.2. This only provides a broad overview however, the steps were performed iteratively and not in a waterfall manner. We worked tightly throughout the entire process and all decisions regarding implementation and the structure of this report were taken together.

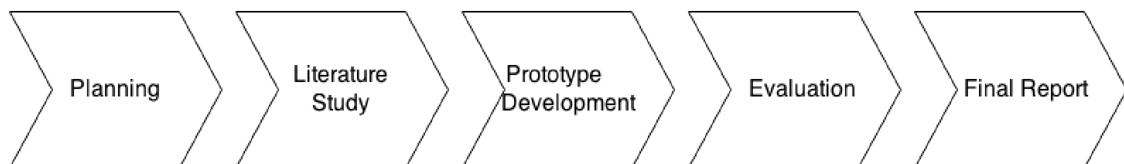


Figure 1.2: Overview of the master's thesis process.

The process started of with a period of planning and study of related work where both of us worked in parallel. This period ended when we started to develop prototypes and getting familiar with the testing framework at Qlik. During this period the first draft of

the report was written where we summarized the results of the literature study and the prototype development iterations.

We implemented the Java application together, which included all the steps defined in Figure 6.2. The Qlik Sense application was developed primarily by Axel. During the evaluation process Jonas was responsible for investigating the hit rate while Axel was responsible for investigating the efficiency. The surveys were all conducted and summarized together. The creation of the drafts for all sections of this report were divided evenly, but Jonas performed the majority of the proof-reading and writing of revisions. Axel developed the majority of the tables while Jonas made the majority of the figures.

Chapter 2
Background

In this chapter we summarize a few key areas in software development that will be necessary in order to fully grasp the work we have done. If these topics are already clear to you, feel free to go directly to Section 2.5 where we summarize how these items relate to this master's thesis.

2.1 Software Testing

Testing is an essential part of software development and is described by Ilene Burnstein as follows:

> *"Software testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality."* [17, p. 27]

Software testing is often divided into two distinct types, Black Box Testing (BBT) and White Box Testing (WBT). In BBT, it is assumed that no knowledge exists of the inner structure of a system or a component, and it is tested using a set of defined preconditions, inputs and outputs. Given a certain state of the system, the input might be 2+3 and the output is expected to be 5. A BBT verifies this and the test case passes or fails based on a comparison between the expected output and the actual output. In contrast to BBT we have WBT, where it is instead assumed that we have access to actual code, allowing us to perform specific tests on inner functions and statements. Similar to BBT, we have a set of preconditions, input and outputs and tests are performed on the inner structure of the system or component to verify that it behaves as expected [17, p. 64].

These principles are applied at different levels of testing, commonly separated into the following four categories: Unit Testing, Integration Testing, System Testing and Acceptance Testing [17, p. 133-134]. White box is mostly applied to unit tests, where small parts of the system are tested independently. In integration testing, groups of components

are combined to verify that the interfacing between them works as intended. Finally, tests are performed on the system as a whole, before it is sent off for acceptance testing, i.e. making sure that the final product meets the requirements defined by the customer in the requirements specification. These latter steps of testing are dominated by BBT.

Software development is an iterative process where functionality and bug fixes are introduced continuously throughout the development process. As such it is of great importance to be able to easily verify that the introduction of a new feature or removal of a bug has not introduced any new issues into the system. These tests are commonly refereed to as regression tests [17, p. 176]. Performing these tests manually can be time consuming, tedious and expensive and for these reasons many companies are becoming increasingly dependent on automated testing. In an automated testing environment a separate set of software is used to perform and evaluate certain aspects of the software under testing [17, p. 483]. Regression tests are ideal for automation, as they are performed very often and in the same manner every time. A plethora of tools exist to assist developers and testers from various environments, programming languages and testing frameworks in performing automated tests [10][11][12].

2.2 Code Coverage

Traditionally code coverage has been used to verify how much of the source code has been covered during the execution of a test suite [17]. In the most simple type of code coverage, this is done by marking every line that has been touched and calculating the ratio between touched and total lines of code in the system, according to Equation 2.1. Here, CV_s denotes the total coverage percentage, S_t denotes the number of executed statements and finally S_p denotes the total number of statements.

$$CV_s = \frac{S_t}{S_p} \times 100 \tag{2.1}$$

This ratio gives an indication of how much of the code is being executed in a given test suite. If a test suite only covers 70% of the code, we can conclude that a third of the source code is never tested and thus have a high probability of containing undiscovered faults. However, 100% coverage and no failures does not prove that the code is free from faults. Even though all the tests in a test suite pass while simultaneously covering each and every line of code, that does not mean that the code is flawless as it might behave improperly for other inputs than the ones used by the given test suite. As such, code coverage percentages tend to be used only as a compliment to other information. A variety of tools are available to automatically perform this coverage, depending on the programming language and environment used [3][6].

2.3 Version Control

Version control is a part of Software Configuration Management (SCM) and is defined as the management of changes that occur in Configuration Items (CI), i.e. items that have been placed under version control [14]. A CI can be any type of document, file of code,

software environment or hardware depending on the context. In this report however, we will be focusing on the management of textual files of code. In a practical sense, version control of software simply means saving a copy of the software at a given state and labeling it sequentially as for example "version 1", "version 2" and so on. This means that the evolution of the software can be traced and also allows a developer to revert to any given state of the software, in case any undesirable changes have been introduced. For a developer, keeping files of code under version control is an excellent way of ensuring stability during development and it is also an essential tool when doing collaborative work, as it allows several developers to work on the same system (and even on the same files to a certain extent) at the same time. A great number of version control tools are used today and many of them will provide a remote repository where the software project and all its history is located [2][4][7].

An example of a version control tool is Git, an open source tool which was originally created to manage the development of the Linux operating system. Git has since evolved into becoming one of the most popular version control tools on the market, being used by everything from amateur developers to major software companies. Git in itself is a textual version control tool used from the command prompt, but several providers offer online repositories and accompanying graphical interfaces, an example being Atlassians' Stash [13]. Most version control tools will offer a difference operator (usually refereed to as diff), a function that allows you to compare any two versions of a file and extracting the changes that have been introduced between them [14]. This functionality is essential for the development of this thesis and we will return to it later in the report.

2.4 Fault Localization

When performing tests on a system, be they manual or automatic, a final result is always achieved which tells you whether the tests passed or fail. If one or more test cases failed, the so-called debugging process is initiated. Debugging is the combined process of finding the fault that caused a test failure to surface and correcting it [24].

When debugging code, there are different techniques used to minimize the time and effort needed to find the fault responsible. The simplest and most popular technique to get information from the system is to add print statements in the code to see what value(s) one or more variables have during a specific part of the execution [24, p. 3]. This technique is intuitively simple but it is also time consuming and forces you to manipulate the code which can lead to changes in execution behaviour. Most of the development tools used today have a debugging option that allows the user to get the same (and more) information without manipulating the code. Most of these use breakpoints that stops the execution of the program and displays the information that is stored in the memory at the breakpoint, such as value, content or size of all variables.

These tools also offer the possibility to jump and execute the next line to see the changes that occur along with many other more advanced features. Stepping through the code in a systematic way to reveal what influence statements or function calls have on variables is the classical way of debugging code [24, p. 3]. Several general principles exist, such as dividing the code into smaller sections (such as functions or classes) and verifying that these return the expected values in order to more easily rule out certain sections from the

search space. For larger and more complex systems it might be difficult to know how the dependencies within the system behaves or even what value a variable should have at any point in execution. At this point, even an advanced debugging tool might not offer enough support and more sophisticated automated tools are necessary.

Aside from the code and execution inspection based techniques described above, other manual methods of narrowing down the search are frequently used. For example, if you know that the code worked as intended an hour, a day or a week ago, it makes sense to look at what has changed since then. Since most developers use a version control tool (see Section 2.3) and these tools offer a diff utility, the developer can diff between the previous working state of the code and the current one to see what has changed since then. Such an analysis can often yield valuable information about where a fault might have been introduced, but similar to the debugging case, it can quickly become unmanageable as the size and complexity of the system grows.

2.5 Application

As described in Section 2.2, the application of code coverage has traditionally been to determine the percentage of code that is covered by a test suite. Here however, we use code coverage in a different manner. By gathering the code coverage information from a specific test, along with the information from the diff utility, it is possible to compare the code that has been executed in a given test with the code that has recently changed. Since it is very likely that changes or new introductions in code that has been successfully executed by tests in the past is the culprit when they suddenly fail, this comparison can give an indication of where in the code you should begin the process of fault localization when investigating test failures. Similarly, if you can determine that none of the executed code has changed since the last successful test execution, this is an indication that the fault is not located in the code at all and is instead caused by an external issue.

In this thesis we apply this theory by implementing code coverage in Qliks testing framework and developing a tool that automatically performs the comparison. The tool is evaluated both by applying it to deliberately introduced issues to verify functionality and also broadly by testers and developers in order to determine usability and expected usefulness.

Chapter 3
Case Description

We have conducted a case study at Qlik from which we aim to give a context for the work done in this master's thesis. In this chapter we provide an overview of what type of company Qlik is and what types of services they provide. We also detail their software configuration management and testing activities as well as their fault localization issues.

3.1 Qlik

Qlik is a company founded in Sweden in 1993 operating in the area of Business Intelligence (BI) [8]. BI refers to techniques and tools used to analyze large amounts of raw data and presenting it in a way that can be readily used for performing business analysis [23, p. 6]. Qlik is considered a market leader in Business Intelligence and have been for many years [26] .

Qlik offers two products for data handling and analysis - QlikView and Qlik Sense - with the latter being the newest addition to the library. Qlik Sense is a data visualization application that can run in a variety of browsers, using a simple drag-and-drop interface to easily handle large quantities of data and presenting them in useful and intuitive manner [9]. The front end of Qlik Sense is created using JavaScripts, while the back-end is written in C, C++ and C#. See Figure 3.1 for an example of how the application can appear when in use.

3.2 Software Configuration Management

The development and maintenance of Qlik Sense is performed continuously by several teams. In order to maintain stability, Qlik Sense is developed using a branching strategy where all new features and other changes are performed on separate branches and merged into the main line upon completion. Qlik employs a continuous integration system called

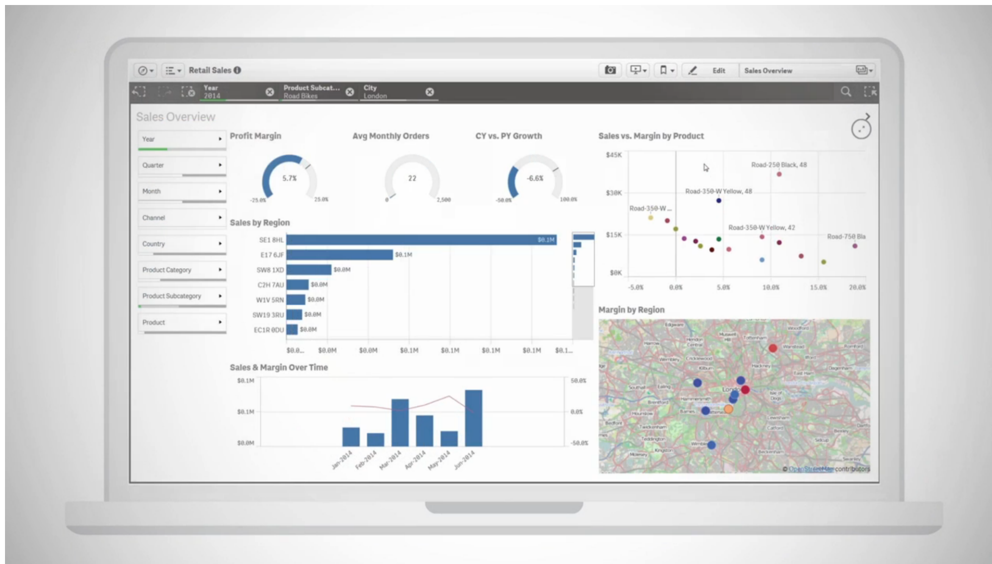


Figure 3.1: Example screen using Qlik Sense [9].

Bamboo that continuously builds versions of Qlik Sense from each active branch. Bamboo automatically builds each component of Qlik Sense, executes a set of test suites and reports their results. Teams are only allowed to merge their new features and changes into the main line if the test suites pass without failure.

Figure 3.2 shows an example of the branching pattern. In this example, Feature A is branched out from the main line upon deciding that a new feature needs to implemented. Feature B is branched out before A is integrated into the main line and thus, a forward merge from the main line to B is required when A is completed. When B is finally completed, it is integrated into the main line, assuming that all tests pass. If failures occur on the main line, integration from branches are halted immediately while the failures are investigated. While this ensures a largely stable main line, it inevitably also leads to decreased productivity because integrations are delayed due to main line restrictions. Along with reporting the test results, Bamboo will also provide a variety of test artifacts such as videos, logs and other items that can be useful when investigating test failures.

3.3 Automated Testing Framework

During the inception of Qlik Sense, an automated testing framework was created to accompany the development. This was done in order to ensure stability by performing regular regression tests. This framework has been dubbed Horsie and is built around Selenium, an automated browser tester [12]. Selenium allows you to create scripts that perform a set of instructions on a web application and verify that the state and output of the application is as expected. The automated testing framework has been of tremendous benefit in maintaining software quality and stability at Qlik, but it has also led to an overflow in information.

Due to the sheer number of active teams and branches working separately on Qlik Sense, a large number of simultaneous builds and test runs in Bamboo also give rise to a large number of test failures. Qlik has pursued these issues by developing internal sys-

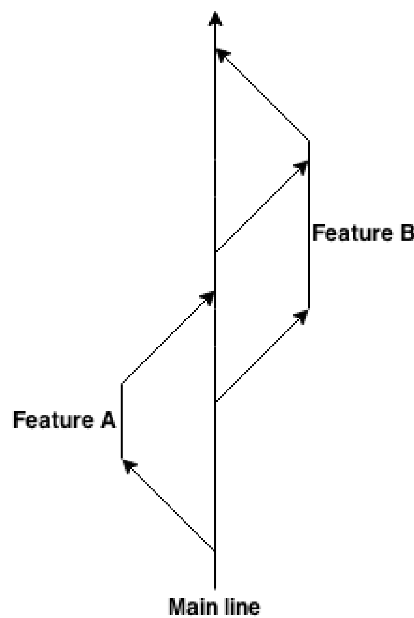


Figure 3.2: Branching strategy example.

tems for automatically handling test failure analysis and clustering, the latest incarnation being NIOCAT. NIOCAT - short for Navigating Information Overload Caused by Automated Testing - analyses test failure information such as error logs and other artifacts to cluster different test failures it deems similar, i.e. caused by the same fault [28]. While NIOCAT has been of great assistance in navigating the overload of information, Qlik still have related issues in their testing process.

3.4 Fault Localization Issues

Under the current operational procedures, a test failure is brought to the attention of the responsible manager(s) as soon as it occurs more than once. The manager will then look at the test itself to see which test team is responsible for the test case, and assign an issue statement to said team with a priority level depending on the severity and frequency of the failure. The test team that has been assigned the issue will then attempt to investigate the root cause of the issue, which can range from anything to faulty code, incorrectly written tests and fluctuations in test environment. Based on their evaluation, they will either perform repair or pass the issue on to another team for further investigation.

The process which the teams undertake for fault localization is not standardized. Currently there is limited information available to assist in investigation. For example, Bamboo will provide a video recording of the test sequence, logs and similar artifacts related to the test case. Aside from these artifacts, a developer investigating a test failure will usually only depend on manual fault localization methods such as attempting to reproduce the error, tracing the path through the code and manually inspecting recent deliveries to the repository. If a tool could be developed that would automatically do a great deal of this work significant improvements in fault localization time could be accomplished.

Since a stable main line is of high priority, this also means that management must be

very conservative with what they allow to be integrated into the main line, as issues arising on the main line essentially halts the entire development process. This can also cause "integration hell" as main line integration requests grow. As such other related improvements from an improved fault localization process could include that more issues would be dealt with in a timely manner and less stringent criteria would be needed for integrating code into the main line, thus shortening development cycles and reducing integration issues caused by late merges.

Chapter 4
Related Work

Due to poor scalability of manual fault localization techniques, there is a growing need for automated solutions for fault localization in software today. A variety of approaches have emerged in this field of research and a few of these are described briefly in this chapter.

Spectrum-based

Spectrum-based fault localization is a technique that finds suspicious code based on code coverage similarities from several failing test cases [22]. The code coverage results are separated into test cases that passed and test cases that failed, and by applying a simple mathematical operation a probability for each line of code (or statement) containing the fault is achieved. For example, if a line of code has been executed by 90% of the failing test cases, but only 10% of the passing test cases, it is likely to contain the fault and thus labeled with a high suspiciousness rating. Conversely, if a line of code has the percentages reversed, it is very unlikely to contain the fault.

Spectrum-based fault localization has shown promise, but it has some limitations. Since the probability of a line containing the fault is calculated by reviewing the combined results from a test suite, it depends on many tests being executed and a number of them using partially similar code [22]. If you were to run only a single test, no valuable information is given (as the total sum of executed lines will be deemed suspicious) and if the code contains a number of faults (i.e., a number of test cases fail due to different faults) the results might be confusing, indicating a similar suspicious rating for large areas of the code. Tarantula is an example of one of the more prominent spectrum-based techniques. Tarantula utilizes Equation 4.1 to calculate the suspiciousness rating of a given statement "s" from 0-1 (lower meaning higher suspiciousness) [21].

$$Suspiciousness(s) = \frac{Passed(s)/TotalPassed}{Passed(s)/TotalPassed + Failed(s)/TotalFailed} \quad (4.1)$$

Tarantula is accompanied by a tool that uses the suspiciousness rating from this equation to visualize the results. This is done by assigning each statement of the code a color from red, to yellow, to green, based on the suspiciousness rating. This gives the developer an overview of the system and a visual indication of where the fault(s) may be located. The tool also uses brightness to indicate general execution frequency of statements, further enhancing the overview [22] .

There are other spectrum-based techniques available, an example being Ochiai. Ochiai is built around the same principles as Tarantula but instead uses Equation 4.2 to calculate the suspiciousness rating [33].

$$Suspiciousness(s) = \frac{Failed(s)}{\sqrt{Failed(s) * (Passed(s) + Failed(s))}} \tag{4.2}$$

Machine Learning-based

Machine learning refers to the construction and study of algorithms that can improve themselves based on pattern recognition and computational learning. Wong and Qi attempted fault localization by means of using a back-propagation neural network, a machine learning model which has been applied to software risk analysis and reliability estimation in the past with good results [30]. The application of this model means using large sets of inputs and corresponding expected outputs along with code coverage data. By comparing the actual output and the expected output one can decide whether a test passed or failed. By analyzing the code coverage data for each test, a suspiciousness rating for each statement is achieved based on how frequently that statement occurs in failing test cases. Machine learning techniques unfortunately rely on a large number of inputs and expected outputs and this type of testing is not always applicable for testing web applications where behaviour driven test sequences are used, instead of typical white box-testing of inner mechanics and algorithms.

One of the issues with the spectrum-based techniques is that they do not immediately account for various tests failing for different reasons, thereby limiting the effectiveness when there are multiple bugs in the code. Briand et al. performed research that improved this aspect of Tarantula's fault localization using machine learning techniques. By clustering test failures based on their statement coverage and instead applying the Tarantula algorithm on the individual clusters a more narrow and accurate suspiciousness result was achieved [16].

Program State-based

Program state-based fault localization is a technique which compares different versions of the system to locate the fault. Zeller performed research based on Cause-effect analysis which lead to a fault localization system called Askigor [31]. The principle is to take two versions of the software under investigation, one which contains the fault (typically a newer version) and one which does not contain the fault. Both versions are then executed and the variables that are stored in the memory are collected and compared to acquire a list of variables that differed. These variables are then split with one part being added from the working to the faulty version and the faulty software executed once again. Depending on whether or not the execution still fails, more variables are added or removed. This

process is then repeated until the faulty variable is located, thereby simplifying the fault localization process [31].

Another program state-based method is predicate switching, proposed by Zhang et al [32]. Fault localization through predicate switching is performed by forcibly changing the branch paths that a faulty program takes until the desired output is achieved. The fault does not need to be located in the predicate itself, but by locating the predicate that needs to be changed in order to achieve correct program execution, one can identify the variables involved in that predicate. Backtracking from that predicate in order to determine where the values of these variables are changed can often provide valuable information about which statement is responsible for the incorrect behaviour. The main issue with the program state-based techniques is that they rely on a high number of test executions being performed in order to locate the suspicious statements. This is problematic in an automated testing environment where external agents need to build the system and execute complete test suites, creating test cycles that take several hours.

Program Slicing-based

Program slicing operates under the assumption that a given variable at a certain statement is responsible for the test failing. Slicing in this context refers to the act of isolating the parts of the code that can have an influence on the variable of interest up to a certain point, commonly refereed to as the slicing criterion [27].

Program slicing can be performed either by static analysis of code or through dynamic analysis, i.e. reducing the slice to the parts of code that actually influences the variable, rather than only the parts of the code that potentially could influence it. The slicing process can be further improved by performing execution based slicing, which in this context means slicing based on the result from a given set of test inputs, rather than any general input.

Chapter 5
Design Methodology

As described in Chapter 3, Qlik employs a completely automated build and test process using the build tool Bamboo and test framework Horsie. Combining these allows Qlik to perform automated regression tests on each development branch of their web application Qlik Sense. While the build and test processes are automated, the fault localization is still performed manually and is thus a slow and tedious process. To improve the efficiency of the fault localization process at Qlik we have developed a tool dubbed FLAC - short for "Fault Localization Automation using Code Coverage Data and Diff Utilities". In this chapter we detail the process we went through in conceptualizing and implementing the FLAC tool at Qlik, the final version of which is detailed separately in Chapter 6.

5.1 Investigation of Needs

In the case study we performed at Qlik it was clear that they were in great need of an automated fault localization tool to help streamline their fault localization efforts and increase efficiency. Before we arrived at Qlik, our supervisor Lars Andersson had already envisioned how he thought this could be achieved. Lars idea was based around comparing what has been executed in a failing test case with what has changed since it passed and Lars requested us to investigate this possibility. In our case study we found that this idea was feasible and made sense for Qlik. Since Qlik already has a working code base, test failures typically only occurs when changes or additions are introduced and because of this it makes sense to incorporate the data that can be retrieved from the version control tool via the diff utility.

Before we arrived at Qlik, they had not yet implemented code coverage into their testing process and it was revealed that there was a great need for this type of data in other areas of Qlik as well. For example, test developers expressed a need for code coverage to measure how well their tests actually cover the code that they intended and also seeing where they might have multiple tests unnecessarily covering the same areas. Several managers also

expressed that there was a need to analyze how much of Qlik Sense was actually covered by the test suites that exist in order to identify areas of the product that need more rigorous testing. From this information we concluded that implementing code coverage into Qliks automated build and test environment was a worthwhile endeavour despite being quite difficult.

5.2 Assessment of Related Work

During our study of related work in the field of fault localization we could not identify any previous attempts at the solution we propose in this master's thesis. A great deal of the solutions that exist do employ code coverage, such as the spectrum-based and the machine learning-based techniques, but none of them use information from the version control tool [22][30]. The issue with only using code coverage is that the algorithms used to identify suspicious code are dependant on the execution behaviour from several test cases. For spectrum-based techniques for example, the suspicious code should preferably only be executed by failing test cases and ideally a number of failing test cases will execute the same suspicious code, otherwise you might get confusing or diluted suspiciousness ratings across vast parts of the code base. For Qlik this can be problematic since it is not uncommon that only a single test case suddenly starts to fail.

One approach that shares similarities with ours is the program state-based techniques proposed by Zeller [31]. In the fault localization system he developed the suspicious code is found by comparing two different versions of the system, one containing the fault and one that does not. By moving variable states from the working to the faulty version of the software the suspicious variable and code associated with it can be identified. This comparison does not employ the version control tool however and instead compares the state of various variables during execution.

In our research we found that Jones et al had further enhanced their spectrum-based fault localization tool Tarantula by creating an accompanying visualization tool [22]. Using this tool, one gets an overview of the code base and a visual representation of what lines of code are deemed suspicious. We recognized that this would be a great way to intuitively present the results from an automated fault localization to a developer and we decided that we would also use a similar approach in our tool.

5.3 Solution Idea

The principle behind automating the fault localization process at Qlik is divided into three parts:

- Applying code coverage to the test suites and saving the code coverage results from each test case individually.

- Collecting the Git diff relating to the commits that have been introduced between the passing and the failing test executions.

	1st It.	2nd It.	3rd It.	4th It.	FLAC
JS Code Coverage	N	Y*	Y**	Y	Y
Git Retrieval	Y	Y	Y	Y	Y
Comparison	Y	Y	Y	Y	Y
Line Output	N	Y	Y	Y	Y
Commit Output	N	N	N	Y	Y
Qlik Sense Visualization	N	N	N	N	Y

Table 5.1: Overview of the functionality offered in each iteration.
*Yes, but not implemented in Qliks testing framework.
**Yes, but not integrated in Qliks continuous build system.

- Comparing the information about what has been executed in a failed test with the diff information about what has changed and presenting it in a manner that is intuitive and simple to understand for developers and testers.

A great deal of code is executed in a given test case and a large amount of files also undergo changes between builds meaning that there is often a lot of code that needs to be examined when investigating a test failure. The premise is that the third step will be able to provide a list of suspicious lines of code that have potentially been incorrectly implemented or changed, thereby narrowing down the search space significantly. Qlik Sense is written using a variety of different programming languages, but in this master's thesis we have limited our work to the front end which only uses JavaScripts. This was done due to time constraints.

5.4 Prototype Development

The development of the FLAC tool was done in an iterative manner in order to get a proof-of-concept for the individual parts required to perform the task. This also allowed us to experiment with various ideas and acquire the necessary knowledge to produce the final prototype. An iteration typically consisted of 2-6 weeks of work where each one ended with the completion of one or more significant functionalities, see Table 5.1 for an overview.

First Iteration

In the initial step, we used an integrated development environment (IDE) called Eclipse, which is a popular Java editor that allows direct execution of code, adding additional plug-ins and other features. A simple Java program with an accompanying JUnit test suite was imported into Eclipse and a plug-in called EclEmma was installed. EclEmma is a code coverage plug-in for Eclipse designed specifically for Java code coverage [3]. Figure 5.1 shows an example of how Eclipse displays source code after applying code coverage with EclEmma.

To test our concept a Git repository was set up for the project and various minor changes in the program were committed to the repository. We wrote the code necessary to run the test suite, collect the code coverage results, retrieve a Git diff of our choice and perform the comparison. The test cases were created in Java and did not use any server-client communication. The comparison at this point was not sophisticated enough to find the potential suspects in terms of exact lines, instead it simply displayed the methods that could contain fault lines. The major bridge crossed in this step was showing we could collect the Git diff and compare it to the code coverage results in a meaningful manner.

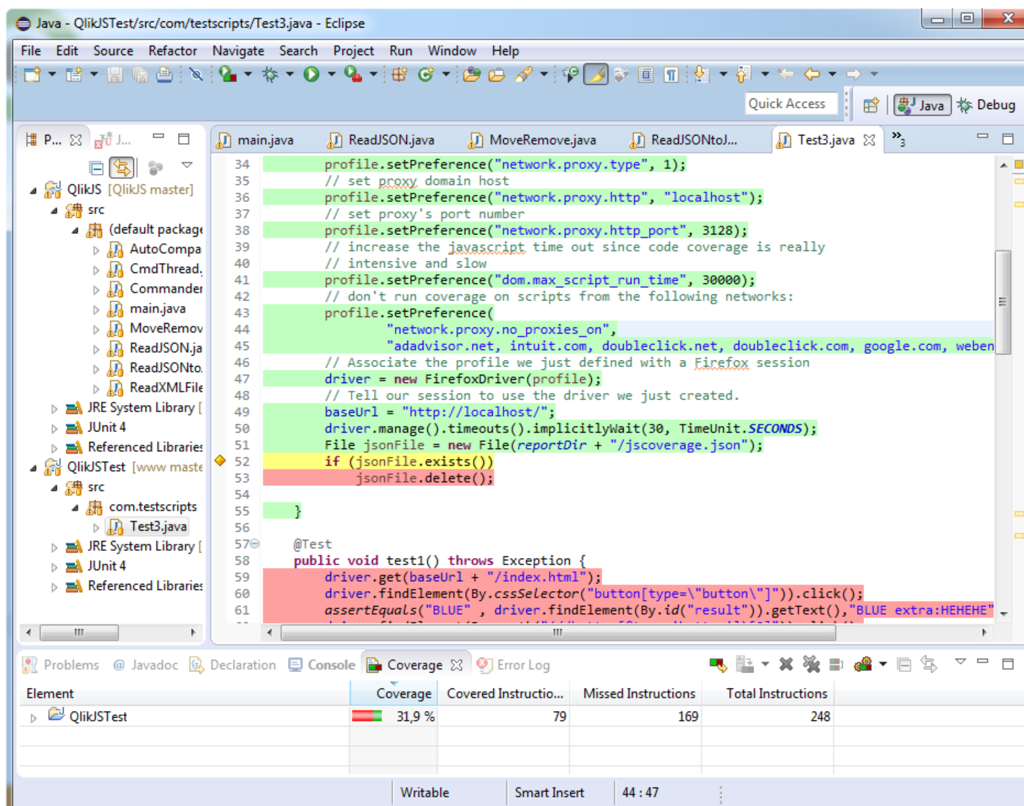


Figure 5.1: Example screen using EclEmma [3].

Second Iteration

In the second iteration, we wanted to show that we could apply code coverage in a situation more similar to the one Qlik operates in. This meant applying code coverage in a browser executing JavaScripts, rather than simple Java code in an IDE. Selenium is the automated testing tool that Qlik bases their test framework on (see Section 3.3 for details), and also the one we used for this iteration. We wanted to use a code coverage tool that could likely be carried over to Qliks testing framework as well, and the decision finally fell on an open-source JavaScript code coverage tool called JSCover [6]. Other tools considered and experimented with include Istanbul and Blanket [1][5]. The primary reasons for choosing JSCover was that an older version had already been successfully tested at Qlik in the past and this along with the easy-to-use output made it the most suitable option.

JSCover takes the JavaScript in question and creates a new copy of the file, where it has added the code that it needs to collect the coverage data. This is refereed to as *instrumenting* the code. There are two ways of using JSCover, either you can instrument all the code in advance on the server, or you can have JSCover act as a proxy and intercept the scripts when they are being sent from server to client. In this approach, instrumentation is done on the spot before passing on the script to the client. In both cases you have the client executing instrumented code instead of the original source code and JSCover collects the data it needs afterwards. We successfully attempted both methods in this iteration.

In order to perform testing, a simple website was created using HTML and JavaScripts. After executing a test suite, the JSCover data can be extracted on the client side as a JSON file by executing a simple command, an example of which can be seen in Figure 5.2. A JSON file contains a JSON object that associates keys with certain values. In this example, the file name acts as a key, and associated to that key is a value, in this case another JSON object. This object is similarly constructed with a key and value pair. The key we are interested in is "lineData", which has an associated array as value. This array contains each line of code from the file in question, described in terms of how many times that line has been executed. Lines with null instead of a number are non-executables, e.g. comments.

```
{"path/testfile.js":{"lineData":[null,null,1,1,0,null]}}
```

Figure 5.2: Example of the code coverage output from JSCover

Third Iteration

The next step in development led us to begin implementation in Qliks test environment. Qliks test development is done in Visual Studio using the programming language C#, which is what we used in this phase. We setup the entire Qlik Sense and Horsie framework in a server debug mode to allow us to perform changes and execute tests locally right away. We instrumented and added the necessary code needed to perform the code coverage and once again implemented the Git diff retrieval and comparison.

```
{"path/testfile1.js":[30,31,33,37],
 "path/testfile2.js":[1,2,3,153],
 "path/testfile3.js":[13,14,81,82]}
```

Figure 5.3: Example output from the third iteration prototype

In this iteration we also made the comparison more intricate and sophisticated. The results were produced as a JSON file, showing exactly which files and which lines of code in each file overlapped between the code coverage and the Git diff, see Figure 5.3 for an example. In this output the file names act as keys and associated with each is an array. This array contains the exact lines of code that were executed in the test and also changed between the two Git commits used to create the diff.

Fourth Iteration

In Chapter 3 we described the software configuration management, continuous build tool and automatic testing framework employed by Qlik. Since we wanted our prototype to be implemented into the daily work for Qlik, we needed to make it run in the continuous build tool and automatic testing framework used. At the time of performing this master's thesis, Qlik used Bamboo as their continuous build tool and the first step of this iteration meant that we needed to make our edited version of Qlik Sense that employs code coverage run in Bamboo.

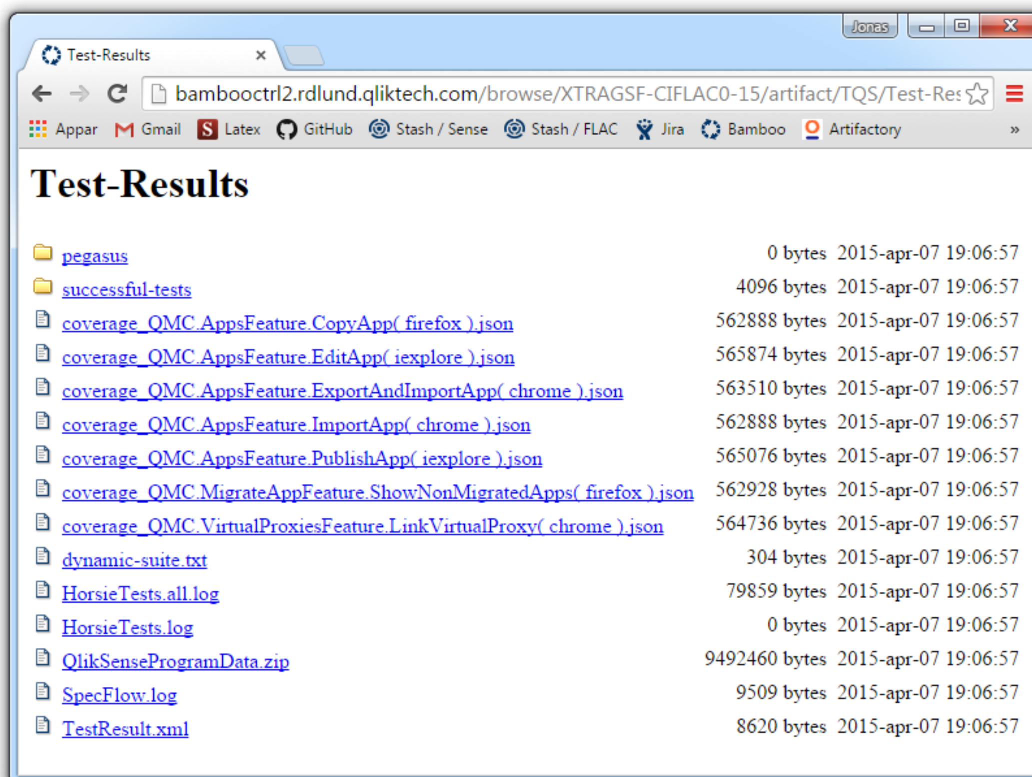


Figure 5.4: Example screen of test artifacts.

Building and running our code in Bamboo was difficult for many reasons, primarily because JSCover performs several noticeable changes to the code it instruments, for example adding lines and also removing comments. The reason this is problematic is that the build scripts used by Bamboo to build the Qlik Sense components have several steps that verifies that the code meets certain criteria. These include checking that specific coding conventions are used, minifying the code and making sure that each JavaScript file has a comment that says who the owner of the file is. These and other related issues forced us to perform extensive changes in the build process to make Bamboo accept our instrumented code.

All the build steps and tests in this automatic build and testing process are performed on dedicated machines refereed to as "agents". The next step of implementation consisted of saving the code coverage result in a manner where it could be retrieved for later use. Conveniently enough the agents already save several artifacts related to the tests and Bamboo

automatically publishes them for later analysis. We exploited this process to also publish our code coverage result in the same manner, simply creating a separate code coverage file for each test case. An example of test artifacts published in relation to a given test suite can be seen in Figure 5.4.

In the last iteration, we hard coded the Git diff retrieval process and allowed Horsie to automatically perform the comparison. This was done for convenience during the testing process, but what we actually want is a separate application that does this when requested to do so. For this reason, we migrated out the Git diff retrieval and comparison process in this iteration and turned it into its own Java application. In this prototype, the input consisted of two hash codes, i.e. the Git commits you wanted to diff between, along with the test case you wanted to examine. The prototype would use these commits to collect the diff and the name of the test case to go to the Bamboo artifact server and collect the code coverage JSON-file. Finally, it used these files to perform the comparison and create the actual result, a brief file describing exactly which files and their associated lines that met the criteria for suspiciousness, i.e. lines that had been executed and also changed. An example of the input screen for this prototype can be seen in Figure 5.5.

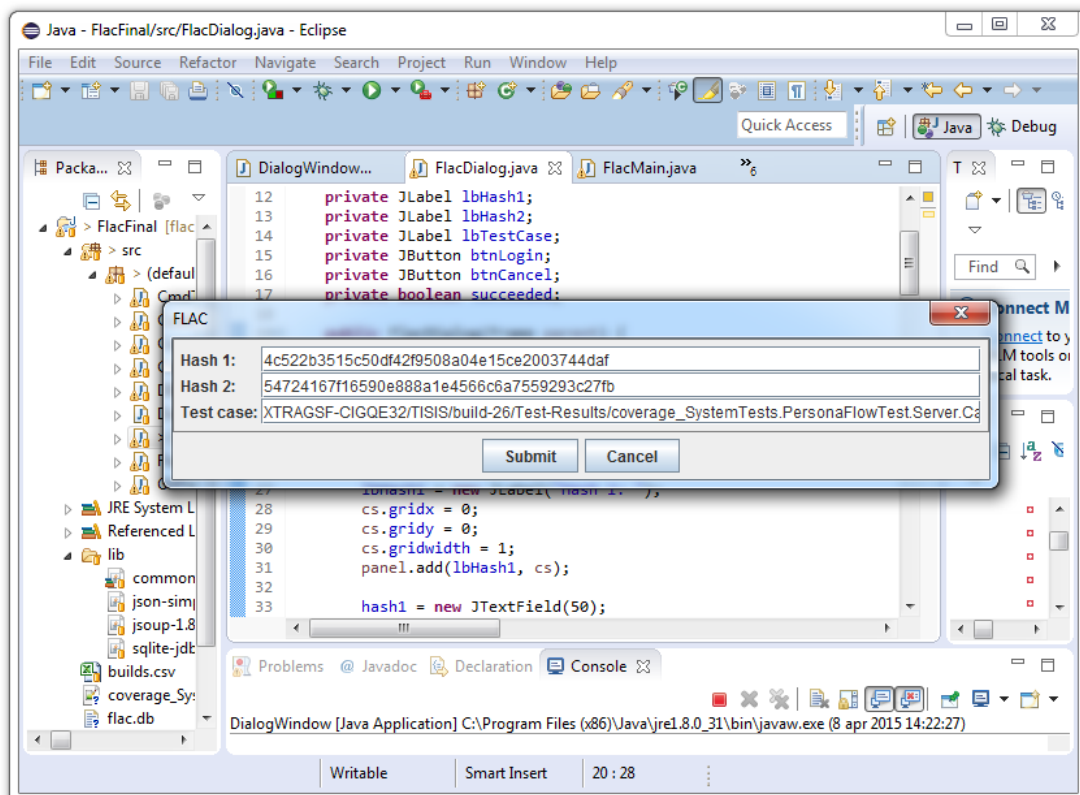


Figure 5.5: Example of FLAC prototype input screen.

Chapter 6

FLAC

In this chapter we detail how the final prototype of the FLAC tool works and how it is intended to be used at Qlik. FLAC uses the code coverage data from each test case and compares it with a Git diff of choice. The output is a set of statements that have been both executed in the test run and that have also been changed or added within the duration of which the diff covers. To improve FLACs usability, we have also developed an accompanying Qlik Sense application that reads the FLAC results and displays them in a way that is easy and intuitive for testers and developers to understand.

Java Application

FLACs primary functionality is offered in the form of a Java application that can be seen in Figure 6.1.

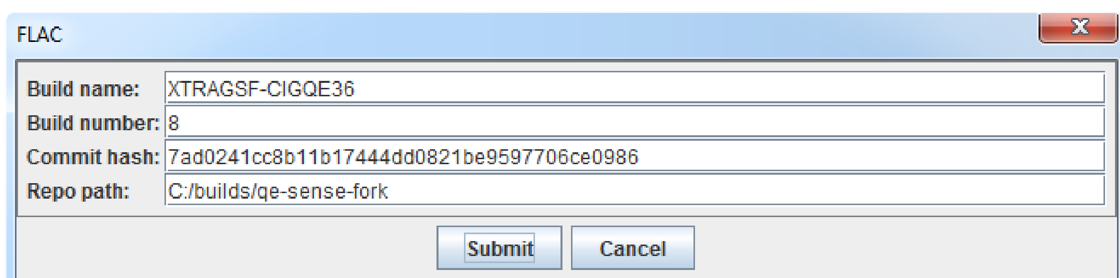


Figure 6.1: Example screen using final FLAC prototype.

FLAC requires the following input from the user:

Build name When executing test runs in Bamboo a build name is chosen to be able to identify the combination of branch and test plan to be used. A branch can be tested in many test plans and similarly a test plan can be used to test many branches. FLAC

uses this name to know which build it should go to in order to retrieve the coverage data.

Build number Test runs are performed sequentially in bamboo and as such FLAC also needs to know from which build number to look at in order to retrieve the coverage data. Associated with each build number is the latest commit that was delivered for the test run. FLAC also retrieves the hash code for that commit which is used to create the diff file.

Commit hash A Git diff is constructed using two commit hash codes and displays all changes that have occurred between those two commits. FLAC automatically collects the most recent commit associated with the execution run from Bamboo, but how far back FLAC should diff against is up to the user. The more recent the commit is, the less false positives you will risk FLAC finding. However, if a commit is chosen that is more recent than when the issue was introduced, FLAC will not find it. It might seem obvious to use the commit associated with the latest passing test run, but sometimes intermittent issues are being investigated and you will need to look further back.

Repo path FLAC uses the path to the project to know where to execute the diff command. In order to be able to execute the diff command the branch must be checked out and up-to-date.

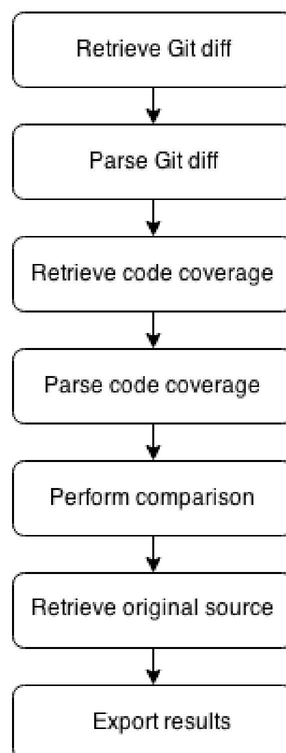


Figure 6.2: FLAC execution steps.

The major steps that FLAC performs can be seen in Figure 6.2. Flac begins with retrieving the Git diff and parsing it to be used in the comparison. A Git diff will have

an appearance in line with the example in Listing 6.1 (a real diff will be much larger and contain a variety of files with a number of changes however). The parser goes through the entire diff and the first thing it does is check whether the changed file is a JavaScript file or not. A typical Git diff at Qlik will include various file types that are part of the project, but we are only interested in the client side code that is executed in the browser, which are JavaScripts. After that, the parser identifies the line that tells you what row numbers are affected by the upcoming change, which is noted by the two at-signs. In this case the diff tells us via the plus sign on this line that the first row number is 61 and the total rows for this change is 6. Next the parser will check for lines that have been changed or added. Unchanged lines start with a space, removed lines start with a minus sign and added lines start with a plus sign. A changed line is simply considered to be the removal of the previous line and the addition of the new one as in our example. The row number for added lines are calculated and included in an array associated with the file name. When the entire diff has been analyzed, a JSONObject is created consisting of file names as keys and these arrays of row numbers associated as values.

Listing 6.1: Example from Git diff

```
--- a/examplefile.js
+++ b/examplefile.js
@@ -61,6 +61,6 @@

        beforeEach( function () {
                setupModule();
-               initiateFactory( selections );
+               initiateFactory( selectionSet );
        } );
```

The next two execution steps deal with retrieving the code coverage data and parsing it. For more details on the code coverage saving and retrieval process, see Section 5.4. JSCover provides the coverage data for each test case as a JSON containing file names as keys and arrays with the same number of elements as lines in the file associated as values. Each element is either a number or null for executable and non-executable lines respectively. The problem in our context is that for a file that contains any sort of multi-line declaration or function call such as the example in Listing 6.2, JSCover labels only the first line as executable and the rest as non-executable. From a traditional code coverage perspective this makes sense, because it can then treat the entire declaration as a single line. However, since we want to compare what has been changed with what has been executed, we need to consider all those lines executable, and give them the same number of executions as the first line. This is what the code coverage parser does.

Listing 6.2: Example code with multi-line declaration

```
sectionIDs: {
        template: "template",
        field: "field",
        time: "time",
        dimension: "dimension",
        measure: "measure",
        visualization: "visualization"
```

}

After retrieving and parsing both the Git diff and the code coverage data FLAC performs a simple comparison of the arrays to see which files and line numbers appear in both the diff and the code coverage data. The comparison results are exported to CSV files to easily be imported into Qlik Sense. Along with this FLAC also retrieves the complete source files from Bamboo as they appeared when executed in the test run. When performing the instrumentation of the code, JSCover saves a copy of the original source files in a separate folder which is what we collect. This is necessary for the visualization in Qlik Sense.

Qlik Sense Visualization

After the Java application has created the CSV files that contain all the information necessary, these files can be imported directly into Qlik Sense using the application we have developed. Once imported the application will display the results as in Figure 6.3. To the left the full list of test cases that were executed in the test suite are displayed. Next to each test case is the result of the test, i.e. `Passed` or `Failed`. Most often failed test cases are of primary interest, but it can also be valuable to analyze and compare the results from other test cases. This could for example be to see if passing test cases have also executed lines deemed suspicious in a failing test case.

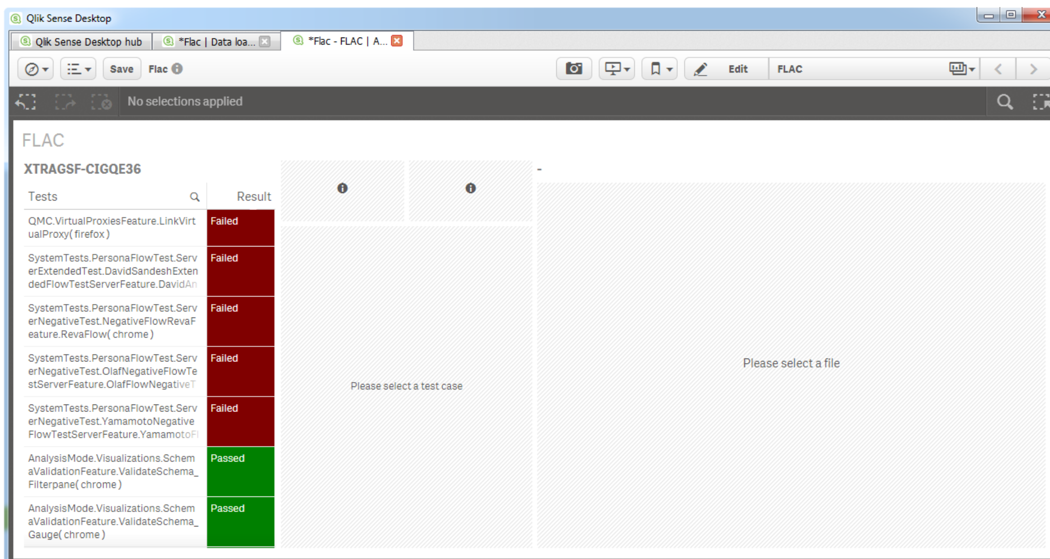


Figure 6.3: Example screen displaying FLAC results in Sense.

After a test case has been selected, the application will display the full list of source files that have been executed in the test case, see Figure 6.4. At the top the most interesting files are displayed, these are files that include lines deemed suspicious. Below are files that are considered of secondary interest. One issue with all previous automated fault localization systems that use code coverage is that the way they find suspicious code depends on the code existing and being executed. What this lacks is some kind of indication that code may have been erroneously removed. The way this has been approached with FLAC is that if the application can not find any suspicious lines, it will still check if a file has undergone

any sort of change while still being executed. In this case the application will label this file as secondary for inspection, meaning that it is worthwhile to look at the version control history of the file to see if necessary code has been erroneously removed. This is only required if the fault can not be located in the lines deemed suspicious.

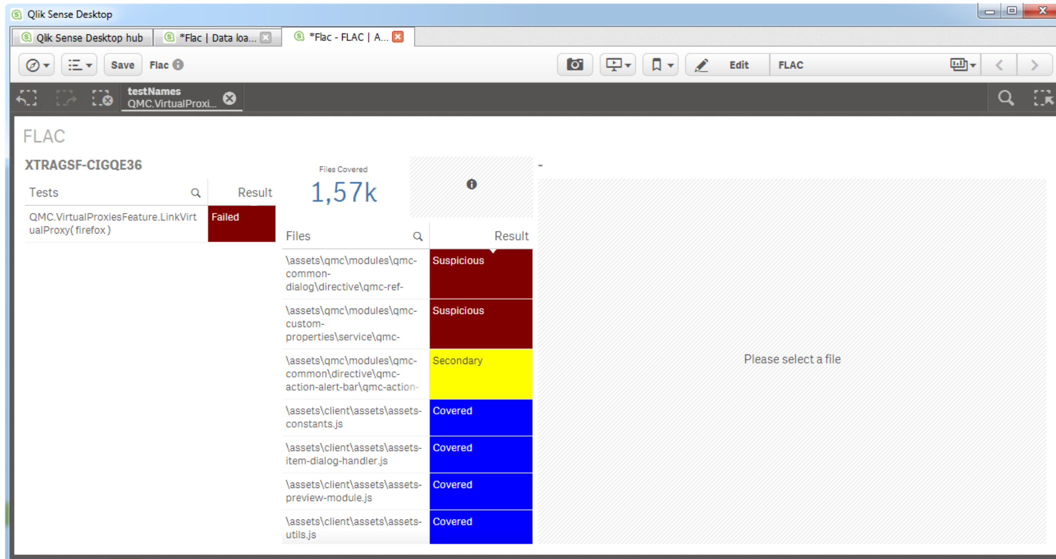


Figure 6.4: Example screen displaying FLAC results after a test case has been chosen.

At this point the application also displays a counter of how many source files have been covered in the test case. From here any file can be chosen for further inspection. Once it has been chosen the complete source code is displayed to the right, with each line highlighted as either `Suspicious`, `Covered` or `Not touched`. Figure 6.5 shows an example of how the application displays the source code of a chosen file. At this point another counter will also appear that displays how many lines in the file were deemed suspicious. As we can see only one line was deemed suspicious which is the highlighted return statement. This statement has been incorrectly altered, causing the test to fail and thus the fault has been identified.

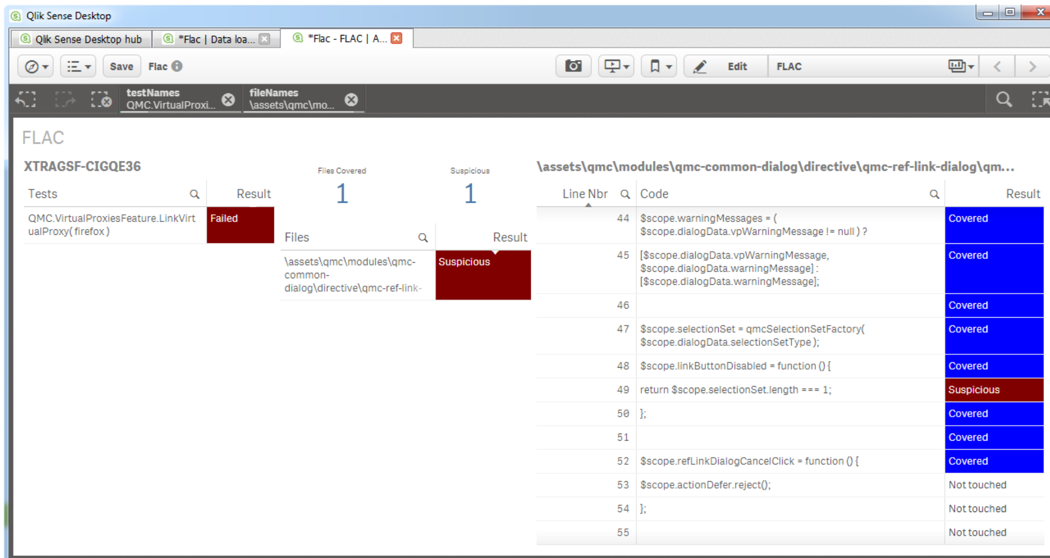


Figure 6.5: Example screen displaying FLAC results after a source file has been chosen.

Chapter 7
Evaluation Methodology

To evaluate the performance and usability of FLAC we have focused on three characteristics from the ISO 9126 quality model, the standard for evaluating software quality:

Functionality Are the required functions available in the software?

Efficiency How efficient is the software?

Usability Is the software easy to use?

There are three more characteristics in ISO 9126: Reliability, Maintainability and Portability [15]. These are centered around the quality of the final product but since FLAC is still a prototype these characteristics will not be taken into account at this stage of development. In our context functionality means verifying that FLAC actually locates the faults that have been introduced. To do this we introduced common types of errors in the source code and analyzed the results with FLAC. To check how efficient FLAC is we wanted to see how much FLAC can narrow the search space of a diff when investigating test failures and also how quickly it performs its functionality. Finally we wanted to determine if a typical user finds FLAC useful and whether or not it actually assists them in their fault localization efforts. The results from the evaluation can be found in Chapter 8.

7.1 Functionality

A great deal of the automated fault localization methods that have been developed, such as the spectrum and program slicing-based techniques, have been tested using standardized test suites [27]. An example of such a test suite is the well-known Siemens suite [25][20]. The Siemens suite is a system of 132 C-programs that have a variety of faults injected. Since FLAC is completely based on analyzing the version control history it makes no sense to apply it to this type of static test suite however. Furthermore the FLAC prototype

is designed to run in Qliks test environment analyzing JavaScript code coverage and as such none of these standardized test suites have been used for evaluation.

To evaluate FLACs functionality we instead took a version of Qlik Sense that did not generate any test failures and injected it with a variety of faults. The exact faults that were introduced can be seen in Appendix A. Test runs were conducted on the now faulty version and the subsequent test failures were analyzed with FLAC to verify whether or not it correctly identified the faults that we had introduced. The faults that were identified were corrected and the new version of the system was re-tested. This cycle was repeated until all test failures disappeared or FLAC no longer found any suspicious code. This entire process is refereed to as an evaluation run. If FLAC correctly identified all faults that caused failures and the final re-test did not produce any failures, the evaluation run was considered a success, see Figure 7.1.

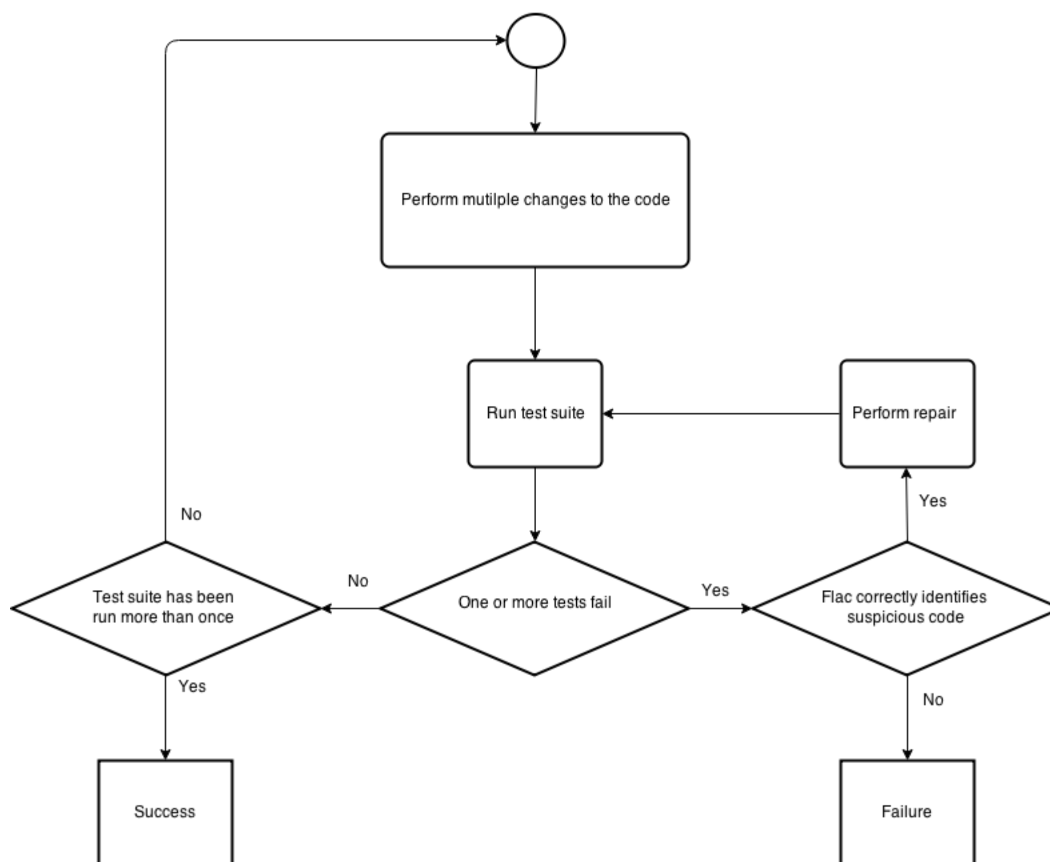


Figure 7.1: FLAC evaluation run.

The following data was gathered for each evaluation run (see Appendix A for complete documentation):

- What changes had been made in which source files.
- Total number of test failures resulting from the changes.
- Which, if any, faults were identified by FLAC and subsequently repaired.
- Which, if any, faults were not identified by FLAC.

- Which, if any, faults did not cause any test failures.

In total 10 of these evaluation runs were conducted, using a variety of different faults for each run spread evenly throughout the code. At the end of each evaluation run, all changes that caused test failures were categorized as either `Found` or `Unfound` by FLAC. Based on these numbers a hit rate was calculated for each evaluation run based on Equation 7.1. The types of faults that were introduced had to be chosen based on a few limitations caused by the competence of Qliks build process. For example no faults could be introduced that were purely syntactical as these were caught immediately by the verification steps in the build process. This meant that the faults had to be limited to operations such as changing value assignments, return statements and predicates.

$$HitRate = \frac{Found}{Found + Unfound} \times 100 \tag{7.1}$$

7.2 Efficiency

When performing manual fault localization a common technique is to look at the project history in the version control tool, i.e. doing a diff between the current and a former version of the software. The problem in a development environment such as the one Qlik operates in is that there is an overwhelming number of changes made on a daily basis and as such it is essentially impossible to go through a diff file manually in search of potential errors. The main operation of FLAC can be seen as taking the diff file and scraping away all the changes that are not of interest (i.e. have not been executed by the test case) and therefore it is of great interest to see just how much FLAC is able to reduce the search space in the diff.

To perform this comparison, we looked at the diff file daily for 3 consecutive days to analyze how much the search space was reduced by FLAC for each test case. The diff was chosen to cover 1, 2, and 3 days of commits to provide diff sizes representable to real-life use cases. In total 68 test cases were executed over 3 days, resulting in 204 number of separate test executions being analyzed. The reduction percentage R for a given test case T_c was calculated according to Equation 7.2, where $Lines(T_c)$ are the number of rows deemed suspicious by FLAC for T_c and $Lines(T)$ are the total number of rows changed in JavaScript files.

$$R(T_c) = \left(1 - \frac{Lines(T_c)}{Lines(T)}\right) \times 100 \tag{7.2}$$

One of the advantages of FLAC compared to other fault localization techniques such as the program state-based is that FLAC does not require the execution of multiple tests nor re-runs to produce a result. The code coverage result from one test execution is sufficient to produce an output and re-runs are only necessary to verify whether or not the repairs performed have been satisfactory in removing the test failures. To indicate that the improvements in time spent on test execution was not off-set by longer calculation times, we gathered the calculation time spent by the computer in performing the FLAC functionalities, both the Java and the Qlik Sense application, from the efficiency executions described above. For the Java application the timing was started as soon as the user hit

ID	Position
P1	Triad Technical Lead
P2	Triad Technical Lead / Strategic Prototyper
P3	Test Developer
P4	Test Developer
P5	Internal App Developer
P6	Test Developer

Table 7.1: Interview participants and their roles.

the "Submit" button and stopped when it finished producing the CSV files. For the Qlik Sense application we simply checked the time it takes to import the CSV files, which is all the execution it needs to perform. The tests were conducted on a 3.2 GHz Intel Core i7 PC with 32 GB RAM running Windows 7.

7.3 Usability

In order to evaluate FLACs usability we conducted a survey at Qlik with a set of participants from various sections of the company. The survey was conducted as interviews with primarily open questions and the participants were encouraged to go outside the predefined questions when inclined. Participation in the survey was completely optional and the participants had the option to opt-out of the study at any point. All answers were recorded in their entirety, but the major points were summarized textually in Appendix C.

Before answering the interview questions the participants were given a brief introduction to what FLAC is and how it is intended to be used. The participants were given the task to investigate a set of test failures that we had purposefully created by injecting the source code with faults. After experimenting with FLAC the participants were requested to answer the questions in Appendix B. The six participants included in the survey can be seen in Table 7.1.

Chapter 8

Evaluation Results

In this chapter we present the results from the evaluation described in Chapter 7. Evaluations were performed based on three characteristics from the ISO 9126 quality model: functionality, efficiency and usability.

8.1 Functionality

To test the functionality of FLAC we performed 10 evaluation runs where we introduced faults into otherwise stable software and used FLAC to investigate the subsequent test failures caused by these changes. The faults that caused test failures were categorized as either `Found` or `Unfound` by FLAC and a hit rate percentage was calculated using Equation 7.1. In 9 out of 10 evaluation runs, the faults were restricted to either introducing new lines of code or altering existing lines. In the final evaluation run the faults were restricted to removal of necessary lines, in order to evaluate the performance of FLAC in locating files of secondary interest (see Chapter 6 for details).

The average number of faults introduced in the initial 9 evaluation runs were 5.33 (ranging from 3 to 9). In 1 out of these 9 evaluation runs, no test failures were caused by any of the introduced faults and because of that no hit rate could be calculated. For the remaining 8 of the evaluation runs the hit rate was 100%, in other words FLAC correctly identified every fault that caused a failure. In 6 of these all faults were detected at the first test run, whereas 2 required re-runs due to some fault(s) blocking the execution of other faults. In the final evaluation run designed to verify the secondary functionality removals of necessary lines were performed in 2 files. FLAC correctly identified both of these files as secondary. The average number of initial test failures for each evaluation run was 21.1 (ranging from 1 to 113). For a list of total amount and types of faults introduced, see Table 8.1.

	Number of faults
Declaration statements	30
Predicate statements	13
Return statements	5
Removed statements	2
Total	**50**

Table 8.1: Types and amount of faults introduced.

(%)	**1-day**	**2-day**	**3-day**
1st Run			
Average	83.6	83.6	84.6
Maximum	84.2	84.1	88.1
Minimum	82.8	82.7	**69.6**
2nd Run			
Average	89.7	88.5	88.4
Maximum	92.2	90.3	90.3
Minimum	72.4	74.5	74.5
3rd Run			
Average	87.3	88.8	88.1
Maximum	**94.5**	92.9	91.6
Minimum	74.6	76.5	77.3

Table 8.2: Reduction percentages for FLAC.

8.2 Efficiency

To evaluate the efficiency of FLAC we conducted a number of tests aimed at checking how much FLAC can reduce the search space in a diff file when investigating test failures. The diff was chosen to cover 1, 2, and 3 days of commits to provide diff sizes representable to real-life use cases. In total 68 test cases were executed over 3 days, resulting in 204 number of separate test executions being analyzed. The reduction percentage for a given test case T_c was calculated according to Equation 7.2, where R represents how much of the original diff file needs to be investigated according to FLAC. The combined average $R(T_c)$ was 87.0%, and the maximum and minimum values observed were 94.5% and 69.6% respectively. For detailed numbers, see Table 8.2.

To indicate that the improvements in time spent on test execution was not off-set by longer calculation times, we gathered the calculation time spent by the computer in performing the FLAC functionalities, both the Java and the Qlik Sense application, from the efficiency executions described above. The most time consuming tasks were those of downloading the code coverage data and the original source code, as well as producing the CSV output files for Qlik Sense. On average, the Java application took 68.1 seconds to perform all its functions and the Qlik Sense application consistently took 9-10 seconds

	1-day	2-day	3-day
1st Run			
Execution Time (s)	68.8	65.1	70.2
Diff File (kb)	7056	8581	8858
Downloaded Content (kb)	46734	46734	46734
2nd Run			
Execution Time (s)	63.8	65.9	70.3
Diff File (kb)	2101	9077	10585
Downloaded Content (kb)	47300	47300	47300
3rd Run			
Execution Time (s)	65.9	78.4	64.5
Diff File (kb)	3005	5036	11967
Downloaded Content (kb)	47459	47459	47459

Table 8.3: Performance results for FLACs Java application.

to import the data. The combined execution time for both applications never exceeded 90 seconds. For complete details on the performance evaluation, see Table 8.3.

8.3 Usability

To evaluate the usability and general impression of FLAC, we conducted a survey where we allowed six participants the opportunity to experiment with FLAC on a set of test failures that we had intentionally created by injecting the code with a fault. After they had experimented with FLAC we conducted a short interview. The goal was to interview a group as diverse as possible. We would have preferred to have included more participants, specifically more developers, but time and resource restraints limited the survey to the participants included. All of the participants either worked closely with the developers or had been working as developers before their current position however.

Response from the survey was generally positive, all participants agreed that they found FLAC useful. Five out of six participants agreed that it was likely to improve their fault localization efficiency, while P6 was simply unsure as her position as test developer rarely makes her look at source code directly. The participants generally gave very similar responses in regards to which techniques they traditionally used for fault localization. These include checking error logs, version control history, screen shots and videos from failed test cases. There is no standardized procedure established and each test failure gets a separate treatment based on severity and who the investigator is.

All participants responded that it was easy to understand conceptually what the intention of FLAC is, but several users found that the application was somewhat confusing at first due to the lack of textual information or examples to guide the user. A common complaint was that no context was given around the suspicious files and several users wanted to see more information about them, for example immediately displaying number of lines that were suspicious in each file instead of having to choose a file to see this number. Several

users also requested the ability to cluster test failures based on similarities in suspicious code which could be useful when there is more than one fault causing the test failures. One user requested that the suspicious lines would be ranked based on how suspicious they are. This could be solved for example by applying spectrum-based fault localization theory and checking how many failing and passing test cases execute a given suspicious line. This is explored further in Section 9.3.

Chapter 9
Discussion

In this chapter we provide a basis for discussion regarding the work performed in this master's thesis. We analyze the results from Chapter 8, discuss known limitations of FLAC and also look at future work that can be done in relation to the work presented here.

9.1 Discussion of Results

In Section 1.2 we established the following three research questions regarding an automated fault localization tool (AFLT):

- How can an AFLT be implemented to find potential fault suspects based on code coverage and file change data?
- How can the efficiency of an AFLT be quantified and measured?
- How can the results from an AFLT be displayed in a manner that is intuitive and easy to understand for developers and testers?

The numbers presented in the functionality section indicate that FLAC is a very promising tool for performing fault localization. The method we implemented to perform the comparison of code coverage and file change data has proven to be very accurate and clearly identified the suspicious code in all of the tests we performed.

To quantify the efficiency of FLAC we investigated how quickly FLAC performs its functionality. In our context, which is that of a very large company that works with a code base consisting of around 1700 files, the accumulative time to perform the FLAC functionalities never exceeded 90 seconds. These numbers are very good but it should be noted that this only deals with the efficiency of FLAC performing its own functionality. To quantify if and how much FLAC can improve the fault localization efforts as a whole, deeper analysis would have to be made. All except one of the participants in the survey

thought that FLAC could increase the efficiency of their fault localization, but an exact measurement of the increased efficiency is nearly impossible to give. This is due to the fact that the fault localization process differs widely from fault to fault.

Another efficiency measurement that we investigated was how much FLAC could reduce the search space of a Git diff. Our results show that on average FLAC can reduce the diff by 87%. The percentage is calculated by comparing the part of the diff that concerns JavaScript-files with the amount of suspicious lines noted by FLAC. If we instead compared FLACs results with the entire the diff the percentage would be even higher. This result indicates that the efficiency in terms of the amount of data that a user needs to investigate is lowered greatly with FLAC compared to simply looking at the diff.

Displaying the results of a fault localization tool in an intuitive manner can be difficult which our survey showed. While all the participants were very positive as to the usefulness of FLAC, many had comments regarding how the usability could be improved. We are confident that performing minor changes to FLAC such as adding textual information to guide the user within the Qlik Sense application will significantly improve how intuitive users find the application.

9.2 Known Limitations

In Theorem 1.1 we described that the principle behind FLAC is that faults that have been introduced into previously working software are likely contained in the set of statements $\Phi = \zeta \cap \beta$, where ζ represents the set of statements that have been executed in a given test and β the set of statements that have changed since it passed. In order to understand the limitations of FLAC, one needs to look at what these two variables ζ and β represent.

Similar to the program state-based fault localization technique proposed by Zeller, FLAC depends on the existence of a working version of the software under testing to perform its functionality. FLAC can not identify faults that are already present within the code, but instead relies on the diff to identify faults that have been recently introduced. These faults are likely to have been introduced with the implementation of a new feature or a bug fix and are thus part of β. This makes FLAC ideal for performing fault localization in an environment in which you are continually working and updating existing software, but it provides no benefit if you are looking at a piece of software that has never worked as intended or lacks a version control history with high granularity.

FLACs current implementation is also dependant upon the existence of added or changed lines within the diff to accurately identify suspicious code. Since a removed line is not executed, it will not be part of the set ζ. To combat this issue, we created the concept of files that are of secondary importance, in this context meaning files that have been changed somehow, but do not contain additions or changes that have been executed. This means that there is a high probability that something has been removed in that file and it is thus worthwhile to look at the files' version control history to see exactly what that was. There is no guarantee that this is the case however, because FLAC will also highlight a file as secondary if the change is an added or changed line that has not been executed. In Section 9.3 we look at possible future implementations that could negate this limitation.

If two (or more) faults have been introduced simultaneously and one fault stops or incorrectly alters the execution path of the code, the remaining fault will not be caught

by FLAC since it has not been executed yet. The second fault will not be indicated as suspicious until the first fault has been corrected and the test has been re-run using the fixed software. In an environment such as Qlik's where a re-run can take hours this has the potential to slow down the fault localization process. This limitation exist because of the nature of code execution and is shared by all automated fault localization techniques that depend on code coverage, but it is still worthy to note.

As described in Section 5.4, FLAC uses JSCover to perform its code coverage analysis. JSCover performs instrumentation of the code that is executed to gather the coverage data. Instrumentation is the act of adding code to the original source file that is used to gather the execution data. Although instrumentation does not change the logical behaviour of the software, it does affect its performance negatively since more operations need to be carried out. Many test cases use timeout limits to determine whether they have passed or failed. A subtle decrease in performance caused by instrumentation can be enough to trigger such a test failure.

9.3 Future Work

As was discussed in Section 9.2, FLAC still lacks sufficient support in identifying lines that have been incorrectly removed. This is only partly solved by the concept of secondary files. One way of overcoming this issue could be to display the code coverage results from previous test runs and comparing it with the new ones. Doing this would allow one to easily see how the changes made have affected the execution and also identify incorrect removal of code. Furthermore it could be beneficial to see if the execution path for a given test case has changed between test runs. The greatest obstacle to overcome here would be the difficulties in visualizing these results.

FLACs current implementation does not indicate whether one or more test cases are likely to have failed due to the same fault. By applying a clustering algorithm that looks through the lines deemed suspicious for each test case, test failures that share similar code execution could be identified, further improving the overview of the FLAC results. Depending on how many of the failing test cases execute a given suspicious line, they could also be given a suspiciousness rating, similar to how the spectrum-based fault localization techniques operate.

Under current operational procedures at Qlik all tests are executed when changes have been made in the code but this is often unnecessary. The concepts that serve as the foundation for FLAC could potentially be applied in reverse as well. By performing the same type of comparison FLAC does, one could identify which test cases have been affected by a change and thereby reducing the amount of tests that need to be executed after a change has been implemented.

Chapter 10
Conclusion

In this master's thesis we have implemented and evaluated a new method of performing automated fault localization. The result of the implementation is a tool called FLAC, which uses code coverage and diff utilities to find suspicious code when test failures occur during test execution. FLAC does this by comparing the code that has been executed by the test that failed with the code that has been changed since it passed. To convey these results in an intuitive manner they are visualized by FLAC in a Qlik Sense application that displays the source code and highlights the suspicious lines.

In the tests we performed FLAC correctly identified all the injected faults that caused test failures. We also showed that on average, FLAC can reduce the search space of a diff by 87%, which is a strong indicator that FLAC can improve the fault localization efficiency. Furthermore, in the tests we performed FLAC generated and displayed the complete results in less than 90 seconds, which leads us to conclude that FLAC does not have any significant drawbacks connected to efficiency in time expenditures. Our interviews with developers and testers also indicated that FLAC will benefit their day to day work.

These results show the potential of our method in making this otherwise tedious and time consuming process easier and more efficient. More work is needed to get the full understanding of the potential and limitations however. The next step for us will be to incorporate FLAC into the daily activities at Qlik in order to evaluate its usefulness in the real world activites of the company. We will also work with the information that we summarized in Section 9.3 to see how FLAC can be improved further.

Bibliography

[1] BlanketJS. Open-Source. http://blanketjs.org/ Accessed: 2015-05-25.

[2] Concurrent version system. The CVS Team. http://www.nongnu.org/cvs/ Accessed: 2015-02-18.

[3] EclEmma. Open-Source. http://www.eclemma.org/ Accessed: 2015-03-09.

[4] Git. Open-Source. http://git-scm.com/ Accessed: 2015-02-18.

[5] Istanbul-JS. Open-Source. https://gotwarlost.github.io/istanbul/ Accessed: 2015-05-25.

[6] JSCover. Open-Source. http://tntim96.github.io/JSCover/ Accessed: 2015-03-09.

[7] Perforce. Perforce Software Inc. http://www.perforce.com/ Accessed: 2015-02-18.

[8] Qlik. Business Intelligence. http://www.qlik.com/us/company Accessed: 2015-03-09.

[9] Qlik. Qlik Sense Tour. http://www.qlik.com/se/explore/products/sense#%2Fse%2Fvideos%2Fhow-to%2Fsense-product-tour Accessed: 2015-03-09.

[10] Ranorex. Ranorex GmbH. http://www.ranorex.com/ Accessed: 2015-02-23.

[11] Rational Functional Tester. IBM. http://www-03.ibm.com/software/products/en/functional Accessed: 2015-02-23.

[12] Selenium. Open-Source. http://docs.seleniumhq.org/ Accessed: 2015-02-23.

[13] Stash. Atlassian. `https://www.atlassian.com/software/stash` Accessed: 2015-02-18.

[14] Wayne Babich. *Software Configuration Management*. Addison Wesley Longman, 1986.

[15] Anirban Basu. Assuring Software Quality with ISO 9126. *IT Managment*, (10):28–29, 2005.

[16] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using Machine Learning to Support Debugging with Tarantula. *18th IEEE International Symposium on Software Reliability (ISSRE '07)*, (10):137–146, 2007.

[17] Ilene Burnstein. *Practical Software Testing*. Springer Professional Computing, 2003.

[18] James S. Collofello and Larry Cousins. Towards Automatic Software Fault Location Through Decision-to-Decision Path Analysis. *Proceedings of the 1987 National Computer Conference*, page 539–544, 1987.

[19] Tai-Yi Huang, Pin-Chuan Chou, Cheng-Han Tsai, and Hsin-An Chen. Automated Fault Localization with Statistically Suspicious Program States. *LCTES '07 Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 11–20, 2007.

[20] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. *Proceedings. ICSE-16., 16th International Conference on Software Engineering*, (10):191–200, 1994.

[21] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. *ASE '05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[22] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. *ICSE '02 Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[23] David Loshin. *Business Intelligence: The Savvy Manager's Guide*. Morgan Kaufmann, 2003.

[24] Norman Matloff and Peter Jay Salzman. *The Art of Debugging*. No Starch Press, 2008.

[25] Manos Renieris and Steven P. Reiss. Fault Localization with Nearest Neighbor Queries. *Automated Software Engineering Conference, IEEE Computer Society*, pages 30–39, 2003.

[26] Rita L. Sallam, Bill Hostmann, Kurt Schlegel, Joao Tapadinhas, Josh Parenteau, and Thomas W. Oestreich. Magic Quadrant for Business Intelligence and Analytics Platforms. 2015.

[27] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[28] Vanja Tufvesson, Nicklas Erman, Markus Borg, Ander Ardö, and Per Runeson. Navigating Information Overload Caused by Automated Testing. *8th IEEE International Conference on Software Testing, Verification and Validation*, pages 1–9, 2014.

[29] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A Family of Code Coverage-Based Heuristics for Effective Fault Localization. *Journal of Computer Software and Applications*, 83(2):188–208, 2010.

[30] W. Eric Wong and Yu Qi. Neural Network-Based Effective Fault Localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(4):573–597, 2015.

[31] Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs. *SIGSOFT '02/FSE-10 Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, 2002.

[32] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating Faults Through Automated Predicate Switching. *Proceedings of the 28th International Conference on Software Engineering*, pages 272–281, 2006.

[33] Peter Zoeteweij, Rui Abreu, and Arjan J.C. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. *PRDC '06 Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

Appendices

Appendix A
Functionality Evaluation Material

Build Name	Change Location	Introduced Errors	Row Number	Correct	Incorrect
XTRAGSF-CIGQE36	(\assets\scripteditor\..	*\assets\scripteditor\scripteditor.js*	140	if (event.keyCode === 27) {	if (event.keyCode !== 27) {
		\assets\scripteditor\directives\qvfilebrowser\qvfile	20	selectedFileName: "=",	selectedFileName: "&",
			30	if (item.disabled) {	if (true) {
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
24	5	140, 20, 30			
25	0	-			SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE55	(assets\hub\..)	*\assets\hub\hub-factory.js*	134	for (var i = 0; i < isOutdated.apps.len	for (var i = 2; i < isOutdated.apps.length; i++) {
			369	scope.streamId = null;	scope.streamId = "12";
			164	isOutdated.streams = true;	isOutdated.streams = false;
		\assets\hub\content\content.js	150	scrollBarPos = scrollCont[0].getBoun	scrollBarPos = scrollCont[1].getBoundingClientRect().
			178	var authData = authentication.userDi	var authData = authentication.userDirectory + authent
			48	scope: true,	scope: false,
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
7	7	48, 369			
8	0	-	134, 164, 150, 178		SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE56	(\assets\storytelling\...	*\assets\storytelling\storytelling.js*	61	if (state === storyHistory.prevState	if (state === storyHistory.prevState && storyId === st
			87	if (State.state === State.States.play	if (State.state !== State.States.play) {
		\assets\storytelling\models\slide-model.js	35	rank = next.qData.rank - 1;	rank = next.qData.rank + 1;
			37	rank = prev.qData.rank + 1;	rank = prev.qData.rank - 1;
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
12	5	61, 87			
13	0	-	35, 37		SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE36	(\assets\qmc\...)	*\assets\qmc\modules\qmc-common\directive\qm*	46	if (timeoutPromise !== null) {	if (timeoutPromise === null) {
		\assets\qmc\modules\qmc-common-dialog\directi	49	return $scope.selectionSet.length ==	return $scope.selectionSet.length === 1;
			62	return item.defaultVirtualProxy !== tr	return item.defaultVirtualProxy === true;
		\assets\qmc\modules\qmc-custom-properties\ser	64	return self;	return null;
			146	enumerable: true,	enumerable: false,
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
26	5	49, 64, 146, 46			
27	5	62			
28	0				SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE56	(\assets\qmc\...)	*\assets\qmc\modules\qmc-filter\factory\qmc-filter*	175	dayInms = hourInms * 24;	dayInms = hourInms + 24;
			186	value = dateTime.stringify(getTimeS	value = dateTime.stringify(getTimeStamp(-(dayInm
		\assets\qmc\modules\qmc-section\service\qmc-s	168	if (exists) {	if (true) {
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
14	0	-	175, 186, 168		NONE
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE56	(\assets\qlikui\...)	*\assets\qlikui\components.js*	19	"assets/qlikui/components/qui-popov	REMOVED
		\assets\qlikui\components\qui-input-number\qui-i	36	$element.select();	REMOVED
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
15	10	19, 36, both files deemed "Secondary" and file hi			
					SUCCESS

Build Name	Change Location	Introduced Errors	Row Number	Correct	Incorrect
XTRAGSF-CIGQE55	(assets\qvangular\...)	*\assets\qvangular\qvangular-directives.js*	245	fn = $parse(data[1]);	fn = $parse(data[2]);
			390	restrict: 'A',	restrict: 'CDA',
			182	added	fn = "NaN";
		\assets\qvangular\qvangular-core.js	230	$element: element,	$element: link,
			255	added	return;
			127	if (!scope) {	if (scope) {
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
10	16	127, 230, 182, 245			
11	0		390, 255		SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE36	(\assets\client\...)	*\assets\client\utils\state.js*	51	previousState = currentState;	previousState = state;
			52	currentState = state;	currentState = currentState;
			171	return currentState;	return null;
		\assets\client\dialogs\confirm-dialog\confirm-dia	19	confirmLabel: "="	confirmLabel: "?"
		assets\client\views\app-overview\directives\app	47	if (!$scope.exp.ignoreOutsideClicks	if (true) {
			68	titleWatcher = null;	titleWatcher = titleWatcher;
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
13	49	51, 52, 171, 19, 47			
14	0	-	68		SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE56	(\assets\core\...)	*\assets\core\models\app.js*	84	if (!that.rpcOptions) {	if (that.rpcOptions) {
		\assets\core\utils\title.js	12	var _isDirty = false;	var _isDirty = true;
			51	ADDED	key = "This.error";
			56	ADDED	item = null;
		\assets\core\utils\resize.js	10	var delay = 250,	var delay = 25,
			16	isStarted = true;	isStarted = false;
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
4	113	10, 84, 12			
5	108	51, 56			
6	0	-	16		SUCCESS
Build Name	**Change Location**	**Introduced Errors**	**Row Number**	**Correct**	**Incorrect**
XTRAGSF-CIGQE55	(\assets\general\...)	*\assets\general\utils\color-cache.js*	13	stackMaxSize = 5000,	stackMaxSize = 50,
			62	} else if (arguments.length >= 3) {	} else if (arguments.length > 3) {
		\assets\general\components\scroll\scroll-area.js	43	var left = $scope.viewportRect.left,	var left = $scope.viewportRect.top,
			44	top = $scope.viewportRect.top,	top = $scope.viewportRect.left,
			103	case 33:	case 37:
			131	case 37:	case 33:
		\assets\general\utils\qv-longtapswipe.js	16	radiusThreshold: 10,	radiusThreshold: 1,
			17	timeThreshold: 510,	timeThreshold: 200,
			18	touches: 1,	touches: 25,
Run Number	**Test Failures**	**Repaired**	**Didn't cause failures**		**Result**
3	1	44, 103, 131, 13, 16, 17, 18			
4	0	-	62, 43		SUCCESS

Appendix B
Interview Questions

FLAC Evaluation Interview

Name: ________________________________
Position: ______________________________

Introduction

Before answering the interview questions, the participant was given a brief introduction to what FLAC is and how it is intended to be used. The participant was given the task to investigate a set of test failures that we had purposefully created by injecting the source code with faults. After experimenting with FLAC the following questions were asked in order to evaluate FLACs usability. Follow-up questions not part of the numbered questions were also used when necessary.

Questions

1. What was your standard procedure for performing fault localization when investigating test failures before trying FLAC?

   ____
   ____
   ____

2. Was it easy to understand the purpose of FLAC?

   ____
   ____
   ____

3. Were the results presented in a manner that was intuitive and simple?

   ____
   ____
   ____

4. Did you find FLAC useful?

   Yes ☐ **No** ☐

5. Why/why not?

   ____
   ____
   ____

6. Do you think FLAC could improve your efficiency when performing fault localization?

   Yes ☐ **No** ☐

7. Is there any functionality or information that you think FLAC lacks?

____

____

____

8. Other comments?

____

____

____

Appendix C
Interview Results

ID: P1
Position: Triad Technical Lead

1. Read logs, look at screenshots, check NIOCAT information, look at information from web sockets, compare the results with older successful test, manually check diffs.

2. Yes, conceptually easy to understand, but the Sense application itself needs clarification. FLAC has lots of potential applications. FLAC indicates too many unnecessary lines however.

3. Needs more textual information to guide the user in the Sense application. Clearly define what a "suspicious" line means and why it's not always so interesting. It is easier to understand the test cases if a description of the test can be shown in FLAC, because the test name does not always describe what it does.

4. Yes

5. It was easy to find the fault in the test example and it would likely have been very difficult otherwise. It is hard to give some exact numbers.

6. Yes

7. It can be hard to understand what suspicious means right away so the information should be more intuitive. When you look in a file for the first time it could be hard to understand that the suspicious line could be hidden at line which is at the bottom of the file and is not intuitive for the user to scroll to find the line.

8. Want to apply to manual testing, want to do more with code coverage results, want to see the execution path of a test.

ID: P2
Position: Triad Technical Lead / Strategic Prototyper

1. Analyzing diff history, decreasing the delta, going back to see when it worked as intended.

2. Yes.

3. Divided, needs textual information and/or examples to describe what the items mean. An editor might have been better than displaying the results in Qlik Sense.

4. Yes.

5. Limits the search space, easy to exclude the client when no files are suspicious, doesn't demand knowledge of how the code works.

6. Yes.

7. The ability to point to which commit is responsible for the suspicious line.

8. Make FLAC an extension in Sense, remove the Java application from the workflow, clustering, compare test case uniqueness, the ability to only look at diff information.

ID: P3
Position: Test developer

1. Analyzing Horsie logs, NIOCAT logs, video recordings
2. Definitely.
3. Yes, but textual information could help and also clustering of test cases with similarly suspicious lines.
4. Yes.
5. FLAC quickly found the issue.
6. Yes.
7. Clustering, more context around the lines marked as suspicious.
8. Using FLAC to exclude the client from fault localization will be beneficial.

ID: P4
Position: Test developer

1. Building and executing tests locally, checking logs, checking commits and giving it to developers for further inspection.
2. Definitely.
3. Yes, it was clear what was a suspicious line and which they were.
4. Yes.
5. It's a good tool, developers from the team will likely find it very beneficial.
6. Yes.
7. Clustering of test cases, commit analysis for suspicious lines.
8. FLAC could be used for test case developing as well, i.e. checking how much of a new functionality is actually covered when developing test cases.

ID: P5
Position: Internal App Developer

1. Analyzing error logs, bisection method to narrow search space
2. Absolutely.
3. Users are often accustomed to seeing the whole screen immediately so FLAC can be confusing at first. Textual information and guidance for what to do would help a lot.
4. Yes.

5. Gives a hint of where to start looking.

6. Yes.

7. Needs more guidance for the user. It would be beneficial to see number of suspicious lines before choosing a file. Both total number of suspicious lines for all files and the number in each individual file.

8. Make it easier for the user to find the suspicious lines and also rank them for "how" suspicious they are.

ID: P6
Position: Test developer

1. Check error message from Bamboo, check logs and images, build and execute tests locally.

2. Quite easy, but files labeled as secondary were confusing.

3. The work flow was easy to understand, but on initial opening of the app the empty windows made it feel like something was wrong.

4. Yes.

5. It was very useful to be able to see the source code.

6. Maybe, I'm not a developer and the question would be better suited for them.

7. Want to see files covered of how many? Seeing and sorting from files not covered as well. Attaching files and test cases to modules, for example what tests cover this functionality and which files are covered by that test?

8. Integrate FLAC with Qliks internal quality portal.

Appendix D
User Guide

FLAC

Guide – How to use

Introduction

FLAC – short for "Fault Localization Automation using Code Coverage and Version Control Diff Utilities" is a tool we've developed to assist the fault localization process at Qlik. When a test case in Bamboo suddenly starts failing even though it passed without issue in the past, the culprit can be a change or addition in the client JavaScript code. FLAC is a two-part tool, the first is a java application that automatically compares the code coverage results from a given test run with arbitrarily recent changes to Git, i.e compares *what has been executed in a test case* with *what has changed since it passed*, thereby finding code deemed suspicious and likely to contain the fault that caused the test case to fail. The second part is a Qlik Sense application that displays the source code executed in the test case you want to investigate and highlights the code that FLAC has deemed suspicious. Our hope is that this will assist in your fault localization efforts when investigating test failures.

How to use

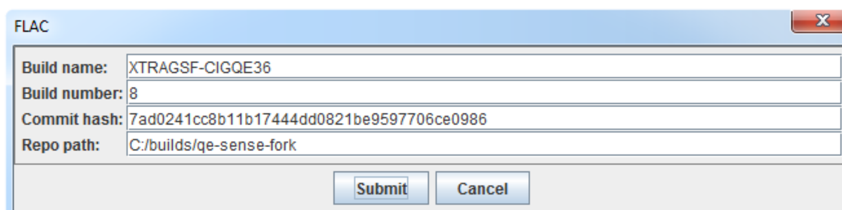
Step 1 – Executing test runs with code coverage

The Qlik Sense source code now includes the option to collect code coverage data. The first step of using FLAC is to branch out from the revision you want to analyze, and for your new branch open the file: *"\prod\client\Client\src\BuildClient-Client.proj".*
In this file locate the parameter:
*<Instrumented Condition="'$(Instrumented)'==''">**false**</Instrumented>*
Change the parameter to true:
*<Instrumented Condition="'$(Instrumented)'==''">**true**</Instrumented>*

Now simply push this change to Git and start a new test run.

Step 2 – Performing the FLAC comparison

Once your new test run has finished, the next step is to perform the FLAC comparison. Launch the java application "flac.jar" and the following window will appear:



Build name – This is the abbreviated name of the branch that has executed the code coverage run. If you go to the test run in Bamboo, the URL will look something like:

"*http://bambooctrl2.rdlund.qliktech.com/browse/**XTRAGSF-CIGQE57**-9*"
What we want is the highlighted section.

Build number – This is the build number for the test run we're looking into. If you're at the test run in Bamboo, it will appear as the final number in the URL above (9 in this case).

Commit hash – The starting point for the diff that FLAC uses is the commit associated with the build number, but how far back you want FLAC to check is up to you. The more recent the commit is, the less false positives you will risk FLAC finding. However, if you choose a commit that is more recent than when the issue was introduced (and please be aware that some issues are intermittent so "latest green build" is not always an accurate point), FLAC will not find it. In short, choose a commit that is as recent as possible but still guarantees it containing the fault. You can always re-run FLAC with another commit if the results are not satisfactory.

Repo path – This is the path to your local sense folder, FLAC needs this path in order to be able to retrieve the Git diff. Please note that you must have the correct branch checked out, and that it must be up-to-date, otherwise the diff retrieval will fail.

Step 3 – Visualizing results in Qlik Sense

To import the results from Step 2 into Sense, launch the app "flac.qvf", go to "Data Load Editor" and hit "Load Data". Once it's completed, click "Open Hub" and finally open the FLAC sheet. To the left you will find all the test cases included in the run and an indication of whether they Passed or Failed. Click the test case you want to investigate and a list of all files from the source code that were executed at some point during the test will appear. These are sorted by 3 parameters:

Suspicious – The file contains code that has been both executed and recently added/modified.
Secondary – The file contains code that has been executed and code that has been recently added/modified/removed, but they do not overlap. This could for example mean that code has been mistakenly removed.
Covered – The file contains code that has been executed.

Once you click on a file you want to investigate further, the app will display the full source code in the right column, indicating which lines are Suspicious, which have been Covered and which have not been touched at all. When performing an investigation first look into lines that are Suspicious, if you cannot find any issues move on to Secondary files. Here you will unfortunately be forced to go to Stash and look at the file history to see which commit could be the culprit. If neither of these provides any valuable results, chances are that the test failure is not caused by a JavaScript being changed, but rather environmental issues or server side changes.

Examensarbete Fault Localization Automation using Code Coverage and Diff Utilities
Studenter Jonas Svalin, Axel Hildingson
Handledare Emelie Engström (LTH), Lars Andersson (Qlik)
Examinator Per Runeson

Att göra fel är lätt, att hitta fel är svårt

“Vad gick fel?” Denna svårlösta och tidskrävande fråga ställer vi oss nästan dagligen i olika sammanhang. Mjukvaruutveckling är inget undantag och därför har vi skapat FLAC, ett verktyg för automatiserad fellokalisering i kod.

I dagens IT-revolution så är hastighet och komplexitet faktorer som alla IT-företag slås med. Kunderna vill ha komplexa och intelligenta system som skall utvecklas och levereras snabbare än någonsin, men i kontrast till detta vill de också att de system som levereras ska hålla en högre kvalité än någonsin. För att kunna möta kundernas behov av kvalité och stabilitet så har automatiserad testing börjat användas i allt större utsträckning. Verktyg för automatiserad testning gör det möjlighet för utvecklare att testa sina system genom simulerade tester som körs via script. Dessa går betydligt fortare att utföra än att en person manuellt skall göra dem och det ger också möjlighet att exakt återupprepa testerna. I och med detta så kan man idag genomföra ett stort antal tester för varje förändring som sker i ett system. När testfall inte passerar som man förväntat sig så återkommer dock frågan: “Vad gick fel?” För att enklare hitta svaret på den frågan har vi arbetat med företaget Qlik, ett världsledande företag i sektorn ”Business Intelligence” som utvecklar applikationer för visualisering av stora mängder data för att förenkla affärsanalys. Tillsammans har vi utvecklat FLAC, ett verktyg som automatiserar den här processen.

FLAC – automatiserad fellokalisering

Premissen bakom FLAC är simpel, om ett test tidigare fungerade som det skulle men nu plötsligt har slutat fungera så bör koden som är ansvarig för detta finnas i det misstänkta snittet mellan koden som har körts i testet och koden som har ändrats sen det fungerade. FLAC analyserar precis det här och presenterar resultaten i visualiseringsverktyget Qlik Sense. All källkod presenteras för användaren och den misstänkta koden markeras tydligt ut.

I testerna vi utfört har FLAC hittat samtliga fel som har introducerats. Vi har dessutom kunnat visa att FLAC kan reducera antalet potentiella förändringar som kan ha påverkat testfallet med upp till 95% och all denna analys tar mindre än 90 sekunder att utföra. Ursprungsfrågan “Vad gick fel?” blir nu betydligt lättare att lösa och resurser som tidigare hade spenderats på manuell fellokalisering kan nu istället fokuseras på utveckling vilket gör det möjligt för dagens IT-företag att fortsätta trenden med snabbare utveckling, mer komplexa system och ökad kvalité.