CHALMERS

Fix Cache Based Regression Test Selection

Master of Science Thesis in the Programme Communication Engineering

Authors: Zhe Wang

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, March 2010

Fix Cache Based Regression Test Selection

Zhe Wang

Examiner: Christer Carlsson
Industrial Supervisor: Greger Wikstrand

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2010

Abstract

Regression testing is a crucial step in the software development process, which ensures the quality of software systems by detecting whether new faults have been introduced into previously tested code. Regression testing becomes costly as more and more regression test cases are created. Regression test selection, which selects a sub-set of the available regression test cases based on different criteria, is a well-known method to reduce test scope and improve the efficiency of regression testing

This paper presents a new method, named fix cache based regression test selection, which computes test case coverage based on what files were updated to fix faults found by the test cases. Our method uses a cache to monitor the most fault-prone files and recommends test cases related with continuously updated files. The method is useful for predicting new faults and selecting the most fault-prone test cases for automatic regression testing.

The thesis explores the concepts and processes for how to implement and evaluate this method.

We have implemented the method and evaluated it during two months' period in a large, industrial, embedded, real-time software system. Our results show that the fix cache based selection method is effective with reaching weekly cache hit rates in the range 50%-80% for a fully automatic regression testing.

Acknowledgements

First of all I would like to thank my supervisor at Ericsson AB, Dr. Greger Wikstrand for giving me the opportunity to work on this exciting topic and thank you my thesis partner Jeevan Kumar Gorantla for giving me good suggestions during our thesis cooperation. A special note of gratitude goes to Lars Johansson and Dr. Johan Natt och Dag for their wonderful support during the entire duration of the thesis work. I would like to thank Thomas Andersson, Andreas Ekberg for providing valuable insights for the project.

And last but not the least I would like to thank my family and friends for their unconditional and amazing support throughout my thesis.

Lund, March 2010

Zhe Wang

CONTENTS

1 Introduction

1.1 Research Context

Ericsson Mobile Platforms mission is to be a leading mobile platform supplier, with complete and optimized solutions in all segments, making their customers truly successful. Ericsson offers a complete platform portfolio across 2.5G and 3G to manufacturers of mobile phones and other wireless devices such as mobile handsets and PC cards. The technology is based on Ericsson's global standardization leadership and the world's strongest IPR (intellectual property rights) portfolio.

Regression testing is a crucial step in the software development to ensure that modifications do not break previously working functionality [6, 25]. However, regression testing is often expensive and time consuming as more and more regression test cases are created. The number of regression test cases can be very large, e.g. including tens of thousands of test cases, requiring days or weeks to execute [5]. Regression test selection, regression test prioritization and regression test reduction are three methods to improve the efficiency of regression testing.

Regression test selection methods select a sub-set of the available regression test cases based on different criteria, for instance, coverage of changed code. Regression test prioritization orders the available test cases so that the most important ones are executed first, but the initial failure detection rate is maximized. The underlying assumption being that this approach enables developers to start working on bugs as soon as possible. Regression test reduction permanently removes test cases which no longer apply to the changed software or requirements.

Developers at Ericsson build a system for advanced automatic regression testing based on a large amount of available regression test cases. The regression test cases of real-world and large-scale systems in Ericsson are selected manually by human experts based on tester working experience today, which is both time-consuming and cost-consuming. The task to be performed in this thesis is to implement and evaluate a method for automatic selecting the most fault-prone regression test cases in order to make regression testing more efficient.

1.2 Background

Regression testing is a traditional testing technique that has been given much research interest over the years [6]. Regression testing provides the only reliable means to verify that code base changes and additions from version N to version N+1 don't "break" an application's existing functionality, and it can have the single greatest impact in controlling product release delays, budget overruns, and the prospect of

software errors slipping into released/deployed products. The purpose of regression testing is to detect unexpected faults — especially those that occur because a developer did not fully understand the internal code correlations when modifying or extending code. Every time code is modified or used in a new environment, regression testing should be used to check the code's integrity.

Regression testing is an important step in the software development. Regression testing aims to reduce the cost of developing and maintaining code, helping build confidence in its correctness and to improve its reliability. Two major types of regression testing are corrective regression testing when the specification has not been modified, and progressive regression testing when the specification has been modified [7]. In this paper our focus is on the corrective regression testing.

Regression testing is one of the most commonly used testing technique and many companies develop their own regression testing tools [2]. Since regression testing becomes costly as more and more regression test cases are created in real-world, large-scale and industrial software systems [21], regression test selection, regression test prioritization and regression test reduction are three good methods to be used for predicting and selecting the minimal set of fault-prone test cases in order to improve the efficiency of regression testing [5, 10].

Some authors have presented their processes which support the regression testing for large software systems [2, 21]. However, with the increasing size of software systems and the increasing efforts spent on developing automatically executable test cases, it is not clear that rerunning all of test cases will continue to be a viable strategy.

White and Robinson [10] presented empirical results on evaluating different regression testing techniques on several large industrial systems. They focused on the use of testing firewalls to group different source codes together so that regression tests can be minimized by only needing to rerun tests for interactions over the group limits. In paper [10] they show that a subset of test cases which are selected over group limited is more efficient and less cost-consuming for rerunning in regression test comparing with rerunning all of test cases on a large industrial system. That means regression test selection on a large industrial system is good for improving the efficiency of regression testing and reducing the cost of rerunning test.

1.3 Aims and Objectives

The regression test cases of real-world and large-scale systems in Ericsson are selected manually by human experts today based on tester working experience. Ericsson has no available method for effectively automatic selecting a subset of most fault-prone regression test cases from the large amount of available test cases. In order to reduce the cost and improve the efficiency of regression testing, the aim of our thesis work is to implement and evaluate a test selection method for automatically selecting the most fault-prone test cases and predicting future faults for regression testing.

Cache algorithms, code-coverage based algorithms and risk based algorithms are three various types of algorithms based on regression test selection methods and they have been investigated by different authors in the previous literatures. These algorithms are used for predicting and selecting a minimal set of fault-prone test cases on large software systems in order to improve the efficiency of regression testing. These three various types of algorithms will select a subset of most fault-prone test cases based on different criteria and use this selected subset of fault-prone test cases in next regression testing cycle. In this paper, we will select a suitable algorithm out of these three algorithms at first. And then we will evaluate the algorithm in order to automatically select the most fault-prone regression test cases based on all available test cases for one Ericsson special product.

1.4 Thesis Outline

The rest of the report is structured in the following way:

- Chapter 2 describes the analysis of three different algorithms based on Ericsson testing environment and then concludes about which algorithm is most suitable to be used for our thesis work. This chapter explains the algorithms in a more detailed manner and compares them with their advantages and disadvantages.
- Chapter 3 introduces the Ericsson tools which will be used in the implementation of our test selection algorithm.
- Chapter 4 describes algorithm design and algorithm implementation with Ericsson Tools -- the processes of pre-fetch cache, cache update and the way of finding the fault-prone test cases corresponding to the most fault-prone source files in the fix cache.
- Chapter 5 analyzes the result of implementing the fix cache algorithm in Ericsson product and compares the hit rate with different linear combinations of pre-fetching the cache.
- Chapter 6 discusses the potential for future enhancements to the thesis work.

2 Algorithm Selection

In this chapter, we will compare three types of algorithms corresponding to their advantages and disadvantages and then decide which algorithm is most suitable for the testing environment in our application.

In the majority of software projects, time and people available are not sufficient to eliminate all faults before a release. Any technique that allows software engineers to reliably identify the most fault-prone software functions or files provides several benefits. It permits available resources to be focused on the functions or files that have the most faults.

Two important qualities of software fault prediction algorithms are accuracy and granularity. The accuracy is the degree to which the algorithm correctly identifies future faults. The granularity specifies the locality of the prediction. Typical fault prediction granularities are the executable binary [13], a module (often a directory of source code) [1], or a source code file [21]. For example, a directory level of granularity means that predictions indicate a fault will occur somewhere within a directory of source code.

Cache algorithms, code-coverage based algorithms and risk based algorithms are three various types of algorithms based on regression test selection methods and they have been investigated by different authors in the previous literates. These algorithms are used for predicting and selecting a minimal set of fault-prone test cases on large software systems in order to improve the efficiency of regression testing. These three various types of algorithms will select a subset of most fault-prone test cases based on different criteria and use this subset of fault-prone test cases in next regression testing cycle. Therefore, regression testing will be more efficient and less cost-consuming with implementing regression test selection method.

Since a typical system developed by Ericsson has tens of thousands of source code files and today regression test cases of real-world and large-scale systems in Ericsson are selected manually by human experts, it is quite time consuming and cost consuming. In order to reduce the cost and make regression testing more efficient, our thesis will be focused on how to evaluate algorithms to predict the most fault-prone source codes at file level. The most fault-prone test cases which can predict more future faults during software regression testing will be selected corresponding to these most fault-prone source code files. Therefore, the accuracy of algorithms at file level is the most important factor to be considered when we select which algorithm will be used in our application.

2.1 Cache Algorithm

The cache algorithm [16, 17], developed by Kim et al, is a fault prediction algorithm, which is executed over the fault history of a software product. The algorithm uses the insight that faults do not occur uniformly in time across the history of a file, but instead appear in bursts.

There are four different kinds of locality where faults frequently occurrences:

- Changed-file locality: If a file was changed recently, it will tend to introduce faults soon.
- New-file locality: If a file has been added recently, it will tend to introduce faults soon.
- Temporal locality: If a file introduced a fault recently, it will tend to introduce other faults soon.
- Spatial locality: If a file introduced a fault recently, relative files (in the sense of logical coupling) will also tend to introduce faults soon.

Following Hassan and Holt [1], Kim et al borrowed the notion of a cache from operating systems research, and applied it for the purpose of fault prediction. The cache algorithm employs only a small subset of the product's files that are most fault-prone (as Kim et al used 10% of the files in their experiments). The cache is a convenient mechanism for holding fault history as a dynamically updated list of the most fault-prone files, and for aggregating multiple heuristics for maintaining the cache.

The cache is a dynamically updated fault history for predicting future faults, and contains the source code files that are for the moment most fault-prone. The switch to a cache involves a subtle but important shift. Instead of creating mathematical functions that predict future faults, the cache selects and removes files based on specific criteria. By consulting the cache after a fault has been fixed [11, 12], developers can predict most-likely fault-prone files. Therefore, the cache algorithm is useful for making a more targeted allocation of available resources on the most fault-prone files.

The cache size can be adjusted based on the number of resources available for testing or verification. A typical cache size is 10% of the total number of source code files, since this provides a reasonable tradeoff between cache size and the accuracy of the cache algorithm. Larger cache size results in higher hit rate which means better prediction for future faults, but the faults will spread out over a greater number of source code files, and it will be difficult to select the real most fault-prone source code out of a large mount of source files. The ideal is to minimize the size of cache while at the same time maximizing the accuracy of the cache algorithm.

The basic process of the cache algorithm at file level is as follows.

There are two main parts of the cache algorithm execution -- Cache Initialization and Cache Operation.

Cache Initialization:

Step 1: Modified source code files are extracted by mining a product's version archive and fault database.

Step 2: Pre-fetch the cache (e.g. files with the largest lines of source code) in the initial product revision, creating the initial state of the cache.

Cache Operation:

Step 3: If at revision *n*, the fault has been fixed in a source file, probe the cache to see whether the corresponding source file is present in the current cache. If yes, count a hit, otherwise a miss.

Step 4: If the source file is missed which is not contained in the current cache revision *n*, fetch the source file considering as fault-prone files due to temporal locality and spatial locality into the cache for future fault predictions starting at revision *n+1*.

Step 5: Also at revision *n*, fetch source files that have been created due to new-entity locality and modified due to changed-entity locality since revision *n-1*. (Optional step)

Step 6: Since the size of the cache memory is fixed, we have to remove files, which are selected using a cache replacement policy.

Step 7: The hit rate is computed by the formula:

$$hitrate = \frac{\#of\ hit}{\#of\ hit + \#of\ miss}$$

Step 8: Iterate over steps 3-7 to cover the existing modified source files and fault history.

The higher hit rate is, more future faults are possible to predict with the updated cache.

The cache algorithm is a dynamically updated method which can make an automatic way for selecting test cases, which is good for automatic selecting fault-prone test cases from a large amount of test cases. The cache algorithm doesn't require testers to access and understand the source code files, this is convenient and time-consuming.

The experimental assessment on seven open source products done by Kim et al showed that the cache algorithm was 73%-95% accurate at predicting future faults at the file level and 46%-72% accurate at the entity level with optimal options [17].

2.1.1 The Fix Cache Algorithm

In [17] Kim et al give two variants of the cache algorithm, the bug cache algorithm and the fix cache algorithm. We will primary describe the fix cache algorithm because Ericsson has no tools which can support for implementing the bug cache algorithm.

The fix cache algorithm will dynamically check and update the cache when a fault has been fixed as shown in Figure 1. The cache contains file names of the modified source codes. The fix cache algorithm is a deployable fault prediction algorithm which shows how to turn localities into a practice fault prediction model.

As shown in Figure 1, the fix cache algorithm will check and update the cache when the fault has been fixed in the source code at t_{fix}.

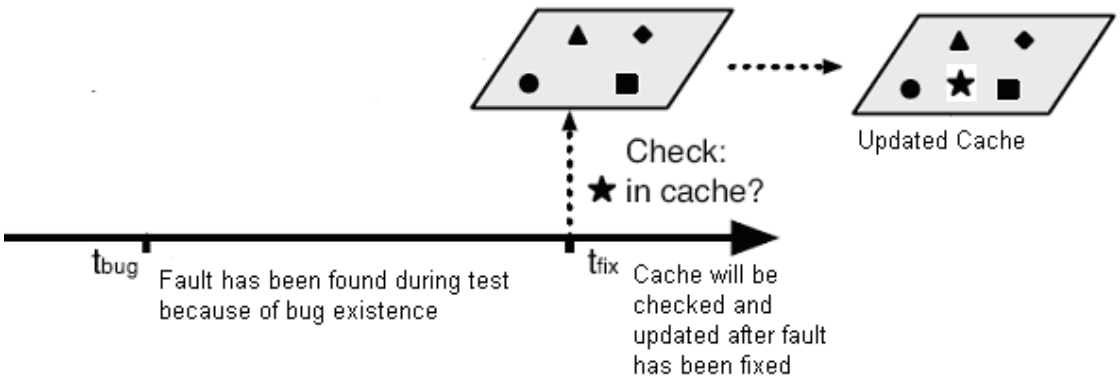

Figure 1. Fix Cache Algorithm

In Figure 1, t_{bug} is the time when a bug was introduced. A bug is a mistake in the source code which maybe leads to a fault. A fault is a situation when the testing result of the source code is an uncorrected result, which will be aware of a bug existence in the source code. A fault will be found during test after t_{bug}. t_{fix} is the time when the mistake in buggy code has been corrected and the fault has been fixed in the new version of source code.

In Figure 1, the cache is illustrated as rhombus. The cache is a fault history which contains a collection of the modified source code files. The “star”, “circle”, “square”, “triangle” and small “rhombus” in Figure 1 are the names of different source code files, which have been modified due to the existence of bugs and the occurrence of faults. Arrows shown in Figure 1 describe the direction about how the cache updates. At the time t_{fix}, the bug in “star” has been fixed, where “star” is the file name of the modified source code, the fix cache algorithm will check if file “star” is in the current cache. Since file “star” is not in the current cache, the cache will be updated and it will include the changed source code file “star” into the cache. In Figure 1, I only show an easy example to explain how the cache updates with modified file but not consider the cache replacement policy. If the modified source code file is “circle” instead of “star” as in the example shown in Figure 1.1, the cache will not update because file name “circle” is already contained in the cache.

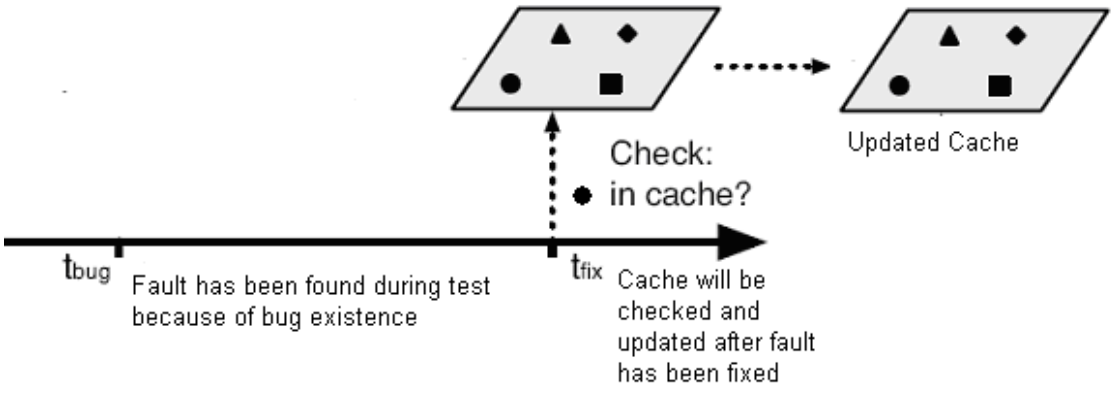

Figure 1.1. Example of Fix Cache Algorithm Update

The hit rate is computed at time t_{fix} by:

$$hitrate = \frac{\#of\ hit}{\#of\ hit + \#of\ miss}$$

If the name of a changed source code file is already in the current cache as the name “circle” in Figure 1.1, it counts as a hit, otherwise it counts as a miss (which is the case with the name “star” shown in Figure 1).

In order to reduce the number of “missed” file names regarding to hit rate computation, we can pre-fetch a subset of potential fault-prone source code files (i.e. according to the greatest lines of code which it is much more possible to have bug in the source code) when initializing the cache.

When the cache is full, the algorithm must unload some files based on the cache replacement policies before it can load other new modified files in order to keep the required cache size. The cache replacement policy will be described in Chapter 4.

Advantage of cache algorithm:

- The algorithm has high accuracy at predicting faults (accuracy at file level 73%-95%) [16, 17].
- The cache model is dynamic and is able to adapt quickly to new fault distributions, since the fault occurrences directly affect the model.
- The algorithm is useful for prioritizing verification and validation resources on the most fault prone files or entities.

Disadvantage of cache algorithm:

- Never estimate the cost-efficiency for company software development

2.2 Coverage Based Algorithm

Code-coverage based algorithm is one of the test case prioritization and test case selection techniques [15, 19] based on the coverage of the code elements, which aims to improve the effectiveness of regression testing by ordering the test cases, so that the most beneficial test cases with highest priority are selected and executed first. In order to achieve code coverage at the fastest rate possible in the initial phase of regression testing which is considered as the aim of speedy achievement of coverage, scheduling test cases helps us to reach 100 percent coverage soonest or to reach the maximum possible coverage.

Formally, a test case prioritization problem is defined as follows [15]:

Given: T, a subset of test cases; PT, the set of all permutations of T; and f, a function from PT to the real numbers.

Problem: Find T'∈ PT such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$$

Where, T' is a subset of test cases which has highest priority, PT represents the set of all possible prioritizations (orderings) of T and f is a function that, applied to any such ordering, yields an award value for that ordering.

The code-coverage based algorithm requires testers to access and fully understand each element in the source code in order to reach the speedy achievement of coverage, even maximum possible coverage. This is quite hard work for testers and it is time-consuming, especially if the number of test cases is very large.

Greedy search technique is one optimization techniques which can be used for code coverage based algorithm as shown below. Greedy search technique may produce suboptimal results because they may construct results that denote only local minima within the search space. We will investigate three different algorithms based on Greedy search technique.

The Greedy algorithm is an implementation of the "next best" search philosophy. It works on the principle that the element with the maximum weight is taken first, followed by the element with the second-highest weight and so on, until a complete but possibly suboptimal solution has been constructed. Greedy search seeks to minimize the estimated cost to reach some goal, which is simple but in situations where its result are of high quality it is attractive since it is typically inexpensive both in implementation and execution time.

Below is one example about how to calculate the weight and how to select the test cases with the Greedy algorithm.

Assume we have eight elements in the source code (E1-E8) and four test cases (TA-TD), which are related as shown in Table 1. The test cases are arranged in code coverage order, i.e. after how many elements of source code they cover. In other words each test case has a weight, which is equal to how many elements the test case covers. The Greedy algorithm will select test case TA first, since it covers six elements of source code and thus has the highest weight. Test case TB, which covers five elements, will be selected next due to the second-highest weight-test case TC and TD cover the same number of elements, therefore the Greedy algorithm could sort these four test cases either TA,TB,TC,TD or TA,TB,TD,TC. The order of the test cases given from the algorithm will be the execution order of test cases. If we want to select the top 50% of the total number of test cases which have the highest weights to execute in regression testing instead of executing all test cases, the Greedy algorithm will select test cases TA and TB out of four test cases in this example.

Test Cases	E1	E2	E3	E4	E5	E6	E7	E8
TA	x	x	x			x	x	x
TB	x	x	x				x	x
TC	x	x	x	x				
TD					x	x	x	x

Table 1. The example of Greedy algorithm

The additional Greedy algorithm is a kind of the Greedy algorithm, but with a different strategy. It combines the feedback from previous selections in order to achieve 100% coverage at first. It iteratively selects the maximum weight element of the problem from that part which has not already been consumed by previously selected elements.

If using the additional Greedy algorithm on the example shown in Table 1, test case TA will be selected first, since it has the highest weight. The uncovered elements are E4 and E5. Test case TB is shipped since it covers neither element E4 nor E5, thus it is not helpful for achieving the maximum coverage. Test case TC and TD cover elements E4 and E5 respectively. Each test case covers only one uncovered element. Thus, the Additional Greedy would execute test cases as the order of either TA, TC, TD, TB or TA, TD, TC, TB.

The 2-Optimal (Greedy) Algorithm is an instantiation of the K-optimal Greedy Algorithm when K=2. The K-optimal algorithm selects the next K elements that, taken together, consume the largest part of the problem. The 2-optimal approach is choosing the pair of test cases first which have the maximum coverage.

If using the 2-Optimal Greedy algorithm on the example shown in Table 1, test cases TC and TD will be executed at first since the combination of test cases TC and TD covers all of eight elements which will achieve the maximum coverage among all pairs of test cases. Therefore, the order of test cases execution will be TC, TD, TA, TB.

Advantage of coverage based algorithm:

- Accuracy at predicting faults at file level is 40%-50% better than manual regression test selection [15,19], where the hit rate of manual regression test selection is around 30.5% in Ericsson.
- Speedy achievement of code coverage.

Disadvantage of coverage based algorithm:

- Code-coverage based algorithms are very difficult to manage thus to all the information obtained from code. It is also hard to create corresponding management matrices when testing a larger or more complex component. Code-coverage based algorithms are not suitable for testing larger components at more abstract levels.
- Require testers to access and to understand the code, which is very time-consuming.
- Language-dependent, it will make the situation more complex.

2.3 Risk Based Algorithm

Risk based algorithm [23] is one kind of safety regression test [4], which ensures that the potential problem areas are properly handled based on risk analysis and assures that the remaining faults will not bring serious failures. There are two important parameters to be considered in this risk based algorithm: severity probability (P) and cost (C). A subset of test cases which have highest value of Risk Exposure will be selected, where Risk Exposure is the multiplication of Cost and Severity probability as formula RE=P*C.

Cost (C) means the cost of the requirements attributes that this test case covers. Cost is categorized on a one to five scale, where one is low and five is high. Cost for the top 20% of test cases with the highest cost will be five and cost for the bottom 20% of test cases will be one. Two kinds of costs will be taken into consideration: the consequences of a fault as seen by the customer, for example, losing market share because of faults; the consequences of a fault as seen by the vendor, for example, high software maintenance costs because of faults.

After running all of the test cases in a subset of test cases group, we can sum up the number of faults uncovered by each subset. Severity probability (P) is computed based on multiplying the number of faults by the average severity of faults. It falls into a zero to five scales, where zero is low and five is high. For each test case without any fault, severity probability (P) is equal to zero. For the rest of test cases, severity probability for the top 20% of test cases with highest estimate severity probability value will be five, and severity probability for the bottom 20% of test cases will be one.

2.3.1 Two Kinds of Risk Based Regression Test Selection

- Model-based selection (for each test case) :

  Step 1: Estimate the cost for each test case.

  Step 2: Derive severity probability for each test case.

  Step 3: Calculate Risk Exposure for each test case.

  Step 4: Select test cases that have the highest values of Risk Exposure as safety tests.

- Risk-based end-to-end test scenario selection (more customer-directed):

  Since end-to-end scenarios involve many components of the system working together, they are highly effective at finding regression faults. There are two rules of selection strategy.

R1: Select scenarios to cover the most critical test cases.

R2: Ensure scenarios cover as many test cases as possible.

Step 1: Calculate Risk Exposure (RE) for each scenario:

Since scenarios consist of test cases, we can simply calculate the Risk Exposure for each scenario by summing up the Risk Exposure for all test cases that this scenario covers according to the traceability matrix. Traceability matrix [23] is a matrix shown the relationship between the test suite and its risk exposure. If one scenario consists of n test cases, the mathematical formula for RE(s) is: $RE(s) = \sum RE(t)_i$, where $1 \le i \le n$ and test case i is covered by this scenario.

Step 2: Select the scenario with highest RE(s)

Step 3: Update the traceability matrix and re-build:

When running the chosen scenario, all test cases covered by the scenario will be executed. According to our selection rules, these test cases should not affect our selection any more. We should focus on those test cases that have not yet been executed.

After the chosen scenario has been executed, we cross out the column for the chosen scenario, and rows for all the test cases covered by the scenario.

Step 4: Repeat Steps 2 and 3 as desired.

Advantages of risk-based selection:

- Cost-efficiency for company software development.
- More customer-directed (especially in the case of test scenario selection).
- Accuracy at predicting faults at file level is around 52.6% better than manual regression test selection [23].

Disadvantages of risk-based selection:

- It is difficult to estimate the cost for each test case in our application.
- There has no fault history and we can not trace back to find bug-introducing change.

2.4 Summary of Algorithm Analysis

We have investigated and compared three various types of algorithms for regression test selection -- cache algorithms, risk based algorithms and code-coverage based algorithms. Regression test selection can be used for improving the efficiency of regression testing. Cache algorithms select a sub-set of most fault-prone regression test cases, which is useful for predicting new faults based on the dynamically updated cache as fault history. Risk based algorithms order the available test cases so that the most critical ones which can detect more faults in the software (with highest risk exposure) are executed first. Code-coverage based algorithms aim to improve the effectiveness of regression testing by ordering the test cases and reaching 100 percent coverage soonest, so that the most beneficial test cases with highest priority are executed.

Based on the experimental results from conference papers, the comparison between different regression test selections is shown in Table 2.

	Fix Cache Algorithm based Regression Test Selection	Risk-based Regression Test Selection	Coverage-based Regression Test Selection	Manual Regression Test Selection
Require of test cases prioritization	No	Yes	No	No
Require of code instrumentation	No	No	Yes	No
Accuracy at file level	73%-95%	52.6%	40%-50%	30.5%

Table 2. Comparison of different algorithms based regression test selection

Since our thesis work consider real-world, large-scale and industrial software systems with large number of testing components, a code-coverage based algorithm is not suitable for our application. Although code-coverage based regression techniques can be applied effectively to regression testing at the unit level, it becomes very difficult to manage all the information obtained from code and to create corresponding management matrices when testing a larger or more complex component, for example, a utility or a subsystem. Therefore, code-coverage based algorithms are not suitable for testing larger components at more abstract levels; secondly code-coverage based algorithms require testers to access and understand the code, which is very time-consuming; finally, code-coverage based algorithms are language-dependent, which will make the situation more complex. Considering the weakness of code-coverage based algorithms above, a code-coverage based algorithm is not a good choice to our thesis work. We don't propose to select a code-coverage based algorithm to be used in our implementation.

We don't propose to select a risk based algorithm to be used in our implementation. Since severity probability (P) and cost (C) are two important parameters in risk based algorithms, we can not estimate the cost for each test case in our test environment. Therefore, risk based algorithms are not suitable to use in our application.

We propose to select the fix cache algorithm for our implementation. The fix cache algorithm has the highest accuracy for predicting future faults in regression testing, which uses dynamic updating cache and relates with new fault distribution. Accuracy 73%-95% at file level is the most important factor as we interested in. In our thesis work, we will select the most fault-prone source code files in order to find relative most fault-prone test cases. Therefore, highest accuracy at file level is the most important advantage for our thesis development. Additionally, the fix cache algorithm doesn't require code instrument and test cases prioritization, which is the best method for our testing environment in practice.

3 Ericsson Tools Introduction

The test environment in Ericsson includes three main tools – CME, FIDO and CQTM.

CME is a tool to enforce the software configuration management (SCM) process and to help developers in their day-to-day work. CME is based on version control, which contains each version of changed source code files. A View in CME will correspond to one product and it is a filter that allows you to see all of source files which are related with this product. The advantage of CME is that it supports parallel development, enabling different development projects to work with the same set of source files concurrently, which allows several developers in the same project and distributes development teams geographically and more efficiently.

FIDO is a tool working as a bug tracking system for incident management in all phases of product development and maintenance. FIDO includes all of the information for each fault from the time of bug introduced until the fault has been fixed.

The workflow in FIDO is shown in Figure 2. After a bug is introduced, the fault is found during regression testing. In order to fix this fault, the tester needs to create an Error Report (ER) to inform the presence of this fault. An Error Report in FIDO is the document created for recording the presence of faults when testers found faults during regression testing for a product. An ER ID is a number which represents the document of an Error Report in FIDO. Each Error Report, thus, has its own unique ER ID number. We can find all information of a fault if we know the relative ER ID.

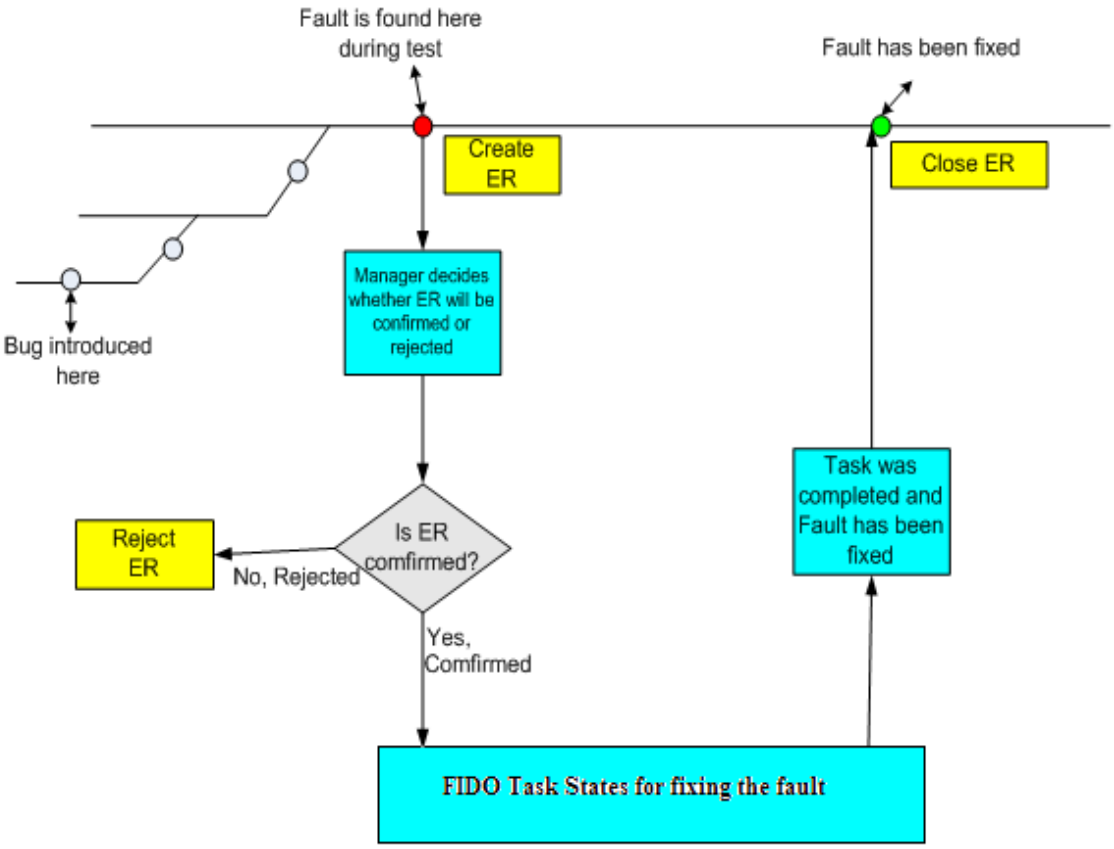

Figure 2. FIDO Workflow

After the Error Report has been created, the manager will decide whether this fault is severe to be fixed or it should be rejected. The Error Report either will be assigned to a developer for fixing the fault or it will be rejected directly by manager. There are four severity levels — Critical, High, Medium and Low, which is a judgment on how severe this problem is.

- Critical Severity: The fault will affect one or more development or test projects or even the whole project will be delayed, and must be fixed at first with the highest priority.

- High Severity: The fault has high impact on functionality in one project.

- Medium Severity: The fault is not easy reproducible during test, it is possible to be fixed if reproduce next time.

- Low Severity: The fault has no effects or low impact on functionality. It is not urgent to fix the faulty issue.

If the Error Report is confirmed by the manager, the ER is assigned to a developer for fixing the fault and the Task needs to be created in FIDO. The Task is a part of the Error Report, which describes an implementation task and requires executing some modifications by developers. Developers will execute activities in order to modify the source code and to fix the fault during the FIDO Task States (as shown in Figure 2). Each time when the source file is modified for fixing the fault, the modified date of this file will be recorded in Error Report. After developers finished code modification, the modified source code will be re-tested with the same test case as previously. If the result of re-test is passed, that means the fault has been fixed, thus Task is completed. Relative Error Report will be closed after the fault has been fixed.

Therefore, FIDO database contains all of the information of each modified source files (for instance, file name and file path, changed date etc) and the changed dates when fault has been found and fixed. For instance, latest changed date and ER created date and ER closed date for each modified source code, where latest changed date is the last time when developer modified the file and ER created date and ER closed date will be the same as the time when the fault has been found during test and the time when the fault was fixed.

Clear Quest Test Manager (CQTM) is a new developed test management tool from IBM/Rational. CQTM provides a database to store information and data about test cases and requirement mapping. Test case ID is a number which represent its relative test case information, such as test case name and what will be tested for. When a test case has been executed, the testing result will be reported automatically and stored in the CQTM database. If a fault has been found during regression testing, an Error Report will be created in FIDO (as described above). The belonging ER ID for this Error Report will be recorded by CQTM. Therefore, the test cases in the CQTM database will be related with the Error Report IDs corresponding to one product. That means you can find test case information from the CQTM database based on relative ER IDs from the FIDO database. Therefore, an ER ID is a common number which will be related with both the fault information in FIDO and the test case information in CQTM. It is easy to find all test cases and the testing result in the CQTM database corresponding to an Error Report ID in the FIDO database.

4 Develop the Fix Cache Algorithm in Ericsson

Considering the available tools and test environment in Ericsson as described in Chapter 3, we will plan how to design the fix cache algorithm with Ericsson tools in the Section 4.1 and describe how to implement the fix cache algorithm with Ericsson test environment in the Section 4.2.

4.1 Algorithm Design

Each CME View contains all available source code files which are related with one Ericsson special product. Since we are focused on .c and .h source files, we only fetch all available .c and .h files from the CME View corresponding to the special product as we are interested in. After we get all file names and file paths from the CME View, we will check both file name and file path for each file in the FIDO database in order to get the latest changed date, and the dates when fault was found and fixed for each relative file. And then we will pre-fetch the cache corresponding to our assumptions.

The cache size can be adjusted based on the number of resources available for testing or verification. A typical cache size is 10% of the total number of source code files, which provides a reasonable tradeoff between cache size and the accuracy of the cache algorithm. Larger cache size results in higher hit rate which means better prediction for future faults, but the faults will spread out over a greater number of source code files and will be difficult to select the real most fault-prone source code out of a large mount of source files. The ideal is to minimize the size of cache while at the same time maximizing the accuracy of the cache algorithm.

The reason why we decide the cache size to be 10% is based on Kim et al [17], where 10% is a typically empirical value and the result of hit rate at file level is 73%-95% accurate in Kim et al [17]. If we select the cache size to be 10%, it is easy to compare the hit rate of our developed algorithm with the hit rate mentioned in Kim et al report, and then we can know if our development is good enough to use for predicting future errors on Ericsson products.

After pre-fetching the cache, the fix cache will contain file name, file path, latest changed date, and ER ID for each source file. Then the algorithm will check each newly modified source code file every x hours and update dynamically and continuously based on fix cache algorithm.

Since the cache memory size is limited, our developed algorithm needs to unload one file from the cache, based on the cache replacement policies in order to keep the required cache size, before loading a new modified file to the cache.

The CQTM database contains all test case information and also relative ER ID numbers, where ER ID is a common number which is related with both the fault information in FIDO and the test case information in CQTM. The most fault-prone regression test cases will be finally selected from CQTM database corresponding to a list of ER ID which is related with the most fault-prone files in the continuously updated cache.

Figure 3 shows the processes about how to develop the fix cache algorithm with Ericsson Tools. In the following sub-chapters, the details about how to pre-fetch cache, update cache and finally find the most fault-prone test cases will be described.

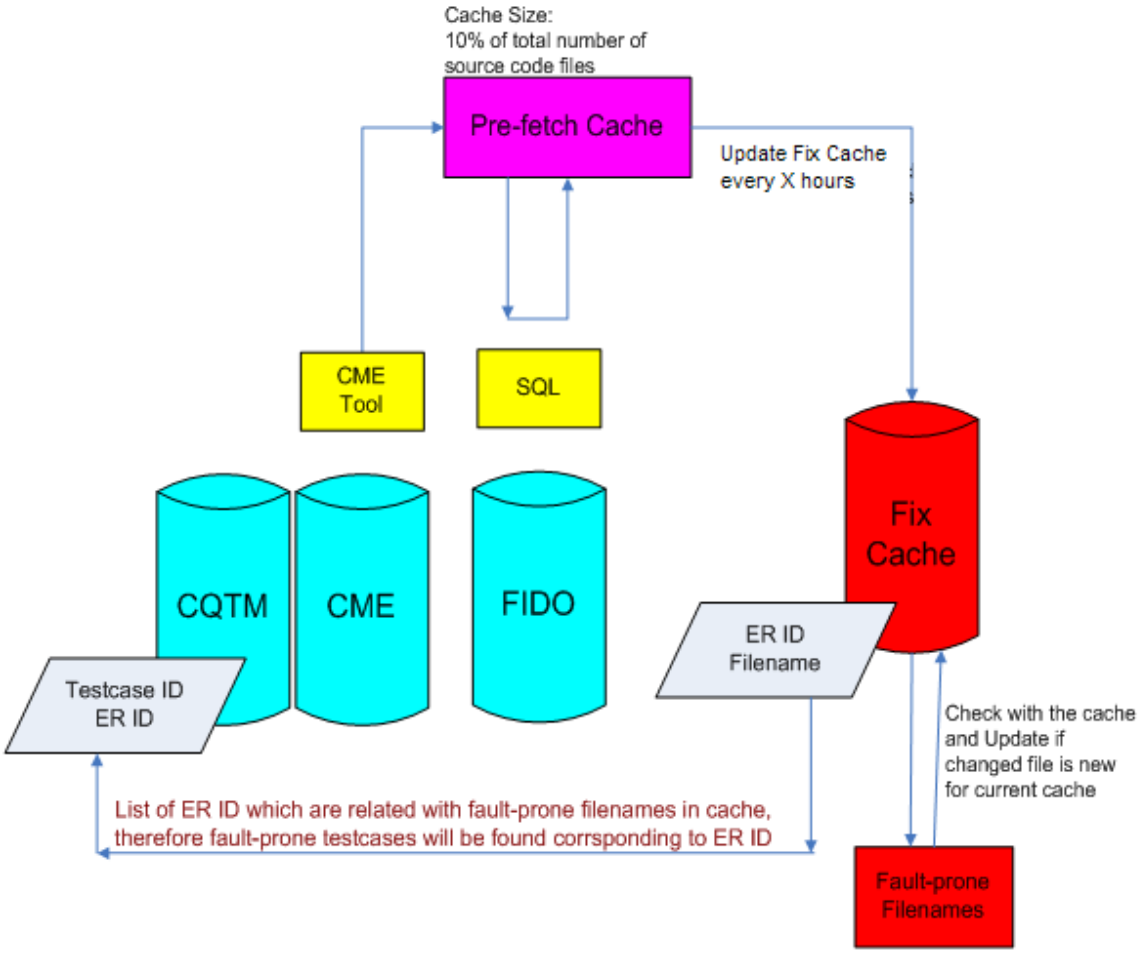

Figure 3. The processes of fix cache algorithm

4.1.1 Pre-fetch Cache

In order to improve the hit rate of the fix cache algorithm, we use pre-fetching techniques for pre-populating the fix cache before update the cache. Pre-fetching means that we load files for which we have not yet encountered a fixed fault. Our motivation is as follows: assume we would load files only when we encounter a fixed fault. As a consequence, we would have inevitable misses since we start with an empty cache. Additionally, it would be impossible to predict faults for files that have exactly one fixed fault in their lifetime. In order to reduce the miss count, we pre-fetch potential fault-prone files in advance by using the algorithm described below.

Initial pre-fetch, which describes, initially the fix cache is empty and in the absence of pre-fetching, this would lead to many misses. We avoid this and initialize the fix cache with the files likely to have faults as assumed by the greatest values of linear combining three parameters for each file, which are the number of changes, the number of relative critical fault reports and the number of lines of code.

Therefore, we will pre-fetch 10% of all available .c and .h source code files in our application, where we assume these files to be the most fault-prone ones.

4.1.2 Update Cache

After pre-fetching the cache, the current cache is the initial revision of the cache, which contains 10% of all available .c and .h source files as assumed as fault-prone ones, together with their relative file path, the latest changed date and the relative fault report.

According to the fix cache algorithm, we plan the way about how to update the fix cache and how to keep the required cache memory size at mean time. After the fault has been fixed, the algorithm will check the current cache with each of the newly modified files. If the file name of a newly modified source code is already contained in the current cache, count as a hit and the algorithm doesn't need to load this file name into the cache. The algorithm only needs to update the latest changed date for this file in the current cache. If the newly modified file is not contained in the current cache, which means it is a new one for the current cache, count as a miss and the algorithm must unload one file based on the cache replacement policies before it can load this new file. After checking each of the newly modified files, the algorithm summarizes the number of hits and the number of misses and then calculates the hit rate of the updating cache.

The algorithm will continuously update the fix cache every x hours during execution cycle, where we choose x=24 in our application. The reason why we update the cache every 24 hours is that the hit rate is easy to track daily.

Below we show one example about how to update the cache once. The current cache is shown in Table 3. Assume that a fault has been fixed in the file named “y.c”. Then the algorithm only needs to update latest changed date for the file because “y.c” is already contained in the current cache. Assume next that a fault has been fixed in the file named “mm.c”. Since the file is not contained in the current cache, the algorithm must unload one file before it can load new file “mm.c” into the cache

File name	File path	Latest changed date	ER ID in FIDO
x.c	C:\ww	2009-07-10	753849
y.c	C:\ww	2008-01-11	253648
z.h	C:\ww	2007-05-09	543879

Table 3. Example of current fix cache

If we use the “Least recently modified (LRU)” cache replacement policy for unloading files, this means that the file in the cache with least recently changed date will be unloaded. In the example in Table 3, the LRU policy will unload file “z.h”. After the cache has been updated, the files contained in the cache will be as shown in Table 4.

File name	File path	Latest changed date	ER ID in FIDO
x.c	C:\ww	2009-07-10	753849
y.c	C:\ww	2008-01-11	253648
mm.c	C:\ww	2009-08-20	864390

Table 4. Example of updated fix cache due to new file “mm.c”

4.1.2.1 Cache Replacement Policies

As the obligation is to maintain the cache size to be 10% of the total number of source code files, if some new files need to be loaded into the cache during the cache update, the same number of files has to be obliterated from the cache before load new file names. The cache replacement policy describes which files should be unloaded first.

We have investigated three different cache replacement policies as described in [17].

- Least recently modified (LRU):

  This algorithm unloads the file that has been least recently modified due to a fault. The LRU cache replacement policy is possible to implement based on the latest changed date of each source file.

- Number of changes (CHANGE):

  When a file has changed many times in the past, it is more likely to have faults in the future. We want to keep such files in the cache as long as possible. Consequently, we unload the file with the least number of changes.

- Number of previous faults (BUG):

  This policy is similar to the change-weighted LRU. It removes the file with the least number of observed faults. The intuition here is that when a file has had many faults, it will likely continue to have faults.

In operating systems, a frequently used policy is the least recently modified (LRU), which will replace and unload the elements used the longest time ago. We decide to use LRU policy for fix cache algorithm in the thesis work and unload the least recently modified files from the cache based on the latest changed date of each file.

4.1.3 Find Fault-prone Test Cases

The duration of algorithm execution for continuously updating cache and selecting the most fault-prone test cases should be the same as the duration of one cycle of regression testing. For instance, one cycle of regression testing for delivery test on Ericsson special product R13/2 is three months, so we should continuously update the cache every 24 hours in three months. The most fault-prone files will be contained in the final revision of the updated cache. Based on these most fault-prone files, the most fault-prone test cases can be selected from CQTM database. And these selected test cases will be executed in the next cycle of regression testing on product R13/2 delivery test instead of running all available test cases.

The most fault-prone test cases are a small subset of the total number of available regression test cases. In our application, we use 10% to be the cache size and the most fault-prone files are selected based on these 10% source files in final updated cache. Therefore, we will select around 10% of the total number of test cases to be executed in next regression testing cycle. The efficiency of regression testing will be improved and next testing cycle will be less cost-consuming due to test case selection.

4.2 Algorithm Implementation

Based on the algorithm design which is described in Section 4.1, the processes about how to implement the fix cache based regression test selection in Ericsson test environment are shown in this Section.

Step 1:Pre-populate the cache with 10% of the total number of available .c and .h source files.

Step 2: Continuously update the cache every 24 hours during the execution cycle.

Step 3: After the execution cycle, the most fault-prone test cases can be found from CQTM database based on the final revision of updated cache.

4.2.1 Pre-fetch Cache

Initially the cache is empty, we will pre-fetch the cache with 10% of all available .c and .h files, where we assume these files to be the most fault-prone files based on our three assumptions as mentioned below.

In our thesis work, we assume that:

- If the source code file has the largest number of code lines, it is most possible to have bugs inside the source code and it is most possible to trigger faulty issues during test.

- If the file has been modified most frequently, it is most possible to be fault-prone.

- If the source code file has been related with a large number of critical ER in FIDO database, where critical ER is the most severity fault that needs to be fixed with the highest priority, it is most possible to be a fault-prone file.

In order to pre-fetch the cache, we should consider three different parameters for each source code file as assumed above -- the number of changes, the number of lines of code and the number of critical ER. We purpose to select the top ten percent of total number of source code files regarding to the linear combination of these three parameters for each source code file.

The linear combination is computed by the following formula:

$$linear = \frac{\#changes}{\#sum\_changes} \cdot a_1 + \frac{\#ER\_criticals}{\#sum\_ER\_criticals} \cdot a_2 + \frac{\#LOC}{\#sum\_LOC} \cdot a_3$$

Where,

- '#changes' means the number of changes for each source file name which can be found in FIDO database; '#sum_changes' means the total number of changes for all source code files.

- '#ER_criticals' means the number of Error Reports for each source file where ER have critical severity; '#sum_ER_criticals' means the total number of critical Error Reports for all source code files.

- '#LOC' means the number of lines of code of each file; '#sum_LOC' means the total number of lines of code for all source files.

- a_1, a_2, a_3 are three parameters for linear combination which are the percentage weight of each assumption, where $a_1 + a_2 + a_3$ =100%. For instance, if we assume the number of lines of code is the most important factor which will affect the selection of the fault-prone source code, we can set the value of a_3 to be high percentage, i.e. 80% and $a_1 = a_2$ =10%; if we assume the number of lines of code is the only factor which will affect the selection of fault-prone files, that means the source code has the largest number of lines which is the most possible to be the fault-prone file, therefore we can set a_3 =100% and a_1, a_2 are 0% instead. It is possible to setup any percentage value to a_1, a_2, a_3 as our assumptions for pre-fetching the most possibly fault-prone files.

According to the idea of linear combination of three different factors which are possibly related with the selection and pre-fetch of the most possible fault-prone source codes, we can assume which factor has highest weight for pre-fetching most fault-prone files. In our thesis, we will try several different linear combinations and then decide which parameter of these three is the most important one and which group of a_1, a_2, a_3 linear combination is the best one for pre-fetching fault-prone source code files based on the hit rate of the fix cache with each linear combination.

4.2.2 Update Cache

After pre-fetching the cache, the current cache will include 10% of all available .c and .h source file names, together with their relative file path, ER ID and the latest changed date. And then we will start to update the fix cache every 24 hours continuously during the execution cycle. The processes of updating fix cache are shown in Figure 5.

Step 1: Find all of newly closed Error Report IDs during the latest 24 hours in FIDO.

Step 2: Check all these closed ER IDs (got from Step 1) and select ER IDs if it belongs to the special product R13/2. R13/2 as shown in Figure 5 is one of product name in Ericsson which we are interested in our thesis.

Step 3: Fetch all newly modified file names from the FIDO database since a fault has been fixed in each of these files. Newly modified source code files are related with the closed ER IDs in FIDO during previous 24 hours.

Step 4: Check each of these newly modified files with the current cache if the file is already in the current cache, count as a hit or a miss. And update the file in the cache.

- Step 4.1: If the file is already in the current cache, update the file with the latest changed date and count as a hit.

- Step 4.2: If the file is not in the current cache, count as a miss. Unload the least recently modified file based on cache replacement policy LRU before loading this newly modified source code file into the current cache.

Step 5: Summarize the number of hits and the number of misses got from Step 4 and then calculate the hit rate.

Step6: Continuously updating cache every 24 hours during the execution cycle following Step1-5.

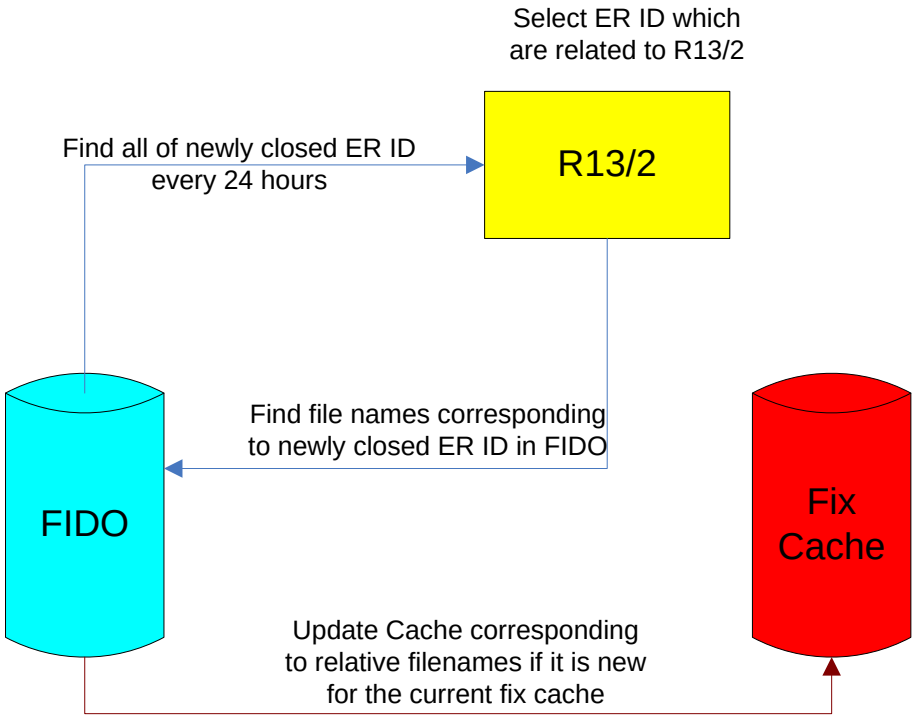

Figure 5. The processes of Cache Update

4.2.3 Find Fault-prone Test Cases from CQTM

After continuously updating cache during the execution cycle, the most fault-prone source code file names will be finally contained in the updated cache. We can find the most fault-prone test cases from CQTM database based on these updated fault-prone file names in the cache.

Since CQTM database contains test case information and also relative ER IDs, where ER IDs are common numbers which are related with both the fault information in FIDO and the test case information in CQTM, it is easy to find all relative test cases based on ER IDs. Since all relative ER IDs can be found in the FIDO database based on the most fault-prone files contained in the final revision of the cache, the most fault-prone regression test cases will be found finally from the CQTM database corresponding to a list of ER IDs which are related with the most fault-prone source code file names in the fix cache.

One example of finding the most fault-prone test cases based on the continuously updated cache is shown in Figure 6. The first table in Figure 6 is an example of the final revision of updated cache evaluation. The second table shows the most fault-prone test cases IDs to be found from the CQTM database corresponding to the ER IDs in the first table.

File name	File path	Latest changed date	ER ID in FIDO
x.c	C:\ww	2009-07-10	753849
y.c	C:\ww	2008-01-11	253648
…	…	…	…
…	…	…	…
m.c	C:\ww	2008-08-20	864390

Test Case ID	ER ID which is related with FIDO
568293564	253648
463286474	864390
…	…
758247432	753849

Figure 6. The example of finding fault-prone test cases

5 Result and Conclusion

5.1 Result Analysis

Our developed regression test selection algorithm based on the fix cache algorithm has been evaluated in two months on one special Ericsson product. We pre-fetched the cache based on 10% of all available .c and .h source code files and we assume these files to be most fault-prone files. We continuously updated the cache every 24 hours during the execution cycle.

The daily hit rate in the whole execution period is shown in Figure 7. The weekly hit rate, which is the average value of seven daily hit rates during one week, is shown in Figure 8. The average hit rate varies between 0.50 (50%) and 0.80 (80%). If comparing Figure 7 with Figure 8, it is easy to see that the hit rate is significantly lower than these averages for some days and the hit rate is significantly higher than the averages on several days. In Figure 7, there are four days during these two months which are even 1.0 (100%) hit rate. However, there is quite a lot of variation even within weeks and also a drop in week 5 since the processing of Ericsson product R13/2 is not stable.

As the results shown in Figure 7 and Figure 8, different groups of linear combination which have different percentage weights to setup a_1, a_2, a_3 values are tested in our application. However, the difference between the best hit rate trend and the worst one which correspond to two different linear combinations is within 10% of the maximum. That means the actual linear combination used to pre-fetch the cache has no significant impact on the hit rate, the simplest linear combination should be selected and implemented when pre-populating cache, for instance only consider two parameters such as the number of changes and the number of lines of code (LOC) to be severity parameters instead of fetching three parameters. Therefore, it will be much easier when pre-fetching the cache due to fewer factors to be considered in the linear combination.

The linear combination used to pre-populate cache does not seem to affect the results of hit rate so much. It could be that this is to be expected since the daily updating will soon overtake any effects of the pre-population cache. It could also be that even with normalized ranges one variable has a more extreme distribution and thus comes to dominate the results.

Since our thesis work has limited duration for evaluating our algorithm, the result is only the hit rates of updating cache in two months' algorithm execution, which is shorter than one cycle of regression testing of Ericsson special product. However, based on our result of hit rate, more evaluation of our developed algorithm will be continued by Ericsson testers after our thesis finished.

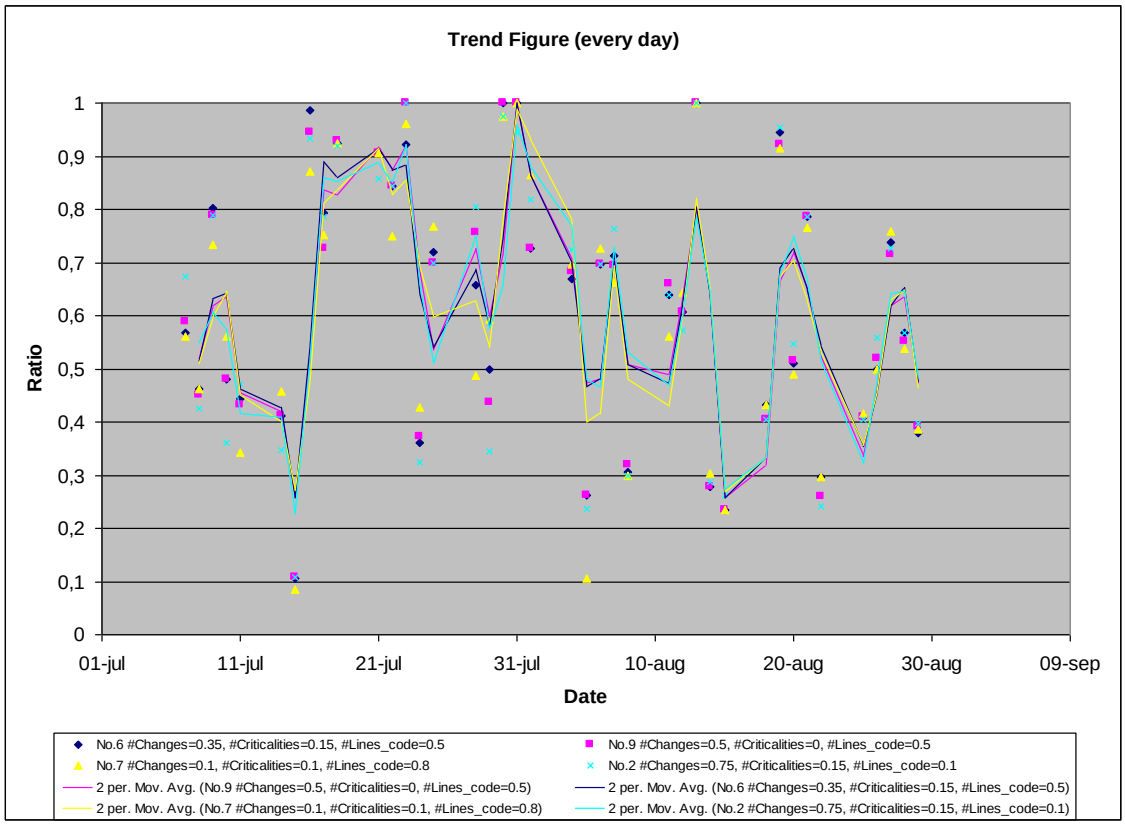

Figure 7. Hit rate in days at file level for updating fix cache

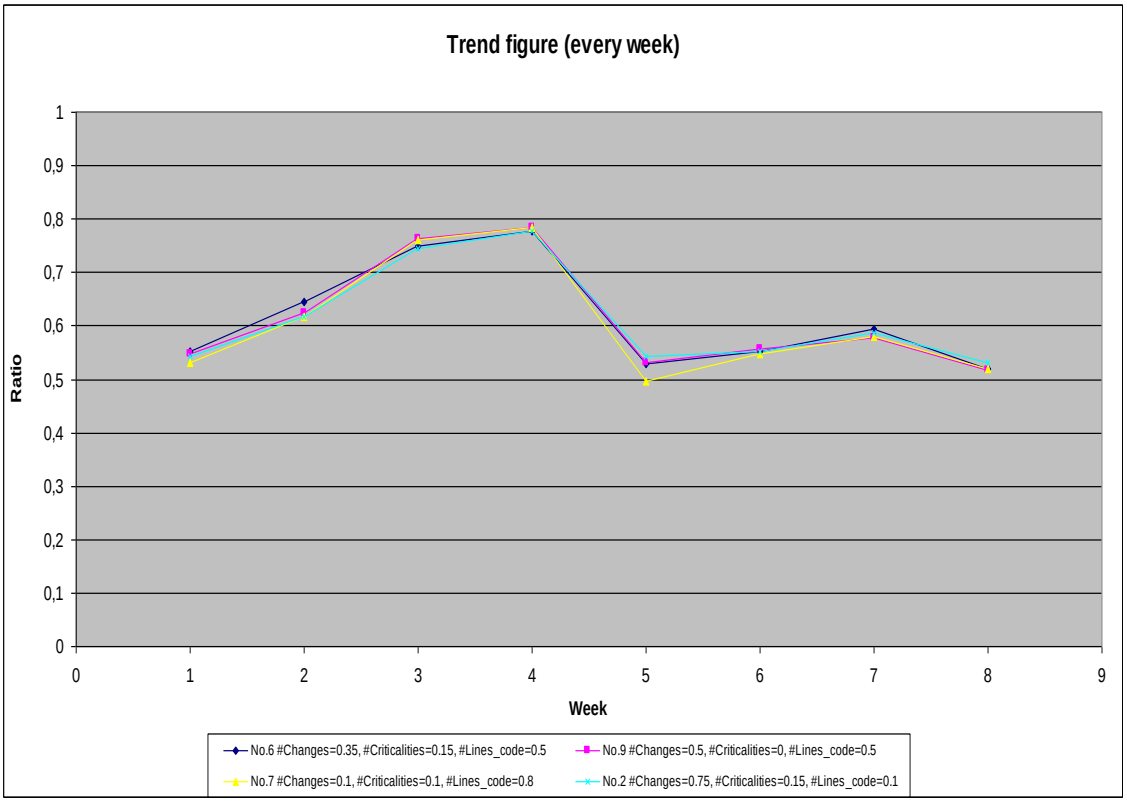

Figure8. Hit rate in weeks at file level for updating fix cache

5.2 Discussion

In this paper we present a new method for regression test selection, named fix cache based regression test selection, based on fixed error reports and modified source code files. Fix cache based regression test selection is a method where test cases are selected based on their ability to detect as many future faults in regression testing as possible. The cache is a convenient mechanism for holding fault history as a dynamically updated list of the most fault-prone files, and for aggregating multiple heuristics for maintaining the cache. In our thesis, we develop a method for automatic selecting the most fault-prone files and test cases based on the fix cache algorithm [17]. The most fault-prone test cases are selected based on the most fault-prone files contained in the continuously updated cache. The hit rate of the fix cache algorithm, which is the accuracy of predicting future faults based on the updated cache, was investigated during two months' execution cycle and it is shown in the final result.

There are two important advantages of fix cache algorithm based regression test selection.

An important advantage of this algorithm, compared with other techniques, is that it can more easily and dynamically adapt to changes in the fault history, which is quite helpful for automatic selecting fault-prone regression test cases. For code-coverage based and risk based regression testing techniques the fault prediction models they develop are more static and rely on factors that have historically been correlated with faults. This can imply that they rely on assumptions that may not hold in practice. We thus think that dynamic and adaptive methods like the one we have investigated here show more potential for real-world and large-scale use. We encourage the research community to investigate and develop such methods to a larger extent than today. However, it is important that evaluation of testing techniques and methods are done on large software systems and not only open-source ones. We would encourage the academic and industrial domains to cooperate more around such studies in the future.

Another important advantage of the fix cache algorithm over other regression test selection models, for instance the coverage based regression test selection and the risk based regression, is that the fix cache algorithm has the highest accuracy for predicting future faults and it can adapt more quickly to new fault distributions. This is an important reason why we decide to use fix cache algorithm in our thesis. In practice, we can see from the final result of hit rate in Figure 7 and Figure 8, the accuracy of the fix cache algorithm is quite high at file level.

The fix cache algorithm doesn't require code instrument and it doesn't require testers to access and understand the source code files, therefore, it is convenient to implement for selecting a subset of test cases in even large software system.

Most previous prediction models found in the literature use fault correlated factors and develop a model to predict future faults. The model is static and it incorporates all precious history and factors after it is developed. In contrast, the fix cache algorithm can cooperate with the fault distributions of each product. Even though products have different fault distributions, the fix cache algorithm adaptively updates and calculates the hit rate based on the number of hit and the number of miss. The cache size will affect the hit rate of updating cache. Larger cache size results in higher hit rate which means better prediction for future faults, but with the faults spread out over a greater number of source code files. The ideal is to minimize the size of cache while at the same time maximizing the accuracy of cache algorithm. Our extension is a way to link source code files in CME View with regression test cases in CQTM database based on Error Reports in FIDO database, where CME, CQTM and FIDO are three different tools in Ericsson test environment.

The fix cache algorithm has been implemented and evaluated in a large, industrial, embedded, real-time software system at Ericsson in Sweden. The software system is consisted by several million lines of code and a five digit number of source code files, which is a software system used by mobile phone manufacturers.

In addition to being an industrial evaluation of the previously proposed fix cache algorithm, a contribution of our thesis work is the way we link test cases and source code files without requiring explicit code coverage measurements and code instrument. This makes the approach useable for systems where coverage information is not readily available, such as in real-time and embedded systems. A further contribution is that our system has also been evaluated in an actual, industrial production environment.

The result of the fix cache algorithm implementation in practice shows that the hit rate of fix cache algorithm which we got in our thesis work are lower (typically in the range 50-80%) than the previously presented ones (73-95%) in the literature [17]. However, there is a very large reduction in the number of test cases comparing with the total number of test cases. We have a list of the most fault-prone test cases which will be recommended in future regression testing. The potential efficiency gains are very large.

In future work we will examine the quality and effectiveness of this list of the most fault-prone test cases, as well as investigate extensions to the fix cache algorithm for a large, industrial, embedded, real-time software system in practice in order to increase the hit rate of updating cache.

6 Future Work

We currently update the cache daily even if regression testing is not done daily. In future work we should investigate if cache updating based on the frequency of regression testing offers better opportunities for optimizing the cache updating scheme. By including more information, i.e. over several days, in a single cache update decision, that decision could potentially be more accurate.

In our thesis work, only test cases which have failed and where the problem has been fixed are included by the algorithm. We are looking into performing low intrusive system level code coverage measurements to be able to associate more test cases with source files. Another way to create better linking information would be to monitor changes to source or test files and correlate them. Thus we could update the links between test cases and source files as changes are made rather than only when faults are fixed. The cache size can also be evaluated and changed corresponding to deleted files during cache update. In our thesis, we use 10% to be cache size and make sure the size to be 10% with cache replacement policy during cache update. In the future work, we can create another database to contain all deleted files due to cache replacement. In this database, we can count how many times each file is deleted during cache update. If there are some files which have been frequently deleted, that means these files are also frequently modified. Therefore, we should enlarge the size of cache and add these files into updated cache.

Another future work is to implement the bug cache algorithm to evaluate module with finding bug introducing changes. By comparing the current software version with the latest tested version, it would be more accurate to find the difference between two versions and find the place of bug introducing changes. Thus it is more effective for predicting future faults.

In this study we have seen considerably lower hit rates on a few days than was reported in [17]. It is not clear what causes this difference. We speculate that differences in development of industrial and open-source software might be the cause. However, for all days in our study where the hit rate is very low it quickly recovered in the following days. Thus, the overall efficiency of cache-based regression test selection schemes should not be affected much, even if the hit rates are temporarily lower. Potentially our method could be enhanced by resorting to other cache update strategies on days when the hit rates are lower than normal. This is a potential area for future research.

6.1 Tool Analysis about eROSE Plug-in

eROSE plug-in [27] uses version archives to make recommendations for guiding programmers, which is effective and unobtrusive for predicting faults. If we can integrate the cache algorithm with eROSE, this tool can automatically suggest further locations to be examined for related faults whenever a fault is fixed [9]. All eROSE needs is a CVS repository [14], which is easy to obtain with CME View version control.

Since there is a feature for software that is 'programmers who changed function X also changed function Y', we could analyze version histories of large software systems and try to identify commonalities and anomalies for this purpose, and also guide the programmer to understand and maintain. The major application for eROSE is to guide users through source code, i.e. the user changes some entities and eROSE automatically recommends related changes in a view. Since this dependency is undetectable by program analysis, eROSE is more useful for improving navigation and preventing errors.

eROSE consists of two parts: Preprocessing and Mining.

Since CVS have some weaknesses, i.e. it is slow and loses information on transactions, fine-grained changes and merges, the preprocessing of data is a fast access to CVS data and necessary in eROSE, which includes four essential preprocessing tasks (data extraction, transaction recovery, mapping changes to entities and data cleaning).

For data mining techniques in eROSE, there are three main advantages comparing with other tools:

- Use full-fledged data mining techniques to obtain association rules from version histories.

- Detect coupling between fine-grained program entities such as functions or variables, thus increasing precision and integrating with program analysis.

- Thoroughly evaluates the ability to predict future or missing changes, thus evaluating the actual usefulness of our techniques.

However, there are three different evidences of data mining in eROSE:

- For stable systems, eROSE gives many and precise suggestions: 44% of related files and 28% of related entities can be predicted, with a precision of about 40% for each single suggestion, and a likelihood of over 90% for the three topmost suggestions.[27]

- For rapidly evolving systems, eROSE gives most useful suggestions at the file level. Overall, this is not surprising, as eROSE would have to predict new functions, which is probably out of reach for any approach.

- In about 4%–7% of all erroneous transactions, eROSE correctly detects the missing change. If such a warning occurs, it should be taken seriously, as only 2% of all transactions cause false alarms.[27]

However, we should take these evidence into account if we implement eROSE plug-in for our cache algorithm in the future work.

7 Reference

[1] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," *Proc. of International Conference on Software Maintenance (ICSM 2005)*, Budapest, Hungary, 2005, pp. 263-272.

[2] A. K. Onoma,W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. Commun. ACM, 41(5):81–86, 1998.

[3] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," *Proc. of International Conference on Software Maintenance (ICSM 2000)*, San Jose, California, USA, 2000, pp. 120-130.

[4] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, vol. 5, pp. 169-180, 2002.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10):929–948, 2001.

[6] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental Regression Testing. In Proceedings of the IEEE Software Maintenance Conference, pages 348–357, 1993.

[7] H. Leung and L. White. Insights into regression testing. In Proceedings of the IEEE Software Maintenance Conference, pages 60–69, 1989.

[8] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In 2005 International Symposium on Empirical Software Engineering, pages 64–73. IEEE, 2005.

[9] J. liwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," *Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, 2005.

[10] L. White and B. Robinson. Industrial real-time regression testing and analysis using firewalls. In Proceedings of the IEEE Software Maintenance Conference, pages 18–27, 2004.

[11] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 19th International Conference on Software Maintenance (ICSM 2003), Amsterdam, The Netherlands, pp. 23-32, 2003.

[12] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE.

[13] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. of 2005 Int'l Conference on Software Engineering (ICSE 2005)*, Saint Louis, Missouri, USA, 2005, pp. 284-292.

[14] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. of 2006 Int'l Conference on Software Engineering (ICSE 2006)*, Shanghai, China, 2006, pp. 452-461.

[15] S. Elbaum, G.Rothermel, S.Kanduri, and Alexey G. Malishevsky, " Selecting a Cost-effective Test Case Prioritization Technique", IEEE, 2004

[16] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, Jr., "Automatic Identification of Bug Introducing Changes," *Proc. of International Conference on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.

[17] S. Kim, T. Zimmermann, E. J.W. Jr., and A. Zeller. Predicting faults from cached history. In ICSE'07, pages 489–498, Washington DC, USA, 2007. IEEE Computer Society.

[18] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, pp. 653-661, 2000.

[19] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Journal*, vol. 11, pp. 19- 37, 2003.

[20] T. M. Khoshgoftaar and E. B. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software," *Proc. of The Ninth International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998, pp. 344-353.

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, vol. 31, pp. 340- 355, 2005.

[22] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.*, vol. 31, pp. 429-445, 2005.

[23] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based regression test selection with risk analysis. In CASCON '02, pages 1–14. IBM Press, 2002.

[24] Y.-F. Chen, D. Rosenblum, and K.-P. Vo. TESTTUBE: a system for selective regression testing. In ICSE'94, pages 211–220. IEEE Computer Society, 1994.

[25] Y. Li and N. J. Wahl. An overview of regression testing. SIGSOFT Softw. Eng. Notes, 24(1):69–73, 1999.

[26] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering, 33(4):225–237, 2007.

[27] Zimmermann.T , Dallmeier.V, Halachev.K and Zeller.A. eROSE: Guide Programmers in Eclipse