Regression Testing of Transcompiled Cross-Platform Applications

Dissertation

der Mathematisch- und Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Matthias Hirzel
aus Münsingen

Tübingen
2018

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	20.09.2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Herbert Klaeren
2. Berichterstatter:	Prof. Dr. Torsten Grust

Abstract

Regression testing is a well-accepted and widely used way to ensure that already existing functionality still works as expected when code has been modified in order to realize new features or to resolve known bugs. To this end, developers re-execute previously created tests and check whether these tests still pass. However, depending on the application, regression testing might be very time consuming. This already applies when testing small units, but plays a more and more decisive role when integrated functions and especially end-to-end tests (such as UI or web tests) have to be re-executed. In the worst case, executing all the tests takes several days or even weeks. Consequently, it takes far too much time to get feedback about test failures. In order to solve this problem, researches have presented many different techniques. Originally, these techniques have been tailor-made for standard desktop applications. With the dawn of web applications, it has turned out that the approaches available so far are not directly applicable to web applications. So more and more researches have attended to the special challenges to realize regression testing techniques for this new kind of application. In both standard desktop and web applications, especially regression test selection and test prioritization have shown to be a good strategy to reduce the testing effort.

Today, another kind of applications becomes more and more important: Transcompiled cross-platform applications. The basic idea is to create an application in a source programming language A and to compile it automatically with a special transcompiler into another target programming language B. This way of developing new software has several advantages in different areas: In web applications for example, it is easier to write the software in a specific source programming language that has sophisticated debug support. In mobile applications, it is a means of cost reduction. The developers write an application only once. By transcompiling the code, they can deliver many different platforms such as Android, iOS, and Windows Phone. Apart from that, with transcompilation, it is basically possible to create both a desktop application and a web application without writing the code twice. Nevertheless, this new way of developing code creates several challenges in regression testing. Once more, existing approaches for desktop or web applications are not directly applicable.

In this thesis, we investigate three main problems that affect regression testing in the special context of transcompiled cross-platform applications. First,

we focus on the test effort reduction problem that comprises three questions: (a) Which tests should run at all, (b) how to execute tests in an efficient, memory- and time-saving way, and (c) which execution order would be suitable if many tests should be re-executed. A main challenge is the nature of UI/web tests as they execute many different functions in different application states rather than isolated, small functions as unit tests do. Even small code changes might affect lots of tests that are all intended to check different functionality. In addition, the distinction between source and target programming language as well as the dependencies in between makes it difficult to determine a reduced set of UI/web test cases that should be re-executed due to code changes.

The second problem addresses fault localization in transcompiled cross-platform applications. UI/web tests execute code in the target programming language. If a test fails, the decisive question is which parts in the code of the source programming language are responsible for the failure. Due to the automatic transcompilation and additional code optimization and obfuscation, it is difficult to identify the reasons.

As a solution to these problems, we present an approach that improves and extends an already existing regression test selection technique. Our approach consists of several steps. First of all, it analyzes which code changes have been made in the source programming language. Based on this analysis, we select all the UI/web tests that run corresponding pieces of code in the target programming language. As a prerequisite, we have to close the gap between the source and the target programming language. To overcome this issue, we introduce and investigate two special code instrumentation techniques. They are universally applicable in different transcompilers and address the special challenges in transcompiled cross-platform applications. Both instrumentation techniques also take into account the problem to identify precisely which code change(s) in the source programming language might be responsible for a failure when running a test on the code of the target language.

In order to make our approach more effective, we explore different settings. Our aim is to reduce the overhead for the analysis and to keep the memory consumption as low as possible. To this end, we propose and evaluate several heuristics that decide individually how fine-grained the analysis of the source code should be. Besides, we apply lookaheads and two search algorithms to detect more possible bugs in a single analysis.

In order to decide which execution order is suitable if many tests have to be re-executed, we combine our regression test selection technique with one of six novel prioritization techniques. The overhead of the extra test case prioritization is very low. At the same time, the prioritization gives a clear test execution order. Thus, the most important tests that have the biggest chance to reveal faults run first. This even enables developers to reduce the set of tests on their own if the test selection chooses many tests for re-execution and if there are constraints (e.g. execution time) that prevent to re-execute all the tests. Moreover, the combination of test selection and prioritization can be used to realize continuous integration even for UI/web tests.

Finally, the third challenge arises from the coverage identification problem. In order to ensure that the software application works as expected, it is very important to have many tests that cover all possible use cases. Usually, this is checked by applying different code coverage metrics. However, existing code coverage tools cannot be applied in transcompiled cross-platform applications. They fail to determine which parts of the code in the source programming language are covered by UI/web tests. We introduce an extension of our instrumentation technique that is applicable for both unit tests, integration tests, and for UI/web tests in particular. It supports standard desktop applications, but more importantly, it calculates multiple coverage measures for transcompiled cross-platform applications.

Zusammenfassung

Regressionstests sind eine anerkannte und weit verbreitete Methode um die korrekte Funktionsweise bereits existierender Features in Software-Anwendungen sicherzustellen, nachdem neue Funktionalität hinzugefügt oder bekannte Fehler behoben wurden. Zu diesem Zweck führen Entwickler bereits existierende Tests erneut aus und prüfen, ob sie noch immer fehlerfrei durchlaufen. Je nach Art der Software-Anwendung können Regressionstests jedoch sehr zeitintensiv sein. Dies kann bereits auf den Test kleiner Komponenten (englisch: *Units*) zutreffen, spielt aber eine immer größere Rolle wenn zusammengesetzte Funktionen oder End-to-end Tests – wie (Benutzer-) Oberflächentests oder Webtests – erneut ausgeführt werden müssen. Im schlimmsten Fall dauert die Ausführung aller Tests mehrere Tage oder sogar Wochen. Folglich dauert es viel zu lange bis man Feedback zu fehlgeschlagen Tests erhält. Um dieses Problem zu lösen haben Forscher viele verschiedene Techniken vorgestellt. Ursprünglich waren diese auf gewöhnliche Desktop-Anwendungen zugeschnitten. Mit dem Aufkommen von Web-Anwendungen hat sich dann herausgestellt, dass sich die bislang verfügbaren Ansätze darauf nicht direkt übertragen lassen. Aus diesem Grund haben sich immer mehr Forscher den speziellen Herausforderungen gewidmet, die sich aus der Realisierung von Regressionstesttechniken für diese neue Art von Anwendung ergeben. Sowohl in Desktop- als auch in Web-Anwendungen haben sich speziell Regressionstestselektion und Testpriorisierung als eine gute Strategie erwiesen, um den Testaufwand zu reduzieren.

Heutzutage gewinnt eine andere Art von Anwendung immer größere Bedeutung: Transkompilierte plattformübergreifende Anwendungen. Der Grundgedanke dabei ist, eine Anwendung in einer Ausgangsprogrammiersprache A zu erstellen und mit Hilfe eines speziellen Transkompilers automatisch in eine andere Zielprogrammiersprache B zu kompilieren. Diese Art der Entwicklung neuer Software hat mehrere Vorteile in unterschiedlichen Bereichen: Beispielsweise ist es in Web-Anwendungen einfacher, die Software in einer speziellen Ausgangsprogrammiersprache zu schreiben die umfangreiche Unterstützung zum Debuggen bietet. In mobilen Anwendungen trägt diese Art zu entwickeln zur Kostenreduktion bei. Entwickler schreiben eine Anwendung nur ein Mal. Durch Transkompilieren des Codes können verschiedene Plattformen wie Android, iOS und Windows Phone bedient werden. Abgesehen davon ist es mit Transkompilation grundsätzlich möglich, sowohl eine Desktop-Anwendung als auch eine Web-Anwendung zu erstellen ohne den Code doppelt schreiben zu müssen. Nichtsdestotrotz bringt diese neue Art der Code-Entwicklung

mehrere Herausforderungen beim Regressionstesten mit sich. Einmal mehr lassen sich bestehende Ansätze für Desktop- oder Web-Anwendungen nicht direkt übertragen.

In dieser Dissertation untersuchen wir drei Hauptprobleme die Regressionstests im speziellen Kontext von transkompilierten plattformübergreifenden Anwendungen betreffen. Zunächst konzentrieren wir uns auf das Problem der Testaufwandsreduktion. Es umfasst drei Fragen: (a) Welche Tests sollen überhaupt ausgeführt werden, (b) wie können Tests effizient, speicherschonend und zeitsparend ausgeführt werden, (c) welche Ausführungsreihenfolge ist geeignet wenn mehrere Tests ausgeführt werden müssen. Eine wesentliche Herausforderung ist die Beschaffenheit von Oberflächen-/Webtests. Sie führen viele verschiedene Funktionen in unterschiedlichen Anwendungszuständen aus, anstatt sich auf kleine, isolierte Funktionen zu beschränken wie es Komponententests tun. Selbst kleine Code-Änderungen können mehrere Tests betreffen, die aber alle unterschiedliche Funktionalitäten prüfen. Des weiteren macht es die Unterscheidung zwischen Ausgangsprogrammiersprache und Zielprogrammiersprache und deren gegenseitige Abhängigkeiten schwer, eine kleine Menge an Oberflächen-/ Webtests zu bestimmen die aufgrund von Änderungen erneut ausgeführt werden sollten.

Das zweite Problem richtet sich an die Fehlerlokalisierung in transkompilierten plattformübergreifenden Anwendungen. Oberflächen-/Webtests führen Code in der Zielprogrammiersprache aus. Wenn ein Test fehlschlägt ist die entscheidende Frage, welche Teile im Code der Ausgangsprogrammiersprache für den Fehlschlag verantwortlich sind. Aufgrund der automatischen Transkompilierung und damit häufig einhergehenden zusätzlichen Code-Optimierungen und -Ver-schleierungen ist es schwierig, die Gründe zu identifizieren.

Als Lösung für diese Probleme stellen wir einen Ansatz vor, der eine bereits existierende Regressionstestselektionstechnik verbessert und erweitert. Unser Ansatz besteht aus mehreren Schritten. Zunächst analysiert er, welche Code-Änderungen in der Ausgangsprogrammiersprache gemacht wurden. Basierend auf dieser Analyse wählen wir alle Oberflächen-/Webtests aus, die entsprechende Code-Teile in der Zielprogrammiersprache ausführen. Voraussetzung hierfür ist, dass wir die Lücke zwischen der Ausgangs- und der Zielprogrammiersprache schließen können. Um diese Problematik zu lösen, führen wir zwei spezielle Code-Instrumentierungstechniken ein und untersuchen diese. Sie sind universell in verschiedenen Transkompilern einsetzbar und gehen die besonderen Herausforderungen in transkompilierten plattformübergreifenden Anwendungen an. Beide Instrumentierungstechniken berücksichtigen auch das Problem, in der Ausgangsprogrammiersprache exakt diejenigen Code-Änderungen(en) zu identifizieren, welche für den Fehlschlag eines Tests bei der Ausführung des Codes der Zielprogrammiersprache verantwortlich sein könnten.

Um unserer Ansatz effizienter zu machen, untersuchen wir unterschiedliche Einstellungen. Unser Ziel ist es, den Aufwand für die Analyse zu reduzieren und den Speicherverbrauch so gering wie möglich zu halten. Zu diesem Zweck schlagen wir mehrere Heuristiken vor und untersuchen diese. Jede Heuristik

entscheidet individuell, wie feingranular die Analyse des Quellcodes sein soll. Außerdem wenden wir Lookaheads und zwei Suchalgorithmen an, um mehr mögliche Bugs in einer einzigen Analyse aufzudecken.

Um entscheiden zu können, welche Ausführungsreihenfolge geeignet ist wenn viele Tests erneut ausgeführt werden müssen, kombinieren wir unsere Regressionstestselektionstechnik mit einer von sechs neuen Priorisierungstechniken. Der Mehraufwand der zusätzlichen Testfallpriorisierung ist sehr gering. Gleichzeitig gibt die Priorisierung eine klare Testausführungsreihenfolge vor. Dementsprechend werden die wichtigsten Tests mit der größten Chance zur Fehleraufdeckung als erstes ausgeführt. Dies ermöglicht es Entwicklern sogar, die Menge der Tests selbst zu reduzieren falls die Testselektion viele Tests hat und es Einschränkungen (z.B. Ausführungszeit) gibt, welche die erneute Ausführung aller Tests verhindert. Darüber hinaus kann die Kombination von Testselektion und Priorisierung verwendet werden, um kontinuierliche Integration selbst für Oberflächen-/Webtests umzusetzen.

Die dritte Herausforderung ergibt sich schließlich aus dem Code-Abdeckungsproblem. Um die erwartungsgemäße Funktionsweise der Software-Anwendung sicherzustellen ist es wichtig, viele Tests zu haben, die alle möglichen Use Cases abdecken. Üblicherweise prüft man die Testabdeckung mit Hilfe verschiedener Metriken. Bereits verfügbare Tools zur Berechnung dieser Metriken sind jedoch nicht in transkompilierten plattformübergreifenden Anwendungen einsetzbar. Sie können nicht ermitteln, welche Teile des Codes in der Ausgangsprogrammiersprache von Oberflächen-/Webtests überdeckt werden. Wir führen eine Erweiterung unserer Instrumentierungstechnik ein, die sowohl für Komponententests, Integrationstests und speziell für Oberflächen-/Webtests anwendbar ist. Sie unterstützt normale Desktopanwendungen, berechnet aber insbesondere für transkompilierte plattformübergreifende Anwendungen mehrere Testabdeckungsmaße.

Acknowledgements

I am indebted to Prof. Herbert Klaeren for the possibility to do my doctorate in his group and for his continuous support even when he had already retired. During my student days, he introduced me to software engineering and in particular to testing as an option to ensure software quality. He offered me the opportunity to do research on this topic and gave advice on papers and early drafts of this dissertation. I am also grateful to my co-supervisor Prof. Torsten Grust for his interest in my work.

A special thanks to Holger Gast for many helpful discussions about technical details of my research and further suggestions. Konstantin Grupp has implemented the very basic infrastructure and an initial version of the code comparison algorithm for our Eclipse plug-in in his bachelor thesis [122]. I give many thanks to itdesign GmbH for allowing me to evaluate my approach on their software Meisterplan. I would also like to express my gratitude to Jonathan Brachthäuser as well as Julia Trieflinger for creating web tests for the software Hupa and for seeding errors in this application. Besides, I want to thank Jonathan Brachthäuser, Dennis Butterstein, Yufei Cai, Paolo Giarrusso, Tillmann Rendel, and Julia Trieflinger for many pleasant lunch breaks and discussions.

Last but not least, I would like to thank my family for having supported me all the time.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability 108
APFD	Average of the Percentage of Faults Detected 16
AST	Abstract Syntax Tree 62
BD	Body Declaration 120
CDG	Control Dependence Graph 20, 39
CFG	Control Flow Graph 20
CFP	Change Frequency-based Prioritization Technique 162
CI	Continuous Integration 14, 116
CID	Code Identifier 81
DCFP	Dynamic Change Frequency-based Prioritization Technique 163
DGFP	Global Dynamic Frequency-based Prioritization Technique 163
DLFP	Local Dynamic Frequency-based Prioritization Technique 163
DOM	Document Object Model 43
DSL	Domain Specific Language 1
E*	Expression Star 120
ECN	External Code Node 55
EJIG	Extended Java Interclass Graph 62
GFP	Global Frequency-based Prioritization Technique 160
GWT	Google Web Toolkit 3
IDE	Integrated Development Environment 33
JABA	Java Architecture for Bytecode Analysis 57
JDT	Java Development Tools 22
JIG	Java Interclass Graph 55
LFP	Local Frequency-based Prioritization Technique 161
NLOC	Non-Empty Lines Of Code 3
NCSS	Non-Commented Source Statements 3
PDG	Program Dependence Graph 39, 207
PDT	PHP Development Tools 22
RAP	RemoteApplicationPlatform 3
RTS	Regression Test Selection 5
SDG	System Dependence Graph 39, 208
SRT	Selective Regression Testing 5
TCP	Test Case Prioritiztion 5
TSM	Test Suite Minimization 5

Chapter 1

Introduction

Today, there exist many special compilers that allow to write an application in a specific programming language and to translate the resulting code into another programming language automatically (e.g. [45, 70, 98, 120, 130, 151, 203, 207, 286]). These special compilers are usually referred to as *transcompilers* or *source-to-source compilers*. The motives for using a transcompiler are eclectic. This includes efforts to solve issues that arise during requirements analysis, software design, and software development. Moreover, it is advantageous for the cost planing. However, using a transcompiler impacts the test and bug fixing process. Dealing with emerging problems in software testing and maintenance is the main topic of the thesis. First, we want to consider the different aspects in more detail that reason the usefulness and the usage of transcompilers.

When creating a new software from scratch, developers are always faced with the question which programming language should be used. Everybody will agree that there is no best programming language that should always be used to realize software projects. Moreover, it is common in software projects that several languages interact in order to accomplish an overall task. Often, *domain specific languages* (*DSLs*) are consulted in order to solve a task. So it is impossible to find an always and universally valid answer to the question which programming languages should be used. Developers have to analyze which programming languages are the most suitable to implement the software. During the analysis, many aspects have to be considered [32, 222]: Among others, developers require information about possible future customers and the field of use of the software. This includes knowledge about future target platform(s) and whether the application should be portable. Beyond that, developers have to assess how easily the application can be adapted to new or changed needs when using a certain language. From a developer's point of view, the effort to solve a problem and the vitality of a language are important factors in order to be able to get the software ready for production as fast as possible. Finally, the application's performance is very important for users.

Sometimes, the analysis might come to a tradeoff between the different aspects when deciding which programming languages to use. This implies that the language of choice has several advantages but also involves some draw-

backs. For example, a language like C# offers platform specific libraries to create user interfaces with a native look and feel. However, when targeting at several platforms, this language cannot be used directly. The same applies to mobile applications, often referred to as apps. The most wide spread platforms Android, iOS, and Windows Phone differ extremely. None of them uses the same language for implementing apps. An app developed for a specific platform (e.g. Android) is naturally incompatible to other platforms (e.g. iOS, Windows Phone). Developing the same app for several operating systems is very time consuming and expensive.

When targeting multiple platforms, web applications could offer an alternative. They benefit from minimal system requirements. Given an up-to-date browser and an internet connection, they are available from everywhere. This is very convenient from both a customer's and a developer's point of view. Developers have a great benefit as it is very easy to offer new functionality or to fix bugs rapidly. Furthermore, today's web applications are not inferior to desktop applications with respect to functionality and performance. Especially the client side does no longer only display contents to the user but is able to perform sophisticated tasks. The power of these applications is often due to JavaScript and due to the advent of AJAX [90]. This technique offers varied possibilities to load additional data or to manipulate contents dynamically and asynchronously in order to display further information in the browser as needed without requiring a full reload of the web page [90]. So, the user experience is highly interactive and does not differ much from standard desktop applications.

Nevertheless, web applications are associated with disadvantages, too. In general, they are considered to be harder to test (e.g. [15, 76, 225]). And due to the dynamically and loosely typed semantics as well as because of the prototype-based inheritance in JavaScript, code is additionally considered to be more error-prone [15, 132]. Other problems arise from diverging browser behavior [132] and from AJAX. Being highly beneficial for the user experience, the asynchronous data transfer impedes testing particularly and therefore adds further challenges (e.g. [15, 86, 188, 226]). Apart from that, another drawback might be the weak support for debugging.

As a general solution to these problems, frameworks that allow focusing on easy development and testing become more and more popular when deciding on the (main) programming language that should be used to create an application. The platform on which the software is going to run plays a minor role. The resulting source code will be compiled into another programming language by a *transcompiler* that is integrated in the framework. This language fits the requirements of the future platform as well as possible. The process of compiling code from a specific source programming language in another target language is called *transcompilation*.

Using such a framework has many advantages. Depending on the transcompiler, the final application might be provided for example as web application, which runs in a browser. In doing so, a strongly typed language with a mature debugging system can be used. This contributes to a reduction of faults and

facilitates the bug fixing. In the area of mobile applications, it suffices to develop the application once. The transcompiler automatically creates code that is tailored for all leading mobile platforms (i.e. Android, iOS, and Windows Phone). Multiple development of the same app may be dropped. Naturally, this reduces the costs for software development and software maintenance. The same applies when a desktop application should be delivered as native app for different operating systems (e.g. Mac, Windows) that uses platform specific libraries to create user interfaces with a native look. The initial application written by the developer becomes platform independent (also called cross- or multi-platform), depending on the framework and its transcompiler.

When having a closer look at examples of transcompiling approaches, we want to point out Google Web Toolkit (GWT) [98], Haxe [130], Remote Application Platform (RAP) [70], Xamarin [286], NeoMAD [203] or Codename One [45]. GWT compiles source code from Java to JavaScript [98], RAP is based on SWT in order to create web UIs [70]. Haxe offers JavaScript, PHP, Java, and other target languages [131]. Xamarin uses source code written in C# and transcompiles it to provide native iOS, Android, or Windows Phone mobile apps [286]. Codename One and NeoMAD achieve the same by transcompiling Java source code [45, 203].

However, despite all the advantages described before, using transcompilers also entails new difficulties. Now, we look at the overall situation and the emerging consequences for transcompiled cross-platform applications.

1.1 Motivation

Testing transcompiled applications leads to new challenges in ensuring their quality and correctness. When looking at the state of the art, approaches to improve the quality of software can be categorized into static and dynamic approaches according to Sneed [259]. Today, there exist many methods whose intention is to reveal and remedy faults as soon as possible. Examples for static approaches are software reviews. These include, as described in the IEEE Standard 1028-2008 [145], technical reviews, inspections, and walk-throughs (among others). They can be applied at a very early stage in the software development even when fully executable code does not exist yet. Another example of a static approach is deductive verification. Deductive verification (e.g. [82]) uses mathematical statements and the Hoare calculus [137] to prove that an implementation is correct in terms of its specification. So naturally, this approach requires that an implementation is available and that it is at least partially finished. Finally, code metrics like non-empty lines of code (NLOC) and McCabe metric [184] as well as semantic analysis help to improve the source code. Both NLOC (also known as non-commented source statements, NCSS, see e.g. Rutar et al. [239]) and the McCabe metric provide information about the complexity of the source code.

As already reported by many authors (e.g. [3, 156, 166]) before, considering these static techniques individually does not ensure software quality and

correctness in a satisfying way. It is important to combine several static testing techniques. But these approaches are still not able to reveal all kinds of faults. For this reason, dynamic test techniques are additionally necessary. Only the combination of all these approaches affords high quality software.

Dynamic approaches include testing in the sense that the software is executed. Testing can be divided in many methods according to the phase in the software development process in which the software is tested. In an early stage, it is already possible to test small units of the software in order to check whether the result of an individual component, a function, or an algorithm matches the expected one. This is called unit testing (see also Section A.2, Paragraph "Unit Test" in the Appendix with a citation taken from the terms and definitions in the "Systems and software engineering – Vocabulary" [149]) and is a rather fast testing process. However, just testing small units is insufficient because it neither ensures the appropriate collaboration of single units nor incorporates the interaction of software modules that consist of several units. This is the goal of a later stage in the software test phase. It is commonly known as integration testing (see also Section A.2, Paragraph "Integration Testing"), which is more complex and more time consuming. Nevertheless, all these dynamic testing strategies only consider the functional part of the software. One of the most complex and time consuming testing strategies affects testing the user interface. Only hereby, it is possible to judge the overall behavior of the application and to inspect the user interface and the associated events. Depending on the kind of software application, this way of testing is called UI or web testing. Here, test engineers usually produce a test case for each use case. To this end, capture and replay tools are frequently utilized. There are many different tools such as TestComplete [258] or Ranorex [221], which are all-embracing testing tools for desktop, web, and mobile applications. Within the scope of web applications, user interactions in the browser can be recorded with the help of for example Selenium [252], HTMLUnit [89], Sahi [244], or Watir [282].

Of course, running any of these tests only once in a while still does not ensure software quality. Even small code changes have the potential to impair already existing functionality that should work as before. In order to ensure that the software still works as expected, basically all existing test cases (i.e. unit tests, integration tests as well as UI/web tests) have to be executed anew. The general term for this testing strategy is *regression testing* (see also Section A.2, Paragraph "Regression Testing" in the Appendix for the definition in the "Systems and software engineering - Vocabulary" [149]). It provides confidence into the software and helps to reveal bugs soon, which is decisive. As described by Boehm and Basili [31] and Schach [247], fixing bugs becomes more and more expensive the later they are discovered in the development process. So we can conclude that as long as a software evolves, coding and testing is an iterative process that never stops. In the style of Sepp Herberger, we can rephrase one of his statements on soccer[1] in the context of testing like this:

[1]Original statement by Sepp Herberger: "Nach dem Spiel ist vor dem Spiel." [53]; In English: "After the game is before the game."

After a test run is before the test run. Yet, the more test cases are created, the longer it takes to run the entire set of test cases. (This set of test cases is referred to as *test suite*.) In fact, regression testing accounts for a major part of the software cost: Rothermel and Harrold [233] have described that according to e.g. Schach [247], two thirds of the total development costs are spent for software maintenance. In addition, they cite (among others) Leung and White [177] who have stated that up to 50% of the maintenance cost can be allotted to regression testing. So we infer that, surprisingly, up to 33% of the software cost may go to regression testing. We even expect this value to be higher when considering web applications. The execution of web tests in a (ideally headless) browser (supported e.g. by SELENIUM [253]) involves extra time for starting-up the browser and especially for waiting for possible server responses.

To reduce the costs for regression testing, researchers have come up with new techniques, often subsumed as [290, page 68] *test suite minimization* (*TSM*), *test case prioritization* (*TCP*), and *regression test selection* (*RTS*) or selective regression testing [38, page 211] (*SRT*), respectively. We discuss these three techniques in more detail in Section 2.2.

However, none of the standard techniques that reduce the regression testing effort have been created with the special characteristics of transcompiled applications in mind. In existing approaches, developers work on the same, shared code basis that is also used by test cases. Here, it is rather straightforward to calculate which test cases might fail due to code changes. By contrast, in transcompiled applications, UI/web tests run code in a different language. For this reason, it is very difficult to determine which tests should be re-executed according to some special criteria such as changes in the code of the source programming language or due to lessons learned about existing tests and their probability to reveal a fault in the source language. Besides, it is a challenging task to locate faults in the source code when a UI/web test fails. This is especially true when the transcompiler additionally obfuscates the code. At the very least, the developer has to be an expert in all the different languages supported by the transcompiler. In case of web applications, this could be for example JavaScript or PHP (see the Haxe transcompiler [131]). In case of mobile devices, this could be both C#, Java, and Objective-C (see NeoMAD [203] or Codename One [45]).

A similar problem occurs when developers wish to determine how thoroughly the source code is covered by test cases. In order to know where additional testing is necessary, it is desirable to know the code coverage (also called test coverage) in the source programming language (see also Paragraph “Test Coverage” in the Appendix A.2). For this purpose, coverage tools usually calculate percentages that describe the coverage for special syntactical elements like statements or branches. Established tools additionally offer the possibility to highlight the source code in order to indicate the coverage. However, these reports naturally display results for the code that has been executed by a test – but this is not the code developers are interested in. In transcom-

piled applications, again, tests do not run code in the language written by the developer.

Hence, it would be desirable to use efficient regression testing techniques and code coverage techniques in transcompiled cross-plattform applications. But up to now, existing techniques are barely usable.

1.2 Summary of Contributions

The thesis contributes to three essential areas of regression testing in the special context of transcompiled applications.

1.2.1 Test Effort Reduction Problem

With the test effort reduction problem, we want to comprise three questions: (a) Which tests should run at all, (b) how to execute tests in an efficient way, and (c) which execution order would be suitable if many tests should be re-executed. Question (a) and (b) have been investigated by Rothermel and Harrold [233] in the context of selective retest techniques as the regression test selection problem (a) and as the test suite execution problem (b). Question (c) has been posed by Wong et al. [284] and expresses the problem to prioritize test cases according to specific criteria. Usually, the criteria define properties that help to decide which tests are the most important ones, for example in order to detect faults in the source code as soon as possible (see also Yoo and Harman [290]).

Up to now, there are only approaches for non-transcompiled applications available. These approaches are based on reducing test suites that run code written by developers rather than code generated by a compiler. In transcompiled applications, this precondition is not met. Our work bridges the gap between code written by developers in a programming language A and its counterpart that results from a transcompilation into a programming language B. Please keep in mind that the tests (UI or web tests) run the code of the transcompiled application – i.e. code generated in programming language B – whereas developers modify solely code written in programming language A. Thus, we are able to conclude which tests are affected by a code change (in language A). We exploit this knowledge to reduce the set of test cases used for regression testing. At the same time, we know which tests should be executed at all. This insight addresses problem (a).

Of course, the analysis has to be fast and should not exceed a certain RAM limit. For this reason, we investigate several ways to reduce the runtime and the memory consumption required for the analysis. On the one hand, we introduce a dynamically adaptable heuristics that performs analyses at different precision levels to keep the runtime and the memory consumption low. On the other hand, we propose a lookahead strategy to reveal more faults in fewer analyses. All these measures contribute to solving problem (b).

Additionally, we present and compare three novel static test prioritization techniques and three dynamic counterparts that prioritize the test execution according to a sequence of formal criteria. This will be an answer to problem (c). The results of the comparison are the basis for a recommendation which of our prioritization techniques is the most effective. Finally, we contribute to overcoming the common nightly build and test cycle. This strategy builds the complete software every evening and schedules test runs during the night in order to inform developers about bugs when they arrive the next morning. With our technique, we provide a fast executable and repeatable cycle of code changing, test determining, test case executing and bug/test case fixing that highly resembles continuous integration [84].

For easy usage, we have implemented a prototype of our technique which is available as Eclipse[2] plug-in.

Both the approaches and parts of the results of our work contributing to the test effort reduction problem have partially been published in the following papers [133–136]:

- M. Hirzel. Selective Regression Testing for Web Applications Created with Google Web Toolkit. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ 14, Cracow, Poland, pages 110-121, New York, NY, USA, 2014. ACM.

- M. Hirzel and H. Klaeren. Graph-Walk-based Selective Regression Testing of Web Applications Created with Google Web Toolkit. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016)*, Wien, pages 55-69, 2016.

- M. Hirzel, J. Brachthäuser and H. Klaeren. Prioritizing Regression Tests for Desktop and Web-Applications Based on the Execution Frequency of Modified Code. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ 16, Lugano, Switzerland, pages 11:1–11:12, New York, NY, USA, 2016. ACM.

- M. Hirzel and H. Klaeren. Code Coverage for Any Kind of Test in Any Kind of Transcompiled Cross-Platform Applications. In *Proceedings of the Second International Workshop on User Interface Test Automation*, INTUITEST '16, Saarbrücken, Germany, pages 1–10, New York, NY, USA, 2016. ACM.

1.2.2 Fault Localization Problem

The fault localization problem arises whenever a test case fails during regression testing. Developers have to investigate the code (written in the source

[2] `https://eclipse.org/`

programming language) in order to locate the fault(s) [285]. In general, this can be a very time consuming and difficult task [285]. In transcompiled cross-platform applications, this is even more complicated. Test cases run code in the target language. Thus, developers have to track and comprehend errors that emerged during the execution of a test case in order to be able to identify the faulty source code.

Our method for reducing the test effort provides crucial data to assist developers in solving this problem. At best, our approach is able to point developers to a single code change that has lead to the test failure. No further debugging is needed. At least, it determines a subset of code changes that might be the reason for the test failure.

The fault localization problem has been addressed in the following papers [133, 135]:

- M. Hirzel. Selective Regression Testing for Web Applications Created with Google Web Toolkit. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ 14, Cracow, Poland, pages 110-121, New York, NY, USA, 2014. ACM.

- M. Hirzel and H. Klaeren. Graph-Walk-based Selective Regression Testing of Web Applications Created with Google Web Toolkit. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016)*, Wien, pages 55-69, 2016.

1.2.3 Coverage Identification Problem

Tests can never prove that there are no faults in a software application. If a test passes, the only assertion is that the executed piece of code does not fail for the given input. Because of this, it is very important to have many tests that seek to cover all possible use cases with many possible inputs. That is, a test suite should be as complete as possible. The problem to decide whether a test suite is complete or which parts of the code require additional tests is known as the coverage identification problem [233].

We propose a technique that is able to calculate the coverage of any kind of test at low cost. Our technique is usable for both unit tests, integration tests, and UI/web tests. The technique supports standard desktop applications, but more importantly, it is the first that calculates the standard coverage measures (statement coverage, branch-, loop-, method-, and class coverage) for transcompiled applications and it highlights the coverage in the code written by developers. To demonstrate the easy usage in the everyday life of a developer, we provide a prototype of our technique as Eclipse plug-in.

The results of our solution for the coverage identification problem in the context of transcompiled applications has been published in the following paper [134]:

- M. Hirzel and H. Klaeren. Code Coverage for Any Kind of Test in Any Kind of Transcompiled Cross-Platform Applications. In *Proceedings of the Second International Workshop on User Interface Test Automation*, INTUITEST '16, Saarbrücken, Germany, pages 1–10, New York, NY, USA, 2016. ACM.

1.3 Outline

The thesis is structured in three main parts. In the first part, we provide background on regression testing and transcompiled cross-platform applications. As the thesis links regression testing and transcompiled applications, two topics that are independent in the first place, we will dedicate an own chapter for both fields.

Chapter 2 examines basics of regression testing in particular. We explain the main intention, general terms and formal notation. Afterwards, we depict existing categories of regression testing techniques. Then, we discuss different state-of-the art techniques which many regression testing techniques rely on. This encompasses concepts of code instrumentation, control flow graphs, and abstract syntax trees. We also differentiate fault localization from change impact analysis, another technique used in the field of regression testing. Finally, we summarize common test creation methods.

Chapter 3 covers transcompiled cross-platform applications in detail. We consider the compilation process with transcompilers in general and present transcompilers used in our evaluations as well as particularities that are important to understand subsequent decisions.

The second part addresses our generic approach for standard and transcompiled cross-platform applications. We propose a solution to the problem of reducing the test effort, localizing faults in the source code written by a developer, and identifying parts that need additional testing.

Chapter 4 presents challenges and the basic idea to solve the test effort reduction problem with an extended regression test selection technique. We elaborate two different possibilities to bridge the gap between code written by the developer and code created by the transcompiler. The results will be evaluated and considered under various aspects in order to find the most appropriate solution. In doing so, we also address the ability to locate faults precisely.

The basic idea explained in Chapter 4 is only the first step. Chapter 5 deals with possible ways to accelerate the analysis and to reduce the memory consumption. Furthermore, we show how the total number of analyses can be reduced. Again, all our concepts are discussed in an evaluation in a real-world application scenario.

To optimize the test effort further, we combine our extended test selection technique with test prioritization. In Chapter 6, we explain problems of existing prioritization techniques and propose novel alternatives that exploit the results

of our extended test selection. The advantages of our new techniques are shown in a comparison with already existing ones.

The coverage identification problem is the topic of Chapter 7. Here, we focus on how to provide common code coverage metrics for transcompiled applications and on how to indicate which source code written by developers are (un)covered by the current test suite. We demonstrate that our approach can be applied to calculate the code coverage of all the different kinds of test cases, namely unit tests, integration tests, and UI/web tests.

Finally, the third part revisits our contributions. The concluding Chapter 8 summarizes our results and presents possible directions for future research.

Part I

Background, Definitions, and Terms

Chapter 2

Basics of Regression Testing

Regression Testing is the general term for re-executing tests repeatedly. In Section 2.1, we remind the main intent of regression testing and explain preconditions first. Afterwards, we summarize in Section 2.2 three main categories of regression testing techniques. We define the basic terms, and depict benefits and drawbacks. Based on these facts, we argue which technique is suitable to support regression testing of transcompiled cross-platform applications. In Section 2.3, we discuss basic approaches for instrumenting software. We outline important terms and the basic set-up. After that, we have a look at the fundamentals of a widely used regression testing approach that is based on graphs (see Section 2.4). With these basics in mind, we introduce in Section 2.5 abstract syntax trees that can be used to create control flow graphs. In Section 2.6, we introduce fault localization and change impact analysis, explain differences, and discuss important implications. Finally, we present in Section 2.7 possible ways to create test cases and we talk about basic assumptions that are relevant for our approach.

2.1 Regression Testing: Intent and Test Categories

Main Intent and Preconditions: When adding new features to a software system, an important part of the development process is to test the new features. At the same time, it is equally important to repeatedly check that the behavior of existing parts of the system is left unchanged. This is the main intend of regression testing (see also Section A.2, Paragraph "Regression Testing"). It ensures that the extended software behaves as expected, and at the same time prevents that new functionality compromises existing parts of the program. For this purpose, a set of test cases t_j, $j \in \mathbb{N}$ is required. These test cases are usually grouped into one or several *test suites* T_i ($i \in \mathbb{N}$) and pass when applied to a specific program version P. As soon as P has been extended by new features or as soon as the implementation of some part of P has been changed, it is difficult to guarantee that the new version P' of the software works as expected. Executing all tests t_{ij} of test suites T_i regularly and observing no faults gives confidence in the system as a whole and in all features

which are covered by the tests. This strategy is called *retest-all* approach (e.g. [232, page 529]) and is the simplest way of doing regression testing.

An important precondition of regression testing is the comparability of test runs. Rothermel and Harrold [232] call this *controlled regression testing assumption* [232, page 531]. When executing tests repeatedly, it has to be ensured that the software under test has exactly the same internal state and that all parameters (such as test inputs, the database, necessary files etc.) are exactly the same as in a previous test run. Only thus, we can conclude from a failed test that existing functionality of a software program has been affected by code changes, newly added, or removed code.

Effects of Code Changes on Tests: While regression testing focuses on running existing tests to reveal bugs introduced by code modifications, it is also important to ensure that new features are tested thoroughly. For this reason, it is necessary to determine which parts of the code in a new version P' are not covered by test suites T_i. According to these findings, new test cases have to be created. In the same way, tests $t_{ji} \in T_i$ might have to be updated or might not be required any more. To distinguish the different possible effects of code changes on tests, Leung and White [177, page 63] split them in five categories: *Reusable*, *retestable*, *obsolete*, *new-structural* and *new-specification*. The first two categories subsume tests after code modifications. *Reusable* tests are not affected by changes and can be reused in the new program version P' directly. The counterpart of this kind of tests are *retestable* tests, that have to be re-executed as they are affected by code changes. The remaining three categories address on the one hand test cases that are not suitable any more to test the software. These tests are called *obsolete* tests. On the other hand, *new-structural* and *new-specifiation* tests have to be created in order to test the structure of a program or the specification in an appropriate way.

Fowler recommends in his article on *Continuous Integration* (*CI*) [84] to integrate changed source code in a *version control system* (also known as *repository*) at least once a day. He also recommends to build the system afterwards and to run test cases, because code changes could impair existing, unchanged functionality.

2.2 Testsuite Minimization, Test Prioritization, and Test Selection

In large applications with big test suites, regression testing can take several hours until results are available (e.g. [233]). Some authors even report on several days [264] or even weeks [235, 264]. Thus, this kind of testing can be very time expensive which in turn is contradictory to continuous integration [84]. According to this principle, developers should integrate their local code modifications into the main software code base as soon as a new feature has been finished or a bug has been fixed, but at least once a day. The code

base should be managed by a *version control system* in a *repository*. Each code integration is followed by building the software and running the tests [84]. The idea is that in doing so, bugs and problems occurring in the interaction of different software components can be detected earlier. Nevertheless, this requires that test results do not arrive with a massive delay as reported before. For this reason, Fowler [84] recommends multiple test stages. While the first one focuses on fast running tests, other stages run more time-consuming tests later on. We agree that this setup is helpful, but we also want to point out that the realization is difficult with large test suites or when systems have to rely on many time-consuming tests that have to run regularly.

As a way out, developers have proposed regression test approaches to reduce the testing effort and thereby shorten the time until test feedback is available. The different approaches can be divided into three main categories (e.g. [290, page 68]): *Test Suite Minimization*, *Test Case Prioritization*, and *Test Case Selection*. We explain the main ideas and the differences in the following paragraphs. For a detailed survey on existing approaches, we refer the reader to Yoo and Harman [290], Engström et al. [78], Rothermel and Harrold [232], or Orso and Rothermel [213].

Test Suite Minimization: According to Yoo and Harman [290], the objective of test suite minimization (TSM) is to run non-redundant tests only. Redundant test cases are tests t_i in a test suite T that satisfy all the same test criterion r_j ($i, j \in \mathbb{N}$). There may be many test criteria defining software quality rules. Consequently, each criterion r_j must be covered by at least one test case in order to be able to check whether a software program fulfills the software quality requirements. During the test execution, only one test case representative is executed per criterion r_j.

Test Case Prioritization: As the name indicates, test case prioritization (TCP) does not reduce the size of a test suite T. It still runs all the tests but changes the order in which individual tests are executed (e.g. [290]). The new test execution order meets a specific performance goal [71]. Usually, the goal is to find faults as soon as possible (e.g. [71, 290]). In order to meet the performance goal, many approaches are imaginable. For example, tests might be ordered according to the number of statements they execute [71]. In general, as already noticed by Li et al. [180], many approaches rely on the *Greedy Algorithms*. It tries to find for a given problem/criterion the currently best choice (e.g. [180, 262]). Li et al. call this the "next best search philosophy" [180, page 226]. At best, all elements are sorted in an optimal order after applying the Greedy algorithm for the first time. However, this is not always the case. For this reason, developers have come up with many more approaches such as the *Additional Greedy Algorithm* that incorporates data from previous iterations (e.g. [180]). We discuss more approaches in Chapter 6 and we present our own, novel prioritization strategies.

Judging the performance of TCP techniques is only possible retrospectively

by means of a metric (e.g. [290]). The reason is that we do not know which tests reveal (a) bug(s) before the tests have been executed [290]. The standard metric in the literature is called *Average of the Percentage of Faults Detected* (*APFD*) [74, page 164]. It measures the ability of an approach to detect faults as soon as possible. As explained by Elbaum et al. [74], the codomain attains values in the interval $I = [0; 100]$, where higher values indicate a better fault detection. The metric requires as input the number of tests n in a test suite T, the number m of code modifications resulting in a failure of at least one of the test cases, and an ordered test suite T'. We refer to the first test in T' that detects fault number i as TF_i. Then, the APFD metric is the result of the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

In order to have an advantage over a retest-all approach, TCP techniques will usually stop as soon as a predefined criterion (e.g. time limit) is fulfilled. However, this leads to the risk that the test run is canceled although there are still tests left which would reveal additional faults if they would have been executed. In the end, TCP techniques might fail to detect all newly introduced faults. Vice versa, it is also possible that that the test run is canceled too late. Thus, tests have been executed unnecessarily which impairs the overall test execution time.

Test Case Selection: Test selection aims to identify and to execute only these tests of the original test suite T that are affected by code changes and therefore are able to detect faults (*fault-revealing tests* [233, page 185]). After selecting tests, we obtain a new test suite T' of fault-revealing tests. More formally, it is $T' \subseteq T$ (e.g. [233]).

Ideally, the new test suite T' is significantly smaller than the original test suite T. To achieve this goal, regression test selection (RTS) techniques try to detect changes made during the development of a program version in order to determine exactly these tests which are affected by the modifications. That is, test selection techniques always consider an original program version P and a new, more sophisticated program version P' and analyze them.

To obtain knowledge about code changes made in a new program version P', Apiwattanapong et al. [12] list several approaches. An obvious and straightforward solution might be to use for example DIFF [121] from Unix or similar tools whose intent is exactly to determine textual differences between two versions of source code files. Indeed, there are several approaches that are more or less inspired by DIFF. Examples mentioned by Apiwattanapong et al. [12] are UMLDIFF, an algorithm proposed by Xing and Stroulia [287], or SRCDIFF, a combination of a special XML-format (called SRCML) with DIFF which has been proposed by Maletic and Collard [182]. For more examples, we want to refer the reader to the original paper of Apiwattanapong et al. [12].

Beyond that, there are many more different approaches that investigate how the analysis could be realized. Rothermel and Harrold [232] list in total

12 RTS techniques (namely techniques based on *Linear Equation*, *Symbolic Execution*, *Path Analysis*, *Dataflow*, *Program Dependence Graph*, *System Dependence Graph*, as well as techniques based on *Modification*, *Firewall*, *Cluster Identification*, *Slicing*, *Modified Entity*, and *Graph Walk* [232, page 548]). Some of them are *safe* [229, page 203], i.e., they select exactly all the tests affected by a code change. Consequently, they achieve the same result as a retest-all approach would do. Among the techniques investigated by Rothermel and Harrold [232], only the following RTS techniques are safe: Interprocedural Linear Equation technique, Firewall, Cluster Identification, Modified Entity, and Graph Walk technique. In the last-named technique, P and P' are represented as a graph. According to Rothermel and Harrold [232], it is the *most precise* RTS technique with respect to the ability to select tests that could be fault-revealing due to code modifications that change the output of P' (Rothermel and Harrold call this *modification-revealing* tests [232, page 530]) and to ignore tests that do not have a chance to reveal faults.

Discussing the Approaches: As already mentioned in the Introduction, an essential goal in our research is to provide a fast and efficient way to do regression testing in order to support continuous integration [84] on all testing levels, especially for UI/web tests. For this reason, we would like to run as few tests as possible. At the same time, we do not want to miss tests that are fault-revealing. Kim and Porter conclude in their paper [163] that the TSM technique applied in their evaluation achieves a high test suite reduction. Unfortunately however, it misses many faults. Chen et al. [38] argue that "in the maintenance phase changes to a system are usually small and are made to correct problems or incrementally enhance functionality. Therefore, techniques for selective software retesting can help to reduce development time" [38, page 211]. Yoo and Harman [290, page 98] remark that RTS techniques have been investigated extensively due to the safe selection of test cases. Neither minimization nor prioritization techniques have a similar concept. Yoo and Harman point out that these techniques require additional metrics to judge their fault detection ability. Besides, TCP techniques and TSM techniques may always fail to reveal faults.

For these reasons, a safe RTS technique seems to satisfy our demand the best. Retrospectively, this choice fits to recent observations by Legunsen et al. [174] who remark in a recent paper (2016) that "Regression test selection (RTS) is the most widely used approach to speeding up regression testing" [174, page 583].

Rothermel and Harrold [232] have shown that graph walk-based techniques are the most precise with respect to the ability to ignore tests that cannot reveal faults while selecting all tests that could be fault-revealing [232]. These results have additionally been confirmed by a more recent review of Engström et al. [78]. Therein, the authors try to categorize 28 regression test selection techniques and to establish an order reflecting their relative merits. They have remarked that unsafe techniques can be highly efficient in reducing the number

of test that have to rerun. However, they have also pointed out that unsafe techniques have a high risk to miss fault-revealing tests. When looking at safe techniques, they have noticed that a graph walk-based technique shows the biggest test suite reduction.

Another motivating argument for using graphs rather than DIFF-based tools has been given once more by Apiwattanapong et al. [12]. As noticed by them, DIFF-based tools are insufficient in the context of object-oriented programming languages. In a small example, they explain the main problem why a Unix-based DIFF tool is not suitable to detect code changes. We reuse their example (see Figure 2.1) and summarize their argumentation to provide a better understanding.

```
1  public class A {
2    void m1() {...}
3  }
4
5  public class B extends A {
6
7    void m2() {...}
8  }
9
10 public class E1 extends
       Exception {}
11 public class E2 extends E1 {}
12 public class E3 extends E2 {}
13
14 public class D {
15   void m3(A a) {
16     a.m1();
17     try {
18       throw new E3();
19     }
20     catch(E2 e) {...}
21     catch(E1 e) {...}
22   }
23 }
```

(a) Program version P

```
1  public class A {
2    void m1() {...}
3  }
4
5  public class B extends A {
6    void m1() {...}
7    void m2() {...}
8  }
9
10 public class E1 extends
       Exception {}
11 public class E2 extends E1 {}
12 public class E3 extends E1 {}
13
14 public class D {
15   void m3(A a) {
16     a.m1();
17     try {
18       throw new E3();
19     }
20     catch(E2 e) {...}
21     catch(E1 e) {...}
22   }
23 }
```

(b) Program version P'

Figure 2.1: Example code used by Apiwattanapong et al. [12, page 7] to illustrate problems in recognizing changes.

Figure 2.1 shows two versions of a Java program. Imagine that an external library hooks into `D.m3` of the program version P depicted in Figure 2.1a and that it passes an instance of `B` as argument. A UI test case t checks the correct functionality. So t traverses line 16 and calls `A.m1()`. In the new program version P' (see Figure 2.1b), the method `B.m1()` overrides the method `A.m1()` (see line 6). As a consequence, the behavior of P' might differ because now, t calls `B.m1()`, a piece of code that was never executed before. DIFF-like tools would just recognize the pure textual change in line 6 (`B.m1()`) and would not

be able to detect the indirectly affected parts of the code (`a.m1()` in line 16). Starting from this, it would only be possible to select tests that traverse `B.m1()`. t would be missed erroneously as it never executed `B.m1()`. If `A.m1()` has been overridden accidentally, it might neither be possible to identify nor to localize this fault in the source code.

Similarly, DIFF-based tools would not be able to recognize changes in the exception handling. In line 12 in Figure 2.1b, `E3` extends `E1` rather than `E2`. Now, exceptions thrown in line 18 could be handled in the catch-block in line 21 rather than the one in line 20. This is extremely problematic because – as observed in a study by Apiwattanapong et al. [12] – changes affect quite frequently dynamic bindings and the type of variables. They group these kinds of changes (with some less frequent kinds of changes) and call them "object-oriented changes" [12, page 19]. As a result, they have noted that even more than 50% of the changes are object-oriented changes. Tools that cannot deal correctly with those changes and their effects on tests such as t are inadequate. Moreover, Apiwattanapong et al. point out that some changes in the code even do not affect the functionality program at all. As example, they enumerate comments or reordered class members (such as functions or global variables).

Finally, there are empirical studies that report on considerable test suite reduction results when applying graph walk-based techniques [29, 78, 119, 126, 231, 232, 234] even though some of these studies also report that in some cases, the test suite reduction was low [119, 126] at rather high analysis costs [232, 234]. On top of this, as we focus on UI/web tests applied to transcompiled cross-platform applications, we have to take special factors into account such as UI/web tests that traverse larger parts of the application code. This might additionally lead to a low test suite reduction rate and a time consuming analysis (e.g. [78, 232]) which of course threatens our endeavor to achieve efficiency. But the safety argument is very important for us. Besides, as noted by Yoo and Harman [290], graph walk techniques seem to be used the most frequently in the literature. For these reasons, we take this concept as starting point for our own technique. In case of low test suite reduction rate, we would address this problems with the assistance of test prioritization in order to get an advantage over the classic retest-all approach.

2.3 Code Instrumentation

There are approaches in all three categories of regression testing techniques that require knowledge about which test cases execute which code elements/ which parts of the code (e.g. [270] (TSM); [56, 71, 153] (TCP); [126, 223, 229, 294] (RTS)). This is usually done by instrumenting code and creating traversal traces (e.g. [233]). That is, additional code is injected into the regular source code in order to create some kind of log messages. Of course, this implies an extra overhead for both adding instrumentation code and creating log messages that has to be compensated by the regression test approach in order to perform better than a classical retest-all approach.

In general, the strategy of adding instructions somewhere in a program is well known for a long time. An early variant has already been used in 1961 by Jacoby and Layton in their paper about "Automation of Program Debugging" [152] in order to analyze errors.

There are two different possibilities of instrumenting code: The first possibility is to insert instrumentation code into binaries or class files. In the second one, instrumentation code is added directly to plain source or target code. Several techniques require this instrumentation task to be done as very first step. This is called off-line or static instrumentation (e.g. [91, 140]). In contrast, other techniques perform this operation dynamically (on-the-fly, e.g. [91, 140]) at runtime. In all cases, the additional instrumentation code has to be syntactically correct and must not change the functionality of a program.

Apart from using instrumentation code for tracing code execution, there are more use cases. Code instrumentation is also used for debugging/logging (e.g. [124, 200]) and profiling (e.g. [26]); see also Appendix A.2, Paragraph "Instrumentation".

2.4 Control Flow Graph

Rothermel and Harrold [232] have found that graph walk techniques are the most precise safe techniques. Both P and P' are represented as a graph. In the literature, different kinds of graphs have been investigated. *Control flow graphs* (*CFGs*) play a very prominent role [290] and seem to be more efficient than *control dependence graphs* (*CDGs*) [233].

In this section, we define some basic terms concerning graph theory in general and control flow graphs in particular. More details about control flow graphs and their usage for representing programs can be found in the papers of Allen [8] and Rothermel and Harrold [233].

Directed Graph: A directed graph is defined as a tuple of two sets V and E. V formally represents the set of vertices. In a software program P, the vertices are *nodes* representing syntactical elements. Allen [8] introduces nodes as blocks of a program. However, we do not want to restrict nodes to blocks. Instead, we consider nodes to be any syntactical element in a program. That is, nodes may represent classes, body declarations, blocks, statements, or even expressions. E is defined by Allen as the set of edges that connects two nodes. Consequently, in a program with $|V|$ nodes, an edge e is a tuple of nodes n_i and n_j $(i, j \leq |V| \in \mathbb{N})$. More formally, it is $E = \left\{e_k = (n_i, n_j) \mid 1 \leq k \leq |V|^2\right\}$. n_i represents the start node, n_j is the target node. Therefore, Allen calls n_j the *immediate successor* of n_i. Conversely, he calls n_i the *immediate predecessor* of n_j.

A software program always has one or several starting points. This kind of node in the control flow graph is called *entry node*. It has no predecessor node. Likewise, a program has one or several end points. The corresponding nodes

are called *exit nodes*. Allen [8] denotes a control flow graph as *connected* if all nodes can be reached via at least one edge.

A directed graph $G = (V, E)$ can be divided into *subgraphs* $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. Each subgraph defines a *path* whose nodes $n' \in V'$ and whose edges $e' \in E'$. The path is expressed as sequence of immediate successors.

Modeling Programs as Control Flow Graphs via Directed Graphs: "A control flow graph is a directed graph" [8, page 2]. Therefore, it uniquely represents all nodes and edges of a program. Its paths describe in which order nodes can be traversed by a program. By instrumenting the program code, it is possible to record for each test t which edges it traverses [233]. Rothermel and Harrold denote these data as *edge trace* for t [233, page 176]. More information is also available in the Appendix A.3.

Let C be the control flow graph of a program version P. Every single modification in the source code of a program, that is new code, modified code or removed code, is reflected by the control flow graph and results in a new control flow graph C'. So both program versions are represented unambiguously by their corresponding CFGs. Depending on the modifications made in the original program P, the new graph C' differs from C in new, changed, or removed nodes and/or in new, changed, or removed edges connecting these nodes. Naturally, the new control flow graph C' shows partially different control flow paths. Rothermel and Harrold recognize changes with the aid of *execution traces* [233, page 178]: When executing a test case t on P and on P', all the traversed statements form execution traces $ET(P(t))$ and $ET(P'(t))$, respectively. If these execution traces do not coincide in their nodes and their edges, the CFGs of P and P' differ. Consequently, P' has been modified and thus, t reveals a modification. Rothermel and Harrold denote t as *modification-traversing for P and P'* [233, page 178]. This makes obvious that execution traces play an important role in the detection of new, modified, or removed code.

2.5 Abstract Syntax Tree

Basically, all the terms and definitions of tree structures apply to abstract syntax trees (AST) as well. For this reason, we summarize the most important ones and continue with syntax specific details. More details can be found for example in the book of Knuth [167], that served as basis for the summary in the next paragraph.

Structure of a Tree, Basic Terms and Definitions: An abstract syntax tree represents source code in a tree. The tree consists of *nodes* and *edges* and has always a *root node*. Each node may have zero, one or several *child nodes*. Child nodes that have the same parent node are called *siblings*. Each child node $c_1, \ldots, c_m$ of a root node r can be considered as a *subtree* $T_1, \ldots, T_m$. The

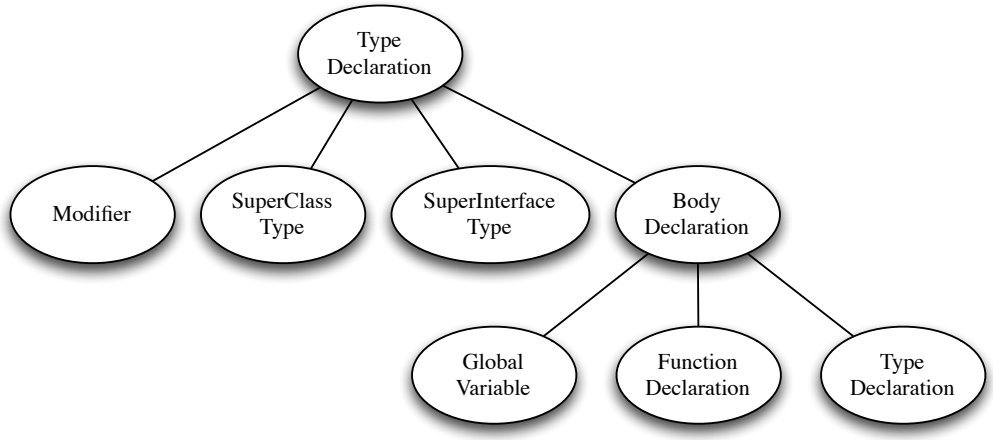


Figure 2.2: Excerpt of a tree representing a class.

root nodes of the subtrees are $c_1, \ldots, c_m$. If a node has no child node, it is called *leaf*. Nodes are connected with other nodes via edges. They describe which nodes can be reached by other nodes. Let r be the root node and let $|l_n|$ be the count of parent nodes that have to be traversed to reach the root node r starting from n. Knuth calls this value the *level* of a node in a tree. Thus, the level l_r of r is 0. Other authors rather use the term *depth* of a node instead of level. For example, Battista et al. [27] defines the depth of a node n as "the number of edges of the path" [27, page 43] of V that have to be traversed to reach n when starting from r.

Syntax specific details – Level of nodes in an AST: The root node of a tree is the most coarse-grained representation of the source code. Each child node provides more information about syntactical components of the parent node. Consequently, leaves provide the most detailed information of the source code. In class-based programming languages for example, a node representing a class may have a child node that represents the class modifier (in order to declare the visibility), a child node that represents potential super-classes or -interfaces and a child node that represents body declarations (see Figure 2.2). Body declarations may have global variables, functions, or other type declarations as children.

The source code of most programming languages can be represented by an abstract syntax tree. The underlying procedure is well-known (e.g. [256]). In order to obtain such an AST, a lexer identifies programming language specific tokens. Afterwards, a parser maps the tokens to syntactical elements of the programming language (as for example body declarations) and creates an AST. There are many tools available that create ASTs, such as the Eclipse *Java Development Tools* (*JDT*) [62, 69] or the Eclipse *PHP Development Tools* (*PDT*) [63, 68]. The information provided by an AST can be used to create a control flow graph.

In case of the programming language Java, a `.java` file in the file system corresponds to a `CompilationUnit` in the Eclipse AST, which in turn could be a Java class, interface, enum, or an annotation declaration (for JavaDoc) [65].

2.6 Fault Localization vs. Change Impact Analysis

Many authors have done research on change impact analysis (e.g. [14], [214], [224], [242], [293]). According to Ren et al. [224], change impact analysis

> "[...] consists of a collection of techniques for determining the effects of source code modifications, and can improve programmer productivity by: (i) allowing programmers to experiment with different edits, observe the code fragments that they affect, and use this information to determine which edit to select and/or how to augment test suites, (ii) reducing the amount of time and effort needed in running regression tests, by determining that some tests are guaranteed not to be affected by a given set of changes, and (iii) reducing the amount of time and effort spent in debugging, by determining a safe approximation of the changes responsible for a given test's failure [...]." [224, page 432]

In contrast, according to Yu et al. [292], the main concern of fault localization is to

> "direct developer attention to likely faulty locations, and thus, reduce the expense of the search." [292, page 201]

In the literature, there exists lots of work on fault localization. Wong et al. have defined eight categories of fault localization techniques in their review [285], namely "slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based and miscellaneous techniques" [285, page 710]. At first sight, slice-based techniques could be interesting as they are usable for test selection, too. But as already explained in Section 2.2, slicing techniques are not safe and therefore, we have decided to refrain from using this kind of technique to do the regression test selection. Applying a slicing technique to solve the fault localization problem (see Section 1.2.2) would involve an additional analysis which causes extra overhead (see also Section A.2 in the Appendix, Paragraph "Program Slicing", Paragraph "Backward Static Slices and Forward Static Slices", and Paragraph "Dynamic Slicing" for more information about slicing). For this reason, we localize faults solely using information that can be obtained from our graph walk technique. This ensures that the overhead to perform the fault localization is negligible. Having regard to these deliberations, our approach to localize faults seems either to belong to the model-based technique or to the miscellaneous techniques in the categorization of Wong et al. [285]. The authors give no strict definition, but as we use a graph to model two versions of a program

that are checked for differences, we would rate our fault localization approach among the model-based techniques. Among the other techniques mentioned above, spectrum-based approaches could be used to enhance our own technique with little effort. More details follow in Section 4.2 and in Section 8.2.

In our research, we only address fault localization. Nevertheless, as change impact analysis can be used to localize faults, we also mention related work on this topic (see Section 4.2).

2.7 Test Case Creation

Basically, our approach to deal with fault localization, test effort reduction, and coverage identification is independent from a concrete test tool. Every test tool is fine as long as it is able to run a test case in the respective target language. Whether the test cases are generated automatically or manually does not matter for us as we do not investigate the time for generating test cases.

To ensure the correctness of software, different software testing methods have been introduced. We summarize the methods that are the most important for our research:

Unit Tests: As long as we want to check the correctness of single functions, unit tests are suitable. Examples are representatives of the family of XUnit frameworks (e.g. [85]) such as CppUnit [88] (C/C++), NUnit [205] (C#, including support for Xamarin), csUnit [10] (C#, C++ and others), JUnit [160], TestNG [273] (both Java), PHPUnit [249] (PHP), UnitJS [271] (testing client side JavaScript functions), and many more. See also Section A.2 in the Appendix, Paragraph "Unit Test".

Capture/Replay: In order to ensure that the user interface behaves as expected, researchers have focused specifically on capture and replay. It is an easy way to create UI/web tests. These tests can be used for both system testing and acceptance testing (see also Section A.2 in the Appendix, Paragraph "Acceptance Test" and Paragraph "System Testing"). As already mentioned in Section 1.1, TestComplete [258] is a suitable testing tool for both desktop, web, and mobile applications. Besides, there also exist special approaches such as the one presented by Joshi and Orso [158]. It enables the user to capture specific subsystems in order to reduce the amount of data that has to be processed during program execution.

When focusing on pure web applications, Selenium [252], HTMLUnit [89], Sahi [244], or Watir [282] are convenient. Especially Sahi seems to be superior according to the feature matrix published by the maintainers [243]. The tool Mugshot [189] overcomes problems with capturing non-deterministic events. To the same effect like Sahi, this might be an alternative to Selenium. Nevertheless, we follow the recommendation made by Google in the context of GWT [34] and use Selenium for our web tests.

Chapter 3

Basics of Transcompilation

We already introduced transcompilation as process that compiles the source code from one programming language into another programming language (see Introduction, Chapter 1). In this chapter, we will have a closer look at transcompilers. In the first Section 3.1, we define some basic terms concerning the input and the output of a transcompiler. Additionally, we compare different output styles. Afterwards, we discuss typical code optimizations that are often performed by transcompilers. Then, we talk about the meaning of the term cross-platform application. At the end of Section 3.1, we consider source maps as an approach to debug transcompiled applications.

In Section 3.2, we investigate transcompilers which we use in our evaluations. We summarize several properties and outline peculiarities like the basic structure of an application or the requirements that have to be met in order to create specific versions for different platforms.

3.1 Transcompilers

Basics: A *transcompiler*, also known as *source-to-source translator* (e.g. [6, page 3]) or source-to-source compiler, always takes source code (usually written by a developer) in a programming language A as input. We refer to this language as the *source (programming) language*[1] like others before (e.g. [6]). The output of the transcompiler is always source code in one or several languages B_i ($i \in \mathbb{N}$). Analogously, we call these languages *target (programming) languages*[2]. Aho et al. and we refer to the resulting code as *target code* [6, page 9] for easy distinction. The arising application is called *target application* or – in the book of Aho et al. [6] – "target program" [6, page 2]. Accordingly, transcompiled applications (also known as source-to-source compiled applications) are written in a specific source programming language in the first place before a transcompiler compiles the source code in another target language to meet special requirements.

[1] Aho et al. [6] just use the term "source language" [6, page 1] without emphasizing *programming* languages in particular.

[2] Accordingly, Aho et al. [6] simply talk about a "target language" [6, page 1].

```
1  private void addStock() {
2    final String symbol = newSymbolTextBox.getText().toUpperCase().trim
         ();
3    newSymbolTextBox.setFocus(true);
4    if (!symbol.matches("^[0-9a-zA-Z\\.]{1,10}$")) {
5      Window.alert("'" + symbol + "' is not a valid symbol.");
6      newSymbolTextBox.selectAll();
7      return;
8    }
9    newSymbolTextBox.setText("");
10   if (stocks.contains(symbol))
11     return;
12   int row = stocksFlexTable.getRowCount();
13   stocks.add(symbol);
14   stocksFlexTable.setText(row, 0, symbol);
15   stocksFlexTable.setWidget(row, 2, new Label());
16   stocksFlexTable.getCellFormatter().addStyleName(row, 1, "
         watchListNumericColumn");
17   stocksFlexTable.getCellFormatter().addStyleName(row, 2, "
         watchListNumericColumn");
18   stocksFlexTable.getCellFormatter().addStyleName(row, 3, "
         watchListRemoveColumn");
19   Button removeStockButton = new Button("x");
20   removeStockButton.addStyleDependentName("remove");
21   removeStockButton.addClickHandler(new ClickHandler() {
22     public void onClick(ClickEvent event) {
23       int removedIndex = stocks.indexOf(symbol);
24       stocks.remove(removedIndex);
25       stocksFlexTable.removeRow(removedIndex + 1);
26     }
27   });
28   stocksFlexTable.setWidget(row, 3, removeStockButton);
29   refreshWatchList();
30 }
```

Figure 3.1: Code excerpt taken from STOCKWATCHER [100].

When transcompiling source code, the code of a target application may look completely different depending on the settings of the transcompiler. Many transcompilers allow to *obfuscate* the code of the target language (see for example the GWT compiler [112]). That is, the source code is not human readable any more for business secret protection reasons. The other extreme is to add additional information to the standard, non-obfuscated transcompiler output about the source code (e.g. verbose variable names in case of GWT [112]). Figure 3.2, Figure 3.3, and Figure 3.4 show all three possible variants of transcompiled results when compiling a piece of Java code into JavaScript using the GWT compiler. Figure 3.1 depicts the excerpt of the original Java code. The code uses both standard Java- and GWT-APIs and it is taken from a small AJAX-based example web application called STOCKWATCHER [100] that is contained in the GWT tutorials. It allows users to add new stocks and to

```
1  function qk(a){
2    var b,c,d;
3    d=Fp(Eb(Ml(a.e),Ut).
4          toUpperCase());
5    xm(a.e);
6    if(!(new RegExp('^(^[0-9a-zA-Z\\\\.]{1,10}$)$')).test(d)){
7      $wnd.alert(\"'\"+d+\"' is not a valid symbol.\ ");
8          Rn(a.e);
9      return
10   }
11   Tn(a.e);
12   if(kr(a.f,d,0)!=-1)
13     return;
14   c=Nm(a.g);
15   ir(a.f,d);
16   Wm(a.g,c,0,d);
17   Xm(a.g,c,2,new En);
18   bn(a.g.b,c,1,Vt);
19   bn(a.g.b,c,2,Vt);
20   bn(a.g.b,c,3,Wt);
21   b=new Bm('x');
22   Pl(b,Rl((Ok(),b.n))+'-'+'remove');
23   Ul(b,new Gk(a,d),(mc(),mc(),lc));
24   Xm(a.g,c,3,b);sk(a)
25 }
```

Figure 3.2: Transcompiled code: `OBFUSCATED` variant.

display them on the screen. For each stock, the application generates an initial random prize that changes in predefined time intervals. Dependent on the change delta, the change rate is colored green or red, indicating a rise or a decline, respectively.

The `OBFUSCATED` variant in Figure 3.2 corresponds to the standard output of the GWT compiler [112]. We have added line breaks to make it more readable, but it has almost no resemblance with the original code written in the source language (see Figure 3.1). Only the existence of some Strings (e.g. line 7, `is not a valid symbol`) that appear only once in the whole StockWatcher application has enabled us to identify the function `qk` as pendant to the original `addStock`-method in Figure 3.1. If we are interested in readable code, passing the `PRETTY`-style flag to the compiler results in code that is comparable to the original code (see Figure 3.3). Finally, the `DETAILED`-style flag results in verbose target code (see Figure 3.4) whose functions and types consist of globally qualified names. Examples are `java_lang_String` or `com_google_gwt_sample_stockwatcher_client_StockWatcher_newSymbolTextBox`.

Code Optimization: Often, transcompilers (such as the GWT compiler [181] or the Haxe compiler [128]) do not just transfer a source language into a

```
function $addStock(this$static){
  var removeStockButton, row, symbol;
  symbol = $trim($getPropertyString($getElement(this$static.
        newSymbolTextBox), 'value').toUpperCase());
  $setFocus(this$static.newSymbolTextBox);
  if (!(new RegExp('^([0-9a-zA-Z\\.]{1,10})$')).test(symbol)) {
    $wnd.alert("'" + symbol + "' is not a valid symbol.");
    $selectAll(this$static.newSymbolTextBox);
    return;
  }
  $setText_1(this$static.newSymbolTextBox);
  if ($indexOf_2(this$static.stocks, symbol, 0) != -1)
    return;
  row = $getDOMRowCount(this$static.stocksFlexTable);
  $add_4(this$static.stocks, symbol);
  $setText(this$static.stocksFlexTable, row, 0, symbol);
  $setWidget(this$static.stocksFlexTable, row, 2, new Label);
  $addStyleName(this$static.stocksFlexTable.cellFormatter, row, 1,
        'watchListNumericColumn');
  $addStyleName(this$static.stocksFlexTable.cellFormatter, row, 2,
        'watchListNumericColumn');
  $addStyleName(this$static.stocksFlexTable.cellFormatter, row, 3,
        'watchListRemoveColumn');
  removeStockButton = new Button('x');
  $setStyleName(removeStockButton,
        getStylePrimaryName(($clinit_DOM(),
        removeStockButton.element)) + '-' + 'remove');
  $addDomHandler(removeStockButton, new StockWatcher$4(
        this$static, symbol), ($clinit_ClickEvent() ,
        $clinit_ClickEvent() , TYPE));
  $setWidget(this$static.stocksFlexTable, row, 3, removeStockButton);
  $refreshWatchList(this$static);
}
```

Figure 3.3: Transcompiled code: `PRETTY` variant.

target language. Instead, they analyze the source code language and optimize it. Some compilers (e.g. the GWT compiler [181]) even perform multiple optimization steps. The first optimization can start even before a single line of the source code has been transcompiled in the target programming language. Typical tasks that might be performed by a transcompiler (e.g. the GWT compiler [181]) might include *unreachable code elimination*, *dead code elimination*, *function inlining*, or *constant folding* and *constant propagation*. After the transcompilation, there is often potential to improve the target code further. So, a second round of optimization might start in order to do similar code changes as in the first optimization phase. To have a common understanding of the optimizations listed above, we explain them in more detail. We follow in our explanations Muchnick. More details can be found in his book [201].

Unreachable code elimination: During the compilation process, the compiler ignores code that can never be traversed.

```
1  function com_google_gwt_sample_stockwatcher_client_StockWatcher_$addStock__Lcom_google_gwt_sample_stockwatcher_client_StockWatcher_2V(this$static){
2    var removeStockButton, row, symbol;
3    symbol = java_lang_String_$trim__Ljava_lang_String_2Ljava_lang_String_2(
           com_google_gwt_dom_client_Element_$getPropertyString__Lcom_google_gwt_dom_client_Element_2Ljava_lang_String_2Ljava_lang_String_2(
           com_google_gwt_user_client_ui_UIObject_$getElement__Lcom_google_gwt_user_client_ui_UIObject_2Lcom_google_gwt_user_client_Element_2(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_newSymbolTextBox), $intern_27).toUpperCase());
4    com_google_gwt_user_client_ui_FocusWidget_$setFocus__Lcom_google_gwt_user_client_ui_FocusWidget_2ZV(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_newSymbolTextBox);
5    if (!(new RegExp('^([0-9a-zA-Z\\.]{1,10}$)$')).test(symbol)) {
6      $wnd.alert("'" + symbol + "' is not a valid symbol.");
7      com_google_gwt_user_client_ui_ValueBoxBase_$selectAll__Lcom_google_gwt_user_client_ui_ValueBoxBase_2V(this$static.
             com_google_gwt_sample_stockwatcher_client_StockWatcher_newSymbolTextBox);
8      return;
9    }
10   com_google_gwt_user_client_ui_ValueBoxBase_$setText__Lcom_google_gwt_user_client_ui_ValueBoxBase_2Ljava_lang_String_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_newSymbolTextBox);
11   if (java_util_ArrayList_$indexOf__Ljava_util_ArrayList_2Ljava_lang_Object_2II(this$static.com_google_gwt_sample_stockwatcher_client_StockWatcher_stocks, symbol, 0) != -1)
12     return;
13   row = com_google_gwt_user_client_ui_HTMLTable_$getDOMRowCount__Lcom_google_gwt_user_client_ui_HTMLTable_2I(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable);
14   java_util_ArrayList_$add__Ljava_util_ArrayList_2Ljava_lang_Object_2Z(this$static.com_google_gwt_sample_stockwatcher_client_StockWatcher_stocks, symbol);
15   com_google_gwt_user_client_ui_HTMLTable_$setText__Lcom_google_gwt_user_client_ui_HTMLTable_2IILjava_lang_String_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable, row, 0, symbol);
16   com_google_gwt_user_client_ui_HTMLTable_$setWidget__Lcom_google_gwt_user_client_ui_HTMLTable_2IILcom_google_gwt_user_client_ui_Widget_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable, row, 2, new com_google_gwt_user_client_ui_Label_Label__V);
17   com_google_gwt_user_client_ui_HTMLTable$CellFormatter_$addStyleName__Lcom_google_gwt_user_client_ui_HTMLTable$CellFormatter_2IILjava_lang_String_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable.com_google_gwt_user_client_ui_HTMLTable_cellFormatter, row, 1, $intern_28);
18   com_google_gwt_user_client_ui_HTMLTable$CellFormatter_$addStyleName__Lcom_google_gwt_user_client_ui_HTMLTable$CellFormatter_2IILjava_lang_String_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable.com_google_gwt_user_client_ui_HTMLTable_cellFormatter, row, 2, $intern_28);
19   com_google_gwt_user_client_ui_HTMLTable$CellFormatter_$addStyleName__Lcom_google_gwt_user_client_ui_HTMLTable$CellFormatter_2IILjava_lang_String_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable.com_google_gwt_user_client_ui_HTMLTable_cellFormatter, row, 3, $intern_29);
20   removeStockButton = new com_google_gwt_user_client_ui_Button_Button__Ljava_lang_String_2V('x');
21   com_google_gwt_user_client_ui_UIObject_$setStyleName__Lcom_google_gwt_user_client_ui_UIObject_2Ljava_lang_String_2ZV(removeStockButton,
           com_google_gwt_user_client_ui_UIObject_getStylePrimaryName__Lcom_google_gwt_dom_client_Element_2Ljava_lang_String_2((com_google_gwt_user_client_DOM_$clinit__V() ,
           removeStockButton.com_google_gwt_user_client_ui_UIObject_element)) + '-' + 'remove');
22   com_google_gwt_user_client_ui_Widget_$addDomHandler__Lcom_google_gwt_user_client_ui_Widget_2Lcom_google_gwt_event_shared_EventHandler_2Lcom_google_gwt_event_dom_client_
           DomEvent$Type_2Lcom_google_gwt_event_shared_HandlerRegistration_2(removeStockButton, new
           com_google_gwt_sample_stockwatcher_client_StockWatcher$4_StockWatcher$4__Lcom_google_gwt_sample_stockwatcher_client_StockWatcher_2V(this$static, symbol), (
           com_google_gwt_event_dom_client_ClickEvent_$clinit__V() , com_google_gwt_event_dom_client_ClickEvent_$clinit__V() , com_google_gwt_event_dom_client_ClickEvent_TYPE));
23   com_google_gwt_user_client_ui_HTMLTable_$setWidget__Lcom_google_gwt_user_client_ui_HTMLTable_2IILcom_google_gwt_user_client_ui_Widget_2V(this$static.
           com_google_gwt_sample_stockwatcher_client_StockWatcher_stocksFlexTable, row, 3, removeStockButton);
24   com_google_gwt_sample_stockwatcher_client_StockWatcher_$refreshWatchList__Lcom_google_gwt_sample_stockwatcher_client_StockWatcher_2V(this$static);
25 }
```

Figure 3.4: Transcompiled code: `DETAILED` variant.

Dead code elimination: Unused variables and instructions whose resulting values are never used again are called dead. Basically, dead code might have been introduced by the developer. But it can also be a result of previous code optimizations. A major advantage of dead code elimination in our special context is that the compiler has to transcompile less code and that the target code requires less memory.

Function inlining, also known as procedure integration, substitutes a function call by the body of the called function. Muchnick especially emphasize the benefit of being able to do further optimizations within the calling function and to avoid aliasing of variables.

Constant folding: Whenever an expression represents a calculation that consists of fixed operands, the compiler can evaluate the expression in order to replace it by its resulting value.

Constant folding is beneficial because it is known from compiler theory for a long time (see for example the book of Aho et al. [5]) that the operands usually would be pushed onto a stack and afterwards evaluated. If the result is required for further calculations, it will be pushed onto the stack again. Compared to the optimization, pushing and pulling operands onto/from the stack takes more time especially when the operation is performed several times.

Constant propagation is the natural continuation of constant folding. The idea is to substitute variables by their values provided that these values are known to be constant by definition or that they have been obtained by previously performed constant folding.

Sometimes, the user does not want all the optimization features as a whole. In order to leave more freedom, transcompilers often offer flags to control the code optimization level [108]. GWT even offers to turn it completely off (level 0) [108]. Other transcompilers like Haxe also offer similar settings (`no-opt` to disable optimizations) [128, 129].

Cross-Platform Applications: According to the Sun Microsystems [268], the term cross-platform application describes the possibility to run an application on different platforms.

In the area of desktop applications, we might think of Linux, Mac OS, or Windows. Applications written for one of these operating systems can never be executed on one of the other systems directly. If we want an application to run on more than one *(target) platform*, one possibility is to re-implement it separately. In fact, there are many examples of applications that are implemented for different operating systems individually. We would like to point the reader for example to Microsoft Office that is available for both Mac OS and Windows as different applications.

There exist well-established alternatives to maintaining different code bases. A famous example is Java that compiles source code into bytecode. Provided that a version of the Java Virtual Machine is installed on the target operating system, the application can be executed.

Another alternative is a web application. Here, the web browser takes on the same role as the Java Virtual Machine to run the desired application on any platform. In practice however, it happens to be more complicated as browsers sometimes differ in supported features and their way to display contents.

Finally, the third alternative is based on transcompilers. It depends solely on the transcompiler which platforms are supported. The application in the target language might run in a completely different environment. That is, the application could be provided for example as web application that runs in a browser, as a mobile application, or as desktop application. This skill demarcates transcompilers from common compilers that do not support other platforms at all. Their intend is for example to do code optimizations or code obfuscation. An example for a compiler that optimizes code is Google's Closure Compiler [110].

Source Maps: Source Maps are a possibility to debug transcompiled code. As per the general source map standard [175], Mozilla's Source Map Library [199], and an article of Fitzgerald and Nyman[3] [83], a source map contains mapping data to display the counterpart of a transcompiled piece of target code in the original source code. To do that, the transcompiler collects information about the source code while parsing the source files. Elements of interest are the name of original source file and the location of the source code within this file (usually the line number and the column). All these information will be persisted in a separate `*map` file. Examples of such files can be found in the Appendix B, Figure B.1 and Figure B.2.

Please note that source maps are no solution to the test effort reduction problem (see Section 1.2.1). Initially, source maps are designed to be used in a running program to display the source code that corresponds to the currently executed target code. But the pure mapping does not provide any information about the relevance of tests for re-execution. Besides, we never want to rerun the entire test suite T on the current program version P'. When considering a previous version P, it is obvious that the test suite T will pass. Otherwise, P would never have been successfully built and deployed. Finally, we can neither conclude from the source map of P nor from the source map of P' which test runs which parts of the application, although this is essential for all categories of regression testing techniques. So as we can see, source maps are only suitable for code debugging, but not to do regression testing. Nevertheless, we will use parts of their implementation as an aid to solve the test effort reduction problem.

3.2 Basics on Transcompilers Used to Investigate our Contributions

As we will show in the following chapters, our contributions are independent of special transcompilers. However, in order to investigate our approach, its

[3]Fitzgerald is one of the source map standard contributors.

ability to reduce a test suite and the overall performance, we have selected several test applications. Here, we focus on GWT-based web applications for two reasons: First, the GWT compiler is open source and can easily be adapted according to our needs. Second, there are several small-, mid- and large-scale GWT-based web applications available that can be used to investigate our approach. Especially large applications help us to judge the performance of our approach.

Nevertheless, we also discuss our approach in the context of Codename One's transcompiler that is able to create mobile apps for all major mobile platforms. This way, we make clear that our approach also works for these use cases. In the introduction, we also mentioned Haxe and Xamarin as examples that use a transcompiler. However, our approach is currently implemented for applications written in Java. For this reason, we will neither investigate Haxe-based applications nor applications implemented with Xamarin as they require code written in Haxe and C#, respectively. But in general, our approach could also be used for these kinds of application. We will discuss this later in Section 4.9. In order to understand the features of GWT and Codename One better, we will have a closer look at the two transcompilers in the following paragraphs.

Google Web Toolkit: In order to join the advantages of web applications (see Chapter 1) and the advantages known from statically typed languages, Google has developed the Google Web Toolkit [98] as an in-between framework for creating AJAX-based web applications. It resembles the JavaServer Faces framework [209] in the sense that it distinguishes between client-side and server-side code, but unlike Java ServerFaces, both server- and client-side code is written in Java as strongly typed language with a mature debugging system. As a result, this eases the programming and testing process [34, 101, 104]. The server-side code is compiled into Java bytecode as usual. But the client-side code is transformed into JavaScript using a *Java-to-JavaScript compiler* (*GWT compiler*) [95] which uses directly the Java source code to perform its task. Bytecode is never considered [33, 35]. Code which is shared among server and client will be compiled in either version. Within the client-side code, GWT explicitly allows developers to write native JavaScript code within Java code. Up to now, this is done with the aid of the JavaScript Native Interface [97]. In the latest GWT release, a new concept has been introduced which is called JsInterop [94].

The GWT compiler creates several versions of the web application. The versions are called *permutations* [102]. Each permutation is tailored for a specific browser and even addresses the peculiarities and possible known bugs of the different browser versions. By default, the code of the whole application is written in a single file per permutation. Of course, this may lead to function name clashes when a method with the same name has been defined in several Java classes. In the same sense, variables might be shadowed or overridden. During the transcompilation process, the GWT compiler has to take care about

those potentially emerging issues. Basically, the GWT compiler resolves this with the aid of globally qualified class names.

Apart from the typical optimizations we already mentioned before, the GWT compiler additionally performs some more language specific tasks. For example [181], it converts monomorphic calls into static calls, and replaces super-types or method calls to more specific ones. Moreover, the GWT compiler normalizes the Java code. For string and array calls, this means that trampolines to native JavaScript code are inserted into the source code. As there are no types in JavaScript, multiple Java catch blocks that differ in their type argument are transformed to case distinctions using instanceof-checks. The former catch block remains internally unchanged. Later on, instanceof checks and specific type casts are replaced by a call to the runtime library. Further replacements are performed for array operations, equality checks and `getClass()`. After transferring the Java code into JavaScript, the GWT compiler performs again optimizations and normalizations on the JavaScript code (e.g. removing unused code, inlining of methods or substituting parameters with their evaluated constant).

The distinction between client- and server-side code and the special treatment of code used on both client- and server-side is also reflected in the standard directory and package layout of a web application via accordingly named packages. This structure is obligatory according to the documentation [113]. There are many more conventions such as predefined folders (src, test, war). Thus, GWT is an example of the convention over configuration design paradigm. Besides, the documentation [115] points out that a GWT application has its own entry point method `onModuleLoad` which is called when launching the application. (In regular Java applications, the entry point is the `main` method.)

Following the GWT conventions is straightforward when developers use their favorite integrated development environment (IDE) with a current version of the GWT plug-in. It also eases testing. GWT comes with special testing support [34, 101]. It is shipped with an extension of the JUnit test framework [160] and allows to write `GWTTestCase`s which enable asynchronous testing with remote procedure calls and testing of native JavaScript code (e.g. widgets, functions) [101]. To this end, GWT relies on HtmlUnit [89], a browser without user interface (also called "headless" [34]). However, as soon as user interactions have to be tested, Selenium [252] tests are recommended to do web testing [34]. So, we can summarize that GWT makes contributions to programming, debugging, and testing, but there is no special support for web tests in general or regression testing in particular. It does not reduce the time exposure for testing the user interface via UI/web tests.

Apart from facilitating the programming and testing process, debugging is much simpler as it would be in a traditional web application development. Usually, developers rely on tools like Firebug [198] to step through the JavaScript code and to investigate the actual state of the application. JSFiddle [159] can be used to check and run small JavaScript components. As described in a recent article [109], GWT has introduced with version 2.7 a special mode, called

Super Dev Mode. It takes advantage of source maps. This way, the browser's internal debugger is able to keep track of most of the underlying Java code that corresponds to the currently in a browser executed JavaScript code. (An exception affects Java field names and values. Even with source maps, only JavaScript variables and values can be displayed to the user [109].) Exemplary excerpts for source maps generated by GWT can be found in Appendix B in Figure B.1 and in Figure B.2.

GWT is used in different products by Google itself [50], but also by many other companies [269]. Moreover, other frameworks for creating web applications such as Vaadin [279] or Sencha GXT [255] use GWT.

Codename One: Codename One supports many different mobile platforms [47]. These include Android, Blackberry, iOS, and Windows Phone. Some platforms have special demands like the presence of platform specific tools. We might think of applications targeting at iOS. Building these apps requires a Mac with XCode [47]. So a transcompiler that is just able to transfer the source code into a target language is not sufficient. For this reason, Codename One offers their transcompiler as an online service in order to be platform independent. It first transcompiles the application and afterwards starts a build for the corresponding target platform. As input, the transcompiler expects Java bytecode that has to be transmitted as `jar` file. The output will be the final application that is ready for deployment. Similar to GWT, Codename One [46] also provides options to decide whether code should be obfuscated.

Regarding the implementation of an application with Codename One, developers can use their IDE of choice [48]. In combination with a special plug-in provided by Codename One [47], developers are able to simulate and debug their application without prompting Codename One's build server to do a full rebuild after every single code change. Besides, the framework [49] supports unit testing and offers a simple test recorder to check whether the functionality meets the expectations.

So basically, developers do not repeatedly have to switch the operating system in order to build the application for multiple platforms. Of course, this goes hand in hand with the necessity to upload the application's `jar` file to Codename One's build server each time when a new build is due for deployment. This may be time consuming depending on the file size. Beyond that, people might feel uncomfortable loosing control what happens to the bytecode.

Part II

Generic Approach for Standard and Transcompiled Cross-Platform Applications

Chapter 4

Graph Walk-based Test Selection

4.1 Introduction

As already explained in Section 2.2 (see Paragraph "Discussing the Approaches"), DIFF-based tools are not well-suited for regression testing and fault-localization in the context of applications written in object-oriented source programming languages. We have also explained that among test suite minimization, test case prioritization, and regression test selection, the latter one is the most promising to achieve our purposes. However, existing graph walk-based approaches can not be directly applied to transcompiled cross-platform applications due to their special characteristics. In general, regression test selection has only been applied to web tests by a few up to now in order to reduce the test effort. Especially graph walk-based techniques are barely applied to speed up the regression test execution in (transcompiled) web applications and to localize faults. This is also reflected in the review of existing techniques for web application testing of Doğan et al. [60]. To the best of our knowledge, we are the first who apply a RTS technique based on a control flow graph (CFG) on transcompiled cross-platform applications in general and on transcompiled web applications in particular.

In the next Section 4.2, we present related work that deals with regression testing, test selection, and fault localization. Our work shows also similarities to change impact analysis and code instrumentation. So we additionally have a look at these topics. In Section 4.3, we discuss a graph walk-based test selection technique proposed by Rothermel and Harrold [233], as well as an extended version created by Harrold et al. [126]. We point out deficiencies that prevent a direct usage of the existing approaches in transcompiled cross-platform applications and we present our own approach that is built on the previously discussed graph walk-based RTS technique and its extension.

Applying our RTS technique requires a solution for adding or transferring instrumentation code into the code of the target language. In Section 4.4, we discuss possible ways of instrumenting source code and emerging challenges.

Based on this, we detail our own requirements and purposes and refine an existing principle that results in two different ways to add instrumentation code. The first one is compiler-independent, but requires a preprocessor to do the code instrumentation. It is the topic of Section 4.5. The second way is compiler-dependent and instruments the source code on the fly during the transcompilation. We explain this idea in Section 4.6. The first approach has been implemented in a prototype tool which we present in Section 4.7. We have investigated the tool in a first evaluation that is part of Section 4.8. We continue in Section 4.9 with a discussion of the general applicability of the two code instrumentation approaches, the effort to support other transcompilers, the effort to support other programming languages, and the ability to solve both the test effort reduction problem (see Section 1.2.1) and the fault localization problem (see Section 1.2.2). Finally, we conclude in Section 4.10 which instrumentation approach is the most convenient for our purposes.

This chapter is partially based on research and findings presented in former publications [133–136].

4.2 Related Work

Our work touches related topics in several areas. No matter what kind of application we consider (desktop application, web application, or mobile application, transcompiled or not), solving the test effort reduction problem and the fault localization problem shares the old and well known problem (e.g. [28]) of identifying changes in the program and the problem of identifying those tests that are affected by the changes. So first of all, our work is partially based on formerly presented techniques to do regression testing in standard desktop applications. Afterwards, we additionally consider regression testing of web applications. Then, we talk about code analysis. Here, we are especially concerned with fault localization, but we also present some existing approaches for doing change impact analysis. Finally, from a technical point of view, our work also builds on previous work that addresses program tracing in general and code instrumentation in particular, which is a well-known technique often applied to get information about program execution or to determine code coverage.

In this section, we try to outline existing contributions in the aforementioned topics. Of course, it is just an overview as it is impossible to cite all relevant work in all these topics. Please note that none of these papers solve the problems described in the contributions of this thesis (see Section 1.2) in the special context of transcompiled applications.

Regression Test Selection: For non transcompiled, non web applications, RTS is a well studied topic and many different approaches (e.g. [12, 25, 29, 38, 81, 87, 93, 119, 126, 142, 204, 208, 227–234, 236, 238, 248, 280]) have been developed to detect changes in the code and to determine all the tests that have to be re-executed due to these changes. We summarize in this paragraph the most important ones. For a more extensive overview on regression test

selection techniques, we want to refer the reader to existing reviews that have been conducted by Engström et al. [78], Orso and Rothermel [213], Rothermel and Harrold [232], or Yoo and Harman [290].

Chen et al. [38] present a safe RTS technique called TestTube. It divides the source code of an old version P of a C program into entities which correspond to functions, variables, types, and preprocessor macros. Afterwards, it runs test units to monitor which entities they execute. When dividing a modified version P' in the same way into entities as P before, TestTube is able to determine which entities have been modified and which test units should be re-executed. Chen et al. need some preconditions in order to make their tool work. Due to these preconditions, only languages without pointer arithmetic and type coercion are suitable.

Rothermel, Harrold and colleagues have published a series of articles with high impact on regression test selection [126], [227], [228], [229], [230], [231], [232], [233], [234], [236]. Initially, the approach of Rothermel and Harrold [228] to do regression test selection has been based on the *control dependence graph* (*CDG*) [228, page 358]. They use one control dependence graph each to represent the old and the new version of a program. Then, they walk through the graphs and compare the corresponding nodes. According to the authors, a major improvement compared with previous publications is that they are able to determine all the tests that could be fault-revealing. Furthermore, they state that their algorithm is more efficient. Later [231], Rothermel and Harrold have improved their technique with respect to precision. According to them, their test selection is safe at a smaller number of test cases that should rerun. Moreover, they distinguish *intraprocedural* and *interprocedural* regression testing [231, pages 171f. and 178f.]. The basic difference is that the intraprocedural variant only considers functions in isolation whereas interprocedural regression testing considers a program as a whole. For the intraprocedural analysis, Rothermel and Harrold [231] now apply a *program dependence graph* (*PDG*) [231, pages 171f.] that incorporates, according to Ferrante et al. [81], both data and control relationships. (A more detailed explanation of the term PDG can be found in the Appendix A.3, Paragraph "Program Dependence Graph".) In contrast, the interprocedural analysis is based on a *system dependence graph* (*SDG*) [231, pages 179f.] which adds some more nodes, but most notably additional edges to model the interprocedural data dependencies. (For SDGs, see Appendix A.3, Paragraph "System Dependence Graph".) Rothermel and Harrold [233] have refined the overall technique once more when they switched to a representation based on control flow graphs. This was due to efficiency reasons and because it turned out that the implementation of the CFG-based algorithm is easier. More details on this algorithm follow in Section 4.3.1 as our own approach is based on it. In comparison with TestTube, Rothermel and Harrold additionally argue that their technique is more precise. Rothermel and Harrold's techniques presented so far focus on the procedural languages C. As C++ is also very popular, they have extended their approach for this language [230, 236]. Finally, Harrold et al. [126] have addressed the special demands of Java. We use some of their adaptations and explain them in Section 4.3.1.

Ball [25] builds his work upon the RTS technique presented by Rothermel and Harrold [233] and exploits that on one condition, their algorithm selects tests for re-execution although the new program version P' does not behave differently from the old program version P. This special condition requires that a node is visited multiple times. It has already been investigated by Rothermel and Harrold before. They call this the *"multiply-visited-node condition"* [233, page 186]. Thereupon, Ball proposes and defines the intersection graph that is generated from the CFGs of P and P'. With this new kind of graph, he defines three novel algorithms that overcome the detected weakness of the algorithm presented by Rothermel and Harrold [233]. However, these algorithms imply greater computation costs. Moreover, Rothermel and Harrold report that a multiply-visited-node did never occur in their evaluation. As stated by Ball himself, it is unclear whether the multiply-visited-node condition could occur in practice when assuming a language like C, C++, or Java. To the best of our knowledge, he never published a follow-up paper that would show such a constellation. For these reasons, we do not employ one of the novel algorithms to create our CFG.

Another work that resembles the approach of Rothermel and Harrold has been presented by Schumm [248]. In a first step, he measures the test coverage with CodeCover [44]. It supports statement, branch-, loop-, and modified condition/decision coverage. Afterwards, the author compares P and P' using a special abstract syntax tree which he calls the "More Abstract Syntax Tree"[1] (MAST) [248, page 42]. This way, he can determine changes made in P' and finally, he can select test cases that have to be re-executed. In contrast to our approach, Schumm uses coverage data (e.g. branch coverage) for determining test cases that are affected by code changes. As we will see later (see Section 4.4.3), this solution is not applicable in transcompiled cross-platform applications. Besides, the MAST is less precise than our approach as it only represents types, methods, and constructors. Fields or expressions such as conditional expressions remain unconsidered. Likewise, some kinds of blocks such as `try`-blocks or as `finally`-blocks cannot be handled by their AST. Changes in these blocks are interpreted as changes in the enclosing blocks. Naturally, this will usually result in a larger number of tests selected for re-execution. The test selection is more imprecise than it could be.

Although we follow Apiwattanapong et al. [12] in their argumentation that UNIX Diff-based tools are unsuitable for RTS in the environment of object-oriented source programming languages (see Section 2.2), we want to mention for the sake of completeness the work of Vokolos and Frankl [280] who have managed to create a safe RTS tool called Pythia for programs written in C by combining the Diff tool with other scripts which utilize the C compiler and other helper programs written for the C programming language. This way, Pythia creates a a special form of the source code without elements hindering a textual comparison such as comments or blank lines. The results of a study

[1] The author just tells that the syntax tree represents packages, types, methods, and constructors. A more detailed explanation why it is called "more abstract" is not available.

presented by Frankl et al. [87] show that both the selected number of tests and the costs for the analysis are similar to the graph walk-based approach proposed by Rothermel and Harrold [233]. Nevertheless, we want to emphasize that PYTHIA is written for the procedural language C rather than for object-oriented languages. So the argumentation of Apiwattanapong et al. [12] still holds (see Section 2.2).

In a recent publication, Öqvist et al. [208] present a safe, extraction-based regression test selection that relies on a dependency graph to find code modifications and to select test cases for re-execution. In contrast to other approaches, the technique works without code instrumentation. Furthermore, the code analysis is extremely coarse-grained as it just considers files rather than classes, methods, or even statements. Besides, the authors update their dependency graph incrementally after code modifications. As a result, the overhead is rather small. However, they admit that their approach is less precise than other approaches that analyze code more fine-grained, for example at method level. For our purposes, coping without instrumentation makes their approach unsuitable. In transcompiled applications, we need to link code changes in the source language with code in the target language in order to determine UI/web tests that have to be re-executed. To this end, instrumentation is highly beneficial. More details follow in the next sections.

Web Application Regression Testing: For web applications, there are also some studies on regression test selection, but due to the fact that web applications are much younger than traditional desktop applications, there are much fewer techniques available than for desktop applications. In general, research has not payed much attention to selective regression testing of web applications so far. This also becomes evident in the review of existing techniques for web application testing of Doğan et al. [60]. To the best of our knowledge, especially the problem of regression testing *transcompiled* applications has not been considered as a whole yet. Instead, efforts have been made to ease problems in debugging that result from e.g. the dynamic typing of JavaScript. Here, for example FIREBUG [198] is an established tool to step through the JavaScript code and to investigate the actual state of the application. JSFIDDLE [159] can be used to check and run small JavaScript components.

The first papers (e.g. [76, 225, 288]) that investigated techniques for analyzing and testing web applications focused mainly on the content of web pages and the navigation from one page to another. They could not be used for regression testing. Apart from this, today's standard technologies like AJAX-calls [90] that change the state of a web application without leaving the currently displayed page have not been invented yet at that time.

Ricca and Tonella [225] identify single web pages as the main entities of a web application. These pages are connected via hyperlinks and may consist of forms and frames. In order to represent these basic elements of a web application under test, Ricca and Tonella introduce a UML model, where nodes correspond to web pages, forms, or frames. Hyperlinks correspond to edges.

The model enables the authors to analyze site structures as well as to do a semi-automatic test generation that exploits the UML model to create sequences of URLs (i.e. test cases).

Elbaum et al. [76] have compared Ricca and Tonella's technique [225] to create test cases with several new techniques that incorporate user session data. With these new techniques, they have been able to reduce the effort to create tests and to provide adequate input data for forms and client requests. In addition, they have investigated the ability of the tests to reveal faults in web applications.

An important step towards regression testing has been made by Alshahwan and Harman [9]. They propose an approach to repair user sessions that have become invalid because of changes. Therefore, session data (sequences of URLs) are tracked from the original program version. As soon as the web application changes, a repair algorithm tries to fix the session. In contrast to the approach presented by Ricca and Tonella [225] (see above), the repaired session data can be used for further regression testing, whereas non-repaired session data might have become useless because the sequences are not valid any longer. Our approach is partially orthogonal to this approach proposed by Ricca and Tonella. We try to identify the tests that could reveal bugs or that have to be changed due to modified use cases and we try to associate these tests with the underlying Java code. Our approach could be connected with (an enhancement of) their idea in order to restore a valid use case without the need of re-recording it manually.

Xu et al. [288] also consider web applications as collection of web pages which is traversed by users. But they have presented a technique that already addresses the necessity to retest web applications. Their technique is based on slicing (see Section A.2 in the Appendix) and tries to analyze which pages or which elements (e.g. links) in a page of a web application have been added, modified, or removed. To this end, the authors consider out-going links that point to another page and in-going links that point to the currently considered page. Besides, they distinguish between "direct-dependent" and "indirect-dependent" relationships [288, page 654]. Direct-dependent relationships represent structural changes in the web site such as added, modified, or removed hyperlinks. Indirect-dependent relationships are obtained via definition-use relationships of variables within a web page. Their technique does not require a complete system dependence graph. Instead, they just consider each web page individually and determine modifications and their corresponding effects. Based on these analyses, they are able to decide about necessary tests. Although Xu et al. pursue some targets that are similar to ours, their approach does not fit our demands. Apart from the fact that the slicing technique is not safe according to Rothermel and Harrold [232], the technique of Xu et al. [288] is restricted to standard changes like adding/ removing a page element or an entire web page. We want to remark that dynamic features like JavaScript events are never mentioned in the paper. AJAX-calls did not exist yet and consequently, the paper has not addressed asynchronous web applications.

Soechting et al. [260] propose a method to do syntactic regression testing. With the aid of special comparators, they are able to cope with tree structures such as those present in abstract syntax trees or HTML/XML files. Their tool, SMART, implements a distance metric, indicating which tests should be re-executed during regression testing. However, we want to remind that in object-oriented programming languages like Java, this is not sufficient due to polymorphism (see also our explanations in Section 2.2, Paragraph "Discussing the Approaches").

One of the few regression test selection techniques for web applications has been proposed by Tarhini et al. [272]. They propose event dependency graphs to represent an old and a new version of a standard web application, consisting of pages and dependencies (visible/invisible effects and links). They determine changed as well as possibly affected elements and select tests that are touched by the changes. The basic idea to identify changed, added, and removed nodes is similar to our approach. They also search in the graph representing the old and the new version for a corresponding node. However, their technique still focuses on web pages and corresponding HTML elements like buttons rather than programming languages that are typically used in transcompilers (e.g. Java or C#). They also do not care about the granularity of the elements and related performance issues. The elements themselves as well as the dependencies are managed in SQL tables. There is no information provided about how to create the event dependency graph and the SQL tables. Notwithstanding the year of publication (2008), they still do not consider the usage of AJAX in detail which plays a vital role in modern web applications.

Several more recent contributions that build upon crawling AJAX code have led to the ability to localize faults in JavaScript and to create unit test cases which in turn can be used to do regression testing. Mesbah and colleagues [186–188] have introduced a tool called CRAWLJAX. This consists of a crawler which is able to automatically analyze *client side code* of AJAX-based web applications with respect to user interface states and how a single state could be altered via user interactions. To this end, a CRAWLJAX extension called ATUSA [186] searches for clickable elements as well as data entry points like forms. For forms, it tries to find suitable default values or assigns, if available, custom values from the Document Object Model (DOM). Besides, it generates a state machine that maintains a state-flow graph. The state-flow graph represents all possible states of the user interface and its state transitions. On this basis, constraints (invariants) and corresponding assertions can be defined for the user interface which allow to detect faults. Finally, starting from the state-flow graph, paths through the web application are calculated which in turn are transformed into JUNIT test cases. Roest et al. [226] apply this approach to perform regression testing of AJAX-based web applications. They especially focus the problems that are associated with the non-determinism of AJAX.

When creating JUNIT test cases for a web application, the technique implemented in CRAWLJAX and ATUSA is powerful. It is orthogonal to our technique with respect to transcompiled web applications. Here, a well established and

widely used method to create web tests is to use capture and replay tools like Selenium. However, creating web tests this way requires a high manual effort. Besides, it might happen that tests do not cover all relevant states and user inputs. The ability of Atusa to automatically generate tests is independent of transcompilation and fault localization in the original source code. So it could enhance the collection of web tests with additional tests. Nonetheless, this approach only works with JavaScript code. As a consequence, it does not help in transcompiled applications to perform regression test selection and fault localization.

Another technique for regression testing JavaScript code has been contributed by Mirshokraie and Mesbah [193] and is also based on invariants. At the time of publishing in 2012, the authors have stated that it was one of the sole regression testing techniques in the scope of JavaScript based web applications. They instrument a web application on-the-fly, run it many times with different variables, and log all the execution traces. Thereof, they try to automatically derive invariants as runtime assertions and inject them in the next program versions. Failed assertions are reported in order to draw inferences and to get information about the code location. The approach of Mirshokraie and Mesbah is implemented in a tool called Jsart and is a plug-in to Crawljax. It is a dynamic analysis technique, whereas our analysis is – apart from the initial test execution – static. That is, Mirshokraie and Mesbah have to re-execute the entire web application to ascertain that all the assertions hold. Just as the approach of Roest et al. [226], a selective test selection comparable to ours is not available. Beyond that, there is no need in our approach to re-execute the modified version P' of the application to reveal broken assertions or to detect changes that may cause failures, respectively.

Based on Jsart, Mirshokraie et al. additionally focus in a more recent work [194] on creating tests for JavaScript functions and events automatically. Besides, they propose an algorithm that generates mutants in order to check the behavior of the web application under test for regression faults.

Ocariza et al. [206] have presented in 2015 AutoFlox, another tool that is based on Crawljax. It does not implement a regression testing technique, but provides a dynamic analysis that automatically localizes JavaScript faults arising during DOM manipulations by means of JavaScript. The authors start in their approach with an on-the-fly code instrumentation of statements which enables them to collect execution traces. Afterwards, they perform dynamic backward slicing (see Section A.2 in the Appendix, in particular Paragraph “Backward Static Slices and Forward Static Slices” and Paragraph “Dynamic Slicing”) in order to find the fault that leads to an exception.

Kumar and Goel [170] rely on the comparison of two event-dependency graphs representing the old and the new version of a web application. Based on these results, the authors are able to identify changes in the new version and to select test cases.

Especially the crawler-based approaches are powerful and promising, but we want to emphasize that they only work as long as the application is directly

written in the target language (here: JavaScript). If the application's target code has been created by a transcompiler, it is difficult to understand due to which code changes in the original source code a test case has failed. This is especially true if the target code has been optimized and obfuscated. (Please remind that for example the GWT compiler creates several permutations of obfuscated and highly optimized code.) So the approach is only helpful if the code of the target language has been written by a developer rather than by a transcompiler. For localizing faults in the source programming language, the mentioned techniques are not applicable.

Due to the popularity of Java, more technologies like JavaServer Pages [212] or JavaServer Faces [209] have been invented that allow to avoid writing JavaScript code (at least to a certain degree). Here, client-side code consists of standard HTML code combined with a special tag library that provides components for an easy creation of user interfaces. Server-side code is completely implemented in Java. As a consequence, there is no need for transcompiling code because server-side code is compiled into bytecode as usual. So, the concepts proposed for regression testing standard desktop applications are still and directly usable.

Based on this insight, it is enough for Asadullah et al. [16] to focus on the server side when considering web applications that employ Java-based frameworks like JavaServer Faces or Spring [218]. The procedure described by the authors in their RTS technique meets special demands like changes in the runtime behavior due to Spring's use of the Factory Pattern. So the main challenges of Asadullah et al. are different from ours. They neither have to deal with transcompilation and the problem to instrument Java/ JavaScript source code, nor do they have to deal with web tests and the problem of reducing the execution time of this special kind of tests. In addition, their approach is less precise as it only handles methods, but no statements (not to mention conditional expressions).

In an elder publication, Huang et al. [143] also exploits the possibility to work directly with bytecode. Again, the technique only handles changes at Java method level. When it comes to modifications in JavaServer Pages or JavaScript code, they apply a syntax analysis to detect them. Then, the technique creates groups of tests. Each of them covers all code modifications. The final test selection is based on a metric which prioritizes the groups according to the risk of the corresponding tests to fail.

Code Analysis and Fault Localization: As already mentioned in Section 2.6, many different fault localization techniques have been developed in the past. In the following paragraphs, however, we mainly focus on code analysis and fault localization techniques that are similar to our own approach. It seems to fit the model-based technique according to the categorization of fault localization techniques introduced by Wong et al. [285] (see also Section 2.6). For a full overview of the available techniques and relevant papers, we refer the reader to the review of Wong et al. [285].

In order to analyze Java code of two program versions for differences, Apiwattanapong et al. [12] extend an existing approach proposed by Laski and Szermer [173]. In an initial analysis, Apiwattanapong et al. compare two program versions in several iterations, starting on class and interface level. They try to find for each class/interface in the old program version P a corresponding class/interface in the new version P'. Based on these data, they do the same with methods. Classes/methods without counterpart are considered as added/deleted. Afterwards, they create special CFGs for all methods that exist in P and P' in order to find corresponding statements. Apiwattanapong et al. call these graphs *Enhanced CFG* (*ECFG*) [12, page 6]. All the graphs are simplified to so-called *hammocks* [12, page 12]. Hammocks are a kind of substitute nodes that might represent several nodes in the ECFG or even other hammock nodes. So in the end, a method can be represented by a single hammock node. In order to determine changes in the code, corresponding hammocks in P and P' are compared to each other. To this end, the hammocks are recursively expanded again. Apiwattanapong et al. associate all corresponding nodes with either the attribute "unchanged" or "modified" [12, page 14]. The analysis continues then with corresponding successor nodes. In order to find such a correspondence in the successor nodes, they compare the edges starting in the current nodes. Depending on the proportion of matched nodes to the total number of nodes, Apiwattanapong et al. decide whether a hammock is considered as matched. This decision depends on a similarity threshold defined by the user. The higher the threshold, the more nodes must be be labeled "unchanged" in order to consider a hammock as matching. The whole algorithm is *intraprocedural* [12, page 8]. That is, it just compares the nodes within a method. As long as classes/interfaces or methods are not renamed, Apiwattanapong et al. are able to recognize corresponding methods (or classes/interfaces) in P and P' automatically. However, in case of renamed methods or modified method signatures in P', the user needs to define manually which elements belong together as the analysis is intraprocedural. Although Apiwattanapong et al. argue that this "could be (partially) automated using similarity metrics" [12, page 8], it seems that this approach lacks the possibility to be applicable in a fully automated environment. In our technique, we adapt an approach of Harrold et al. [126] that uses an *interprocedural* analysis (e.g. [233, page 189]). We discuss this in Section 4.3.2 in more detail. It does not require the user to do manual tasks. Renamed classes and methods will be recognized directly as soon as there is a call. Dead code may be optionally included in our analysis on demand. Apiwattanapong et al. have implemented their algorithm in a tool called JDiff. We want to point out that it is tailored to work with non-transcompiled desktop applications only. When applied on a transcompiled application, it could at best analyze the transcompiled target code. But in contrast to our approach, it cannot localize changes/faults in the source code of the transcompiled applications.

Another approach to identify changes has been proposed by Nguyen et al. [204]. They model classes and methods as nodes. Interactions between these elements are modeled as edges. The authors call this an *interaction-centric*

approach [204, page 572] and they state further that their approach enables them to find all kinds of changes in classes and methods, including renaming, reordering, modifying, adding, and removing of elements. However, due to their interaction concept, they are not able to analyze code changes on a more fine-grained level such as statements.

Raghavan et al. [220] expand in their tool DEX a normal abstract syntax tree by semantic information, which is useful to "connect literals and declarations to their types and variable references to their variable declarations" [220, page 189] in programs written in C. They call this expanded AST the *Abstract Semantic Graph* (*ASG*) [220, page 189]. According to the authors, the original intent of DEX has been to analyze bug fixes. Nevertheless, they explicitly emphasize the usefulness of their tool in terms of regression testing and in the analysis of code changes.

Hoffman et al. [138] report on a semantics-aware analysis. They apply a differencing algorithm on two program traces obtained by running the old and the new version of a program. Each trace consists of a series of quintuplets encapsulating (among others) data about the current object, thread, and method. The algorithm generates *semantic views* [138, page 453] consisting of specifically processed information about incidents in the traces. This way, the authors are able to isolate a number of changes that might be the reason why tests have failed. In contrast, we select tests and get information about code that might be responsible for a test failure in a static way without the need to run the new program version in order to obtain a second trace.

Baah et al. [24] have developed the *Probabilistic Program Dependence Graph* (*PPDG*) [24, page 189] as a way to both localize and understand faults. As the name already suggests, the graph builds on a common program dependence graph, but adds sometimes some nodes and edges in the graph to meet special requirements. Each of the nodes in the PPDG is associated with an abstract state unifying all the possible states that can be adopted by a statement. In addition, the authors calculate the conditional probability for each node to be related with their parent nodes in the graph. According to Baah et al., "conditional probabilities measure how nodes are influenced by their parent nodes in the PPDG." [24, page 194] Based on these data, the authors are able to rank nodes according to the probability to be responsible for a fault.

Hao et al. [125] have presented a technique that accompanies the developer during the fault localization process until the reason for a test failure is detected. Their technique is based – just as many other techniques – on the information which tests execute which statements and which tests pass/fail. From this data, the technique selects the statement that is the most likely the cause for a test failure and proposes a breakpoint for the debugging process. Afterwards, the developer has to give feedback whether or not the statement was really the cause for the test failure. If not, the technique recalculates another statement that should be checked. According to the authors, the advantage is that the developer does not have to decide from a set of possible code changes which one could be the cause for the test failure. Instead, their technique indicates

a statement to start with. This approach seems to be very interesting and is orthogonal to our own approach. We could imagine to integrate their idea in our technique. We could arrange the code changes identified during our analysis according to the probability to be responsible for a test failure.

Although we do not deal with change impact analysis, this technique can be used to localize faults in the source code as explained in Section 2.6. CHIANTI is such a change impact analysis tool and has been introduced by Ren and his colleagues [223, 224]. It builds on previous work proposed by Ryder and Tip [242], two of the contributors of CHIANTI. The tool determines tests whose behavior in the current program version might differ from the one in a previous program version. To this end, the authors extract atomic changes and investigate them for syntactic dependences. A comparison of the two program versions by means of a CFG is not necessary. Instead, CHIANTI applies call graphs (see Section A.3 in the Appendix, Paragraph "Call Graph"). In the first prototype, Ren et al. [223] use static call graphs. In a subsequent version [224], Ren et al. investigate dynamic call graphs. On this basis, they are able to depict which modifications might be responsible for a test failure. In addition, atomic changes can be used to create so-called "intermediate program versions" [224, page 433] that contain only some of the modifications made in the current version [224]. According to Ren et al. [224], the test selection is safe. In contrast to our work, CHIANTI does not compare code in parallel and analyses the code on method level, not on statement level or below as we do. Besides, we do not require the new program version P' to be completely "syntactically correct and compilable" [223, page 2], which is one of the preconditions CHIANTI imposes. Our analysis of P' is completely static and can already be started while some parts of P' are not finished yet. As a result of such an early analysis, we obtain important information about changes or newly introduced features that might impair correct functionality. This information is helpful for example in code reviews. If the application consists of (compilable) subprojects, we can even select already existing tests for re-execution.

Chesley et al. [39] apply CHIANTI in an Eclipse plug-in called CRISP. Beginning with two program versions and a failed regression test case, it produces a compilable intermediate program version that is based on the previous version plus some of the changes introduced in the current version. The user decides which changes are contained in the intermediate version. This way, it is possible to create versions that will not fail when rerunning the test. Creating several versions by using CRISP gives some indication of faulty changes.

Stoerzer et al. [266] have created JUNIT/CIA, another Eclipse plug-in that uses CHIANTI. As the name already implies, the tool is JUNIT-based and is able to estimate which changes could cause a test failure. Based on this estimation, it categorizes these changes with different colors according to the probability that they really give rise to a test failure.

Zhang et al. [293] propose FAULTTRACER, an approach that couples change impact analysis with spectrum-based localization of faults. Compared to CHIANTI, it also determines atomic changes, but takes fields and overridden meth-

ods in a more precise way into account. Besides, it relies on already existing spectrum-based fault localization techniques whose basic idea is to order statements according to their risk to be faulty. For example, Zhang et al. apply a statistical analysis of test execution traces that has been proposed in a paper by Jones et al. [157]. Their analysis assumes that the probability of a statement to be faulty rises with the number of failing tests covering this statement. As opposed to Zhang et al., we do not use statistics to localize faults. Instead, we use the analysis of edges as decider where a change is located in the code. Moreover, similar to Hoffman et al. [138], Zhang et al. have to re-execute the new program version in order to apply their technique. As a consequence, this does not reduce the test effort as it is not a test selection. We do not necessarily need to re-execute all the tests.

Although our approach to localize faults seems to fit the model-based technique [285] (see also Section 2.6), we take a peek at spectrum-based techniques. There are many publications in the area of fault localization techniques that rank suspicious parts of the source code via special metrics according to their likelihood to be the cause of a test failure (see the review of Wong et al. [285] for an overview). Abreu et al. [2] consider several similarity coefficients used in spectrum-based fault localization techniques to determine such a ranking. They also show that their own similarity coefficient proposed in an earlier paper is superior to the other coefficients. In addition, they report on the ability of different parameters and their effect to recognize correctly in which part of the source code the cause for a test failure is located. Our own approach considers program changes. Thereof, it determines a set of possible fault locations without any ranking of faults according to their suspiciousness. However, our approach could be expanded by such a similarity coefficient to find faults faster. Execution logs obtained via code instrumentation – a major prerequisite of spectrum-based fault localization – are also available in our approach. So this means no extra overhead.

Dallmeier et al. [51] have proposed another idea to detect and localize defects. They consider sequences of method calls and state that this approach is cheap as well as that it has a higher chance to locate defects than common coverage techniques.

Finally, Yoo et al. [291] have formulated a variant of the fault localization problem, the *fault localization prioritization problem* [291, page 19:1]. As the name already indicates, it is a combination of fault localization and prioritization. So, the main concern is:

> "Having found a fault, what is the best order in which to rank the remaining test cases to maximize early localization of the fault?" [291, page 19:2]

The authors address this problem in their new technique called FLINT, which is an acronym for "Fault Localization using INformation Theory" [291, page 19:2]. In Chapter 6, we ourselves introduce an own prioritization technique. But our intention is to order test cases that should be re-executed due to code modifications.

We want to emphasize that research on topics like cross-language program analysis (e.g. [183, 219, 267]) is – despite the similar name – *not directly related to our work* as this kind of research focuses on dependencies, references, and links among different languages within the same application. In contrast, we investigate kinds of applications that are written in a single source programming language. This kind of application is transcompiled in one or several distinct target applications. The source programming language and the target programming language do not share any dependencies, references, or links.

Code Instrumentation: As already mentioned in Section 2.3, RTS techniques require knowledge about which test cases execute which code elements/ which parts of the code. An old but still very popular way to obtain this information is to use program tracing. For example, in a paper published in 1993 [171], Larus discusses efficient techniques that can be used to trace systems. In particular, he explains control-flow tracing which we outline briefly. For the full details, we refer the reader to his paper [171]. Larus takes a *control-flow trace* [171, page 55] as a recording of instructions that have been traversed during program execution. However, he explicitly points out that due to efficiency, the entries in the control-flow trace normally represent whole basic-blocks instead of single instructions. When it comes to the details how to record such a control-flow trace, Larus mentions several ways, including the possibility "to assign each block a unique identifier and to append the identifier to the trace record every time the block executes" [171, page 55]. According to him, this procedure has as major drawback that "the tracing code greatly increases both a programs size and execution cost" [171, page 55]. For this reason, he suggests to consider solely the transition from one block to another. Using these control-flow traces and a special "trace recognition routine" [171, page 55], he is able to reproduce the program trace. Despite the author's objections, we use the first-mentioned procedure as the basis for our own approach (see Section 4.4.3).

In other work (e.g. [26]), code instrumentation is used to assign an integer value to every edge in such a way that the accumulation of these values is unique. In more detail, accumulating the integer values while traversing a specific path in the program results in a final score that differs from the final scores of all other paths. This approach is the starting point for further slight variations (e.g. [172]). However, the intention of these approaches differ from our purposes. For example, Larus [172] wants to find paths that are relevant for performance improvements or compiler optimizations. Ball and Larus [26] additionally mention software test coverage as possible purpose. For our purposes, an accumulation of values is unsuitable as we need to know exactly which parts of the code are traversed in the target programming language and to which parts in the source programming language they correspond. This is important to locate potential faults in the source programming language.

As already noted by Chawla and Orso [36], a wide-spread approach in Java programs is to add instrumentation code to the bytecode (see also the discussion

in Section 2.3). Their tool, InsECT, implements this approach in a dynamic and generic manner. To this end, it introduces instrumentation tasks to define which parts of the code have to be instrumented (e.g. "method entry"/"method exit", "before method call"/"after method return", "field read"/"field write", "start of basic block"/"end of basic block", "before a branch"/"after a branch", "throw"/"catch" [36, page 2]) and which information has to be gathered. This way, the authors wish to achieve that the tool is easier to reuse and to modify.

Similar approaches have also been used for tracing the execution of a program. For example, Ayers et al. [23] have published a patent in which they use identifiers to instrument blocks. When executing the program, they obtain sequences of block identifier. These sequences enable them to reproduce the different steps that led to a program error.

In Section 2.3, we have briefly discussed several ways to perform source code instrumentation. Geimer et al. [92] have presented a prototype that implements this approach. It pursues a similar target as InsECT that we have just summarized before. The intention is to configure the instrumentation code flexibly in order to support different programming languages. At the time of publication, however, the authors were only able to provide a generic instrumentation for function entry and exit. The prototype implementation is based on a performance system called TAU that has been introduced by Shende and Malony [257]. It supports investigating "performance technology problems" via "instrumentation, measurement, and analysis" [257, page 309]. Analyzing the performance can be done in several ways, "including powerful selective and multi-level instrumentation, profile and trace measurement modalities, interactive performance analysis [...], and performance data management" [257, page 309]. TAU adds instrumentation code in the source code of a multitude of programming languages, including Java, C/C++, Python, and Fortran via a preprocessor. In addition, TAU also supports, among others, compiler-based instrumentation, binary instrumentation, component-based instrumentation, and virtual machine-based instrumentation. However, the instrumentation is designed for specific performance events such as "events defined by code location (e.g. routines or blocks), library interface events, system events, and arbitrary user-defined events" [257, page 289].

Technically, the instrumentation approaches discussed before are not enough for our purposes as we need more detailed information about the execution of single statements or even expressions rather than entry/exit events in case of Geimer et al. [92] or whole blocks in case of Larus [171] or Ayers et al. [23]. Notwithstanding this, as we deal with different potential transcompilers that support several source programming languages, the ideas pursued by Chawla and Orso as well as Geimer et al. to offer a generic, reusable instrumentation technique are basically very interesting. Nonetheless, we focus in the first place on the basics of our instrumentation and one possible realization.

Due to our focus on transcompiled applications and the need to localize faults in the source programming language, source code instrumentation is relevant. For this reason, existent tools such as Pin [146] that insert instrumen-

tation code in the executable [147] rather than in source code are not suitable. Our instrumentation differs from the usual one as our target is not to determine whether a Java code element is covered by a branch of the CFG. In case of GWT for example, web tests only execute JavaScript code. So client-side Java code is never executed directly. Our instrumentation is highly specialized in such a way that the GWT compiler is able to maintain the binding to the underlying source code when transferring Java code into JavaScript. This is very important when mapping Java code modifications to the target code executed by a web test.

Moreover, we want to mention the interesting findings of Li et al. [179]. They have checked with regard to branch coverage whether the results of a bytecode instrumentation differ from those obtained in a source code instrumentation. First of all, they have noticed that there exist only three tools under active development that offer branch coverage for methods. One of them (EclEmma) does bytecode instrumentation, the other two perform source code instrumentation. But most interestingly, they have found that – at least for the tool (EclEmma) used to investigate branch coverage – "Bytecode instrumentation is not a valid technique to measure branch coverage" [179, page 387].

More related work (e.g. [40, 276]) dealing explicitly with instrumentation in terms of code coverage follows in Section 7.2.

4.3 Regression Test Selection for Transcompiled Cross-Platform Applications

Having knowledge about code modifications and their potential to change the execution of a test is the prerequisite for a safe test selection. So it is crucial to determine which parts in P' have been changed compared to P. Based on this insight, we have to find out which tests execute these parts of the code. More generally, we need to know for the entire application which pieces of code are traversed by which test case. In the literature, there exist already approaches that address this problem in a specific context. We have a closer look at the background, now. It is important in order to understand how our own approach differs.

4.3.1 Background

In the area of safe graph walk-based RTS techniques, one of the most influential approaches has been presented by Rothermel and Harrold [233]. They propose two algorithms that compare the code of a modified program version P' with the code of the original program version P in order to determine all the tests of a given test suite T that execute modified code. Rothermel and Harrold distinguish *intraprocedural* and *interprocedural* test selection [233, pages 179f. and 189f.]. While the first algorithm considers functions in isolation, the latter one additionally embraces calls between functions. Consequently, it can be

used to analyze the entire code with its interactions and to find tests that are affected by changed interactions. In the following subsection, we summarize directly the interprocedural algorithm presented by Rothermel and Harrold [233] as it is based on the intraprocedural one. For full details, we refer the reader to the original paper of Rothermel and Harrold [233].

Basic Algorithm

We have already sketched in Section 4.2 that the basic algorithm of Rothermel and Harrold is the result of several previous versions that have been based on control dependence graphs, on program dependence graphs, and on system dependence graphs. Due to efficiency reasons and because of implementation simplicity reasons, the basic idea of the current algorithm [233] is to model both program versions P and P' as control flow graphs. While in our approach, a node in a CFG may represent any kind of code element (even expressions, see Section 2.4), the nodes in the CFGs of Rothermel and Harrold are restricted to represent statements exclusively. As usual, edges represent in their approach the control paths which may be passed through during program execution, depending on the values the statements are evaluated to. To distinguish the nodes, they always use the textual representation of the corresponding statement as label. Some specific kinds of edges have a label, too. In case distinctions for example, labels are used to represent the different cases. In order to determine whether a test case is modification-traversing (see Section 2.4) or not, the algorithms proposed by Rothermel and Harrold start from entry nodes that occur both in P and in P'. They traverse the two CFGs of P and P' in parallel and look for equally labeled edges that have coincident successor nodes. As soon as no matching edges can be found, all the tests executing these edges are selected for retesting.

Technically, the interprocedural test selection summarized above starts by determining the entry procedure of P. In the programming language C for example, this is the `main()`-function. For each procedure $p \in P$ that is about to be analyzed, a corresponding procedure $p' \in P'$ is looked for. The name of p is minuted in a data structure called *proctable* and p is marked as "visited" [233, page 191]. Afterwards, Rothermel and Harrold construct a CFG for both p and p' with entry nodes i and i', respectively. Starting from these entry nodes, pairs of nodes $(n_p, n_{p'})$ are considered, where $n_p \in p$ and $n_{p'} \in p'$. To avoid multiple analysis of the same node pairs, n_p is marked as "$n_{p'}$-visited" [233, page 190]. Now, for each successor node s_p of n_p, another node $s_{p'}$ will be looked for that follows $n_{p'}$, fulfilling the one condition that the edges $e_p = (n_p, s_p)$ and $e'_p = (n_{p'}, s_{p'})$ have the same label. Provided that the node s_p is not marked as "visited", the labels of s_p and s'_p are tested by a simple lexicographical comparison for coincidence. If there is no coincidence, all the tests executing e_p are added to the set of tests for re-execution. If the node s_p represents a call to another procedure, the algorithm depicted just now is repeated. That is, for the called procedure q in P, a counterpart q' is searched for in P' and the CFGs for $q \in P$ and for $q' \in P'$ are traversed in parallel. In case that these

procedures are already marked as "visited" or "selectsall" [233, page 191], no analysis is necessary and the algorithm continues with the calling procedures. A procedure might be marked as "selectsall" if the exit node cannot be reached. Then, all these tests that execute nodes of the corresponding procedure are added to the set of tests for re-execution.

The information which edges are executed by which test can be obtained from the execution traces collected during the test execution of p (see Section 2.4). Rothermel and Harrold store these data in the form of a "bit vector" [233, page 177] whose values describe for each edge in P whether it has been executed by a test case in the original test suite T.

Discussion: The algorithm of Rothermel and Harrold has several important attributes. First of all, the authors are able to prove that their approach is *safe* [232, page 532], [233, page 185] provided that P' is tested with equivalent parameters and settings as P before [233]. The authors call this *controlled regression testing* [233, page 184]. According to Rothermel and Harrold, this is because "the modification-traversing tests are a superset of the fault-revealing tests" [233, page 185]. Furthermore, the analysis is static and does not require the whole execution of P'. So, the algorithm of Rothermel and Harrold summarized above is suitable to serve as first concept to solve the test effort reduction problem we are faced to (see Section 1.2).

Form a technical point of view, it is important to note that the algorithm presented by Rothermel and Harrold [233] does not analyze all nodes. Nodes that can only be reached via modified nodes are omitted [233]. This implies that solely those procedures are analyzed that can be reached without a precedent code change. Areas in the CFG requiring a call from a procedure that is already marked as "selectsall" [233, page 191] (i.e. all tests are modification-traversing) are not investigated. Even if there are more code changes in the ignored code parts, all affected tests are already selected due to precedent code changes. On the one hand, this approach ensures that the entire program with all its functions will be analyzed while reducing the effort for the traversal and the comparison of nodes. Nonetheless, we want to point out the fact that this is insufficient for fault localization. If there are more faulty code changes in these parts of the code that have been ignored, it is essential to know about all code positions that might be the cause for a test failure. As Rothermel and Harrold stop the comparison as soon as a modification has been detected, subsequent changes remain undetected. For example, let us assume that the change due to which the comparison has been stopped is not the reason for a test failure. Instead, there is a faulty code change later on in another part of the code. But as this change is still undetected, we cannot incorporate it during fault localization.

In the context of transcompiled cross-platform applications, the approach of Rothermel and Harrold [233] cannot be directly borrowed as it focuses on procedural languages only. More generally, it is not directly applicable to object-oriented languages as concepts like polymorphism or dynamic binding

cannot be handled yet. This has already been ascertained by Harrold et al. [126]. To overcome these limitations, they have extended the idea of Rothermel and Harrold [233, 236] towards a highly specialized CFG.

Specialized CFGs for Java Software

Dealing with Java language constructs like inheritance, polymorphism and dynamic binding, as well as exceptional handling needs adaptations in the CFG. In order to address these special demands, Harrold et al. [126] propose the *Java Interclass Graph* (*JIG*) [126, page 316]. We summarize this representation in this subsection as it serves as starting point for our own approach to handle UI/web tests in transcompiled cross-platform applications. For full details, we refer the reader to the original paper of Harrold et al. [126].

The first special characteristic of the JIG affects the representation of types. Harrold et al. also use CFGs and traverse them in parallel to search for edges that do not match, just as proposed by Rothermel and Harrold in the original approach. Harrold et al. call these edges *dangerous edges* [126, page 315]. Therein, primitive types do not need a special representation, yet. They are represented by their type information which is added to the name of the variable. Object types in Java, by contrast, are part of a type hierarchy. For this reason, they are represented by *globally-qualified class names* that reflect the entire inheritance chain. In doing so, the location of a change in the type hierarchy can be determined.

The next characteristic focuses inheritance. If a call can reach multiple methods because of polymorphism, each of the possible calls have to be represented by an edge. Harrold et al. distinguish several kinds of edges. Intuitively, edges representing calls are denoted as *call edges* [126, page 317]. In order to determine which methods can be reached, Harrold et al. use the class hierarchy analysis proposed by Dean et al. [52]. Of course, some calls from methods declared by the programmer (so-called *internal methods* [126, page 315]) might point to *external methods* [126, page 315] declared in libraries. A call to these methods is modeled by a call edge as usual and uses the class name of the external method as label. But as the internals of external methods are usually not available, they are just expressed by a method entry node and a method exit node. The entire control flow between these nodes is symbolized by a *path edge* [126, page 317]. Vice versa, it might also be the case that external methods perform callbacks to internal methods. To handle such calls, the JIG is extended by an artificial *External Code Node* (*ECN*) [126, page 318]. An internal method that is reachable from an external node is represented by an edge from the ECN to a class entry node, representing the class that declares the internal method. The class name is used as label of the edge. Another call edge connects the class entry node with the node representing the internal method. The regular control flow within internal methods is straightforward represented by normal *CFG edges* [126, page 314]. Calls to other internal methods are modeled by call edges, which are labeled with the method name that declares the invoked method. Again, the control flow within the called method

is represented by CFG edges. Finally, there is another special call edge called *default call edge* [126, page 318]. Harrold et al. use *default nodes* [126, page 318] to model methods that are defined in an external class. Each class having access to external methods is connected with a default node via a default call edge. Harrold et al. need this in order to handle newly introduced/removed internal methods correctly that override external methods. The default call edge is labeled with a "*" [126, page 318].

As we can see, there exist several kinds of starting points for the algorithm that investigates P and P' for code changes. Harrold et al. [126] introduce three different ones: Apart from the `main`-method(s) as standard starting point(s) in the Java programming language, there might also be nodes typifying static methods as well as the nodes that are connected to the ECN mentioned before. Each of these entry nodes serves as initial input for the analysis.

Finally, the JIG needs to handle exceptions. For this purpose, Harrold et al. [126] explicitly model each `try`, `catch`, and `finally` block as node. Path edges connect try nodes with nodes representing catch blocks. The exception is used as label for the path edge. A CFG edge (labeled "caught" [126, page 319]) connects the catch node with the corresponding statements declared in the catch block. CFG edges to the finally node are unlabeled.

Assumptions in the Approach

Harrold et al. [126] emphasize some conditions that have to be fulfilled when applying their technique:

First of all, all tests have to be repeatable, that is, a re-execution of tests has to yield the same results as a previous execution if no changes have been made in the code [126]. For standard desktop applications, this condition affects primarily the compiler, the runtime environment as well as possibly the network infrastructure and databases. Regarding concurrent systems, however, multi-threaded processes must be linearized if a test depends on the order of computations. Thus, tests have to be deterministic.

Another condition is that only classes and methods declared in the project are considered and analyzed (internal classes and methods [126]). Code from referenced libraries (external classes or methods) is excluded as it is often written by third parties and thus is usually not available [126].

Finally, it has to be ensured that no reflection is utilized in internal classes [126]. Reflection would complicate the analysis greatly. Furthermore, reflection could make it necessary to inspect even the external code that stems from e.g. imported libraries. However, this would be contradictory to the previous condition.

Deficiencies of Existing RTS Techniques

From the Related Work Section, it is evident that there is a wide range of RTS techniques. However, as we have outlined in this section, the techniques

have been developed for one specific kind of application that relies solely on the source programming language. For example, Rothermel, Harrold and colleagues have focused on RTS techniques for standard desktop applications written in C, C++, or Java (e.g. [126, 228, 230–233, 236]). Other researchers have addressed the special problems that occur when applying regression testing on web applications (e.g. [9, 186–188, 193, 226, 272]). Especially some of the basic ideas proposed by Rothermel, Harrold and colleagues would actually fit our demands. But please remind that transcompiled cross-platform applications distinguish between a source language and one or several target languages (see Section 3.1). In general, the strict separation of desktop and web applications is not necessarily valid any more. In transcompiled cross-platform applications, it is basically possible to deploy an application as desktop, mobile, or as web application. So what we miss is a RTS technique that has all those special characteristics in mind.

We want to explain the consequences in more detail that arise if we would directly apply standard RTS techniques created for either desktop applications or web applications on a transcompiled application created with GWT. Figure 4.1 sketches the overall situation. Missing details in existing RTS techniques like the possibility to flexibly model the whole range of code entities (complete classes, or even details like conditional expressions; see Section 2.4) are left out in the discussion as they are less decisive. Notwithstanding, these details constitute a deficiency.

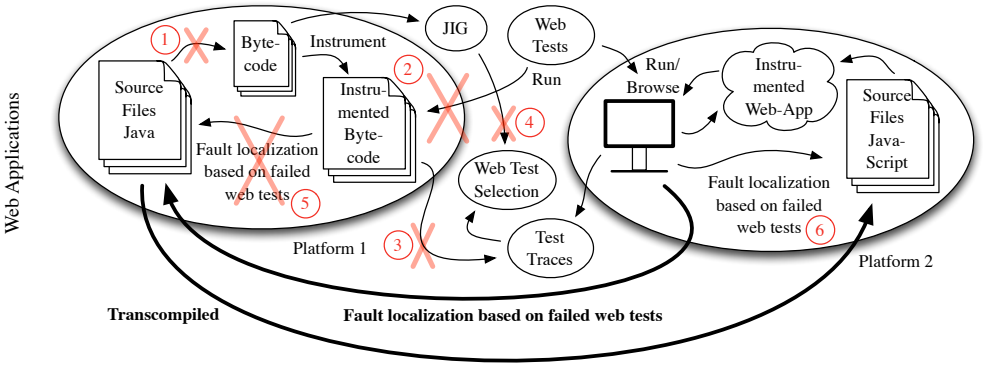


Figure 4.1: Deficiencies in ordinary RTS techniques.

When applying for example the "Regression Test Selection For Java Software" presented by Harrold et al. [126] (see Section 4.3.1, Paragraph "Specialized CFGs for Java Software"), the very first problem affects the entire test selection. For this, Harrold et al. rely on the bytecode that is executed when running test cases. So they compile the Java source files into bytecode as usual. Then, they inspect the resulting Java class files to model the control flow and to build the JIGs for P and P' by applying a special analysis, the *Java Architecture for Bytecode Analysis (JABA)* [13]. In order to obtain the information that describes which tests execute which parts of the application,

```
1  public class StockWatcher
       implements EntryPoint {
2    // ...
3    private Button addStockButton
4      = new Button("+");
5    private ArrayList<String>
6      stocks = new ArrayList<>();
7
8    public void onModuleLoad() {
9      // ...
10   }
11   private void addStock() {
12     // ...
13     stocks.add(symbol);
14   }
15 }
```

```
1  public class StockWatcher
       implements EntryPoint {
2    // ...
3    private Button addStockButton
4      = new Button("Add");
5    private ArrayList<String>
6      stocks = new ArrayList<>();
7
8    public void onModuleLoad() {
9      // ...
10   }
11   private void addStock() {
12     // ...
13     if (stocks.contains(symbol))
14       return;
15     stocks.add(symbol);
16   }
17 }
```

Figure 4.2: StockWatcher: original version (left) and modified version (right).

they instrument the bytecode. Rothermel and Harrold call this *test trace information* [232, page 535]. We simply refer to it as *test traces*. A more precise definition that incorporates essential details of our own approach follows in Paragraph "Technical Demands on Code Identifier" in Section 4.4.3.

As opposed to Harrold et al. [126], we cannot use Java bytecode because the GWT compiler does not compile the source files of client-side code at all (see cross ① in Figure 4.1). The reason is that Java bytecode provides no advantages when optimizing the converted JavaScript code for different browsers [33, 35]. Instead, GWT works directly with `.java` files when generating JavaScript code. So, as soon as the transcompilation process starts, we leave platform 1 (see Figure 4.1) and obtain target code that runs in browsers. This JavaScript code is executed by web tests. They never run bytecode on the client side (see cross ②) and thus, we never get test traces from instrumented bytecode although this is fundamental for selecting (web) tests (see cross ③). We want to emphasize that this is not specific to GWT. In general, UI tests do not necessarily execute the code of the source language (here: Java code). Consequently, adding instrumentation code into the bytecode is completely useless when trying to get information about which web tests execute which parts of JavaScript code of the transcompiled web application. In the end, selecting tests by comparing the JIGs and checking test traces as otherwise customary is not possible (see cross ④). The same is true for the determination of a set of code changes that might be responsible for web test failures (see cross ⑤).

In order to obtain test traces from web tests, one could come up with the idea to just instrument the JavaScript code in order to get adequate web test traces and to adhere to the JIGs created with the aid of JABA. This way,

it would be easily possible to identify code modifications in the Java code. However, this involves the problem that the detected changes are completely unrelated to the web test traces. We want to illustrate this by reusing the small AJAX-based GWT web application StockWatcher (see Section 3.1) as running example. Again, the code (see Figure 4.2) is an excerpt taken from the GWT tutorial [100]. The left half of Figure 4.2 shows the original version. In the code on the right, we have used a different label for the button (see line 4) and we have added an additional check (see line 13-14).

In order to be able to identify web tests that are affected by the two modifications in the Java code in Figure 4.2, the developer would have to hunt up the changes manually in the generated JavaScript code. This is of course an extensive and tedious task especially because the GWT compiler obfuscates the whole JavaScript code when used with standard settings (see Figure 3.1, Figure 3.2, and Figure 3.3 in Section 3.1). So the names of JavaScript functions differ completely from the names of the method declared in the Java code. Solely if the compiler option `PRETTY` is used (see Section 3.1), the GWT compiler generates a more readable JavaScript code shown in Figure 4.3. But even then, it remains difficult because there is many additional code originating from libraries (see lines 12-20). Moreover, we have to remind that GWT combines code originating from different Java classes in a single file per permutation (see Section 3.2) and that this code is highly optimized to deliver a performant web application, but also to handle all established browser types, including possibly existing deficiencies in single browser versions [102]. So it is still hard to extrapolate from changes made in the Java code to the produced JavaScript code and to determine afterwards those web tests that have to be rerun. As we can see, it is quite difficult to bridge the gap between changes identified in the Java code and the web tests that cover generated JavaScript code.

When the application of RTS for standard Java desktop applications is rather cumbersome in the context of transcompiled applications, another possibility could be to use special RTS techniques intended for use in a specific target language. In our running example, this is JavaScript. As already depicted in the Related Work (Section 4.2), there are several approaches available that could be used in pure web applications based on JavaScript. However, none of them do really meet our requirements. Some of them need to re-execute P' (e.g. [138]). And even if we could statically determine a set of web tests that have to rerun, the problem still arises how to refer back from JavaScript to Java in order to localize which code changes in the source language are responsible for a test failure when running web tests in the target language. At best, existing techniques are able to localize changes in the target language (see ⑥ in Figure 4.1), but not in the source language. In general, we stumble over the same problems as we have already explained when trying to reuse RTS techniques for standard Java desktop applications. So again, even if there is no code obfuscation, additional code originating from libraries (see Figure 4.3, lines 12-20) makes it hard to localize the corresponding Java code somewhere in the multitude of Java classes. Dynamic typing additionally complicates the transfer of change information from JavaScript back to Java. As type information is

```
1  function StockWatcher_0(){
2    this.addStockButton = new Button_0('Add');
3    this.stocks = new ArrayList_0;
4  }
5  function $onModuleLoad(this$static){ // ... }
6  function $addStock(this$static){
7    // ...
8    if ($indexOf_2(this$static.stocks, symbol, 0) != -1)
9      return;
10   $add_4(this$static.stocks, symbol);
11 }
12 function AbstractHashMap$EntrySetIterator_0(this$0){
13   var list;
14   list = new ArrayList_0;
15   this$0.nullSlotLive && $add_4(list,
16     new AbstractHashMap$MapEntryNull_0(this$0));
17   $addAllStringEntries(this$0, list);
18   $addAllHashEntries(this$0, list);
19   this.iter = new AbstractList$IteratorImpl_0(list);
20 }
```

Figure 4.3: One of the permutations representing the modified version of StockWatcher in JavaScript.

missing, a precise identification of the Java code responsible for a change is often not possible. In Figure 4.3, line 3 shows an example for type information that has been dropped (compare line 5 in Figure 4.2, missing `String`).

As we can see, existing RTS techniques in general are not directly applicable to GWT web applications in particular and in the special environment of transcompiled cross-platform applications in general. Beyond that, existing techniques often focus on one specific problem (e.g. regression test selection or fault localization). We want to offer a solution for transcompiled cross-platform applications that fits all our purposes (see Section 1.2).

4.3.2 Calculating Changes Made in the Source Code of Transcompiled Cross-Platform Applications and Selecting Tests

In order to provide a RTS technique that is able to handle transcompiled cross-platform applications, our method for calculating changes made in the source language is divided into ten steps (see numbers in Figure 4.4). It builds on the basic idea of Harrold et al. [126] and Rothermel and Harrold [233] (see Section 4.3.1). In particular, we trust in their experience that using a CFG in a graph walk-based approach is more efficient than using a CDG, PDG, or SDG. Consequently, we do not consider these kinds of graphs in our approach.

In the very first step, developers create the initial version of a new application in a certain source language (see ① in Figure 4.4). Afterwards, they instrument and transcompile this original program version P (see ②) and ob-

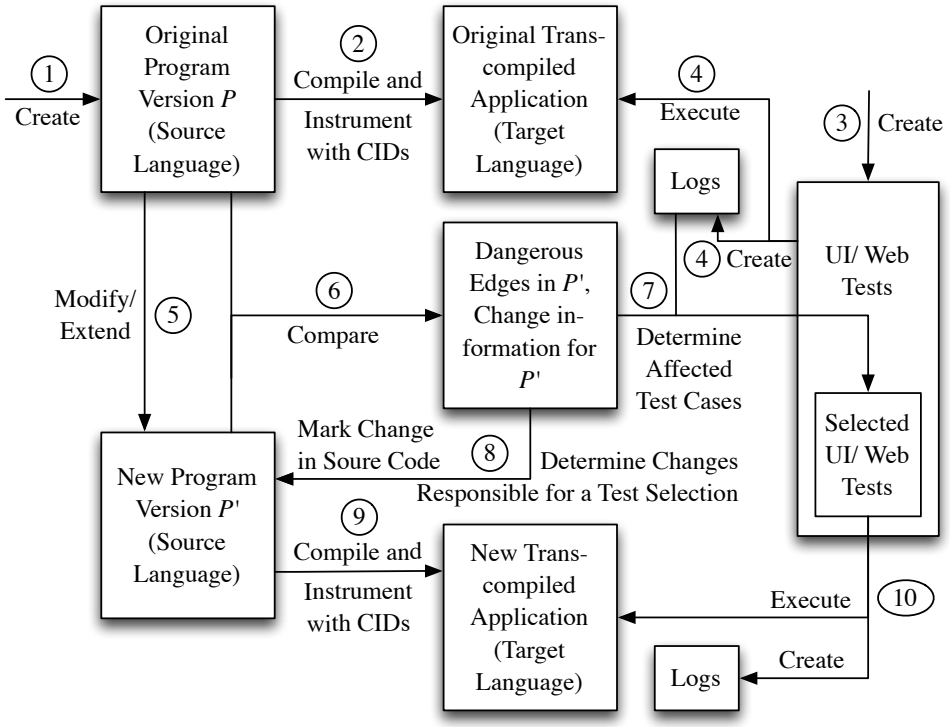


Figure 4.4: Steps in our technique.

tain application code in the desired target language. The instrumentation and transcompilation is done automatically, for example via a plug-in in the IDE. Depending on the kind of application, the developers create UI/web tests (see ③). In step ④, they apply the UI/web tests to ensure that the application behaves as expected. While running the application, the tests traverse instrumentation code that produces log files that contain information which tests execute which parts of the code.

Until now, our RTS approach did not change the overall workflow at all. Each of the steps presented so far has to be done anyway. The only differences are that the transcompiled code contains instrumentation code, and that running a test case produces automatically a corresponding log file. Here, the structure and the location of the instrumentation code in the source code are decisive for both the web test selection and the fault localization in the source code. Details about instrumentation follow in Section 4.5 and in Section 4.6.

Now, the developers continue with the development of the application as usual. They modify or extend the application (in the source language), respectively. As a result, they get a new version P' (see ⑤). The target of the next step ⑥ is to determine all the code changes made in P'; or more specifically, to find all the dangerous edges and finally the modified nodes. To this end, P' is compared with P. By combining this information about changes with the data in the logs, it is possible to identify all the tests that are affected by

the code modifications and that have thus to be rerun (see ⑦). Please note that this is not straightforward. In case of web applications, the data collected during web test execution are only available in the web browser. Writing these data to the local hard drive automatically is not possible due to the sandbox security mechanism. Similar problems might also occur in mobile applications. We solve this problem by means of an own logging server. Based on the test selection and the information obtained in step ⑥, we can additionally mark the code modifications in the source code (application written in the source language) and we are able to present the developers a set of changes that might be responsible for the test selection (see ⑧). All the work for selecting tests and localizing faults is done. The developers transcompile the application (see ⑨) and execute the selected tests (see ⑩).

The transcompiled application should also contain instrumentation code as there might be new or modified web tests without corresponding logs that are required for subsequent analyses. Another reason is that a test case might traverse other parts of the source code due to a modification. Besides, as already described by Elbaum et al. [72], information that describe the program execution of a test seems to become invalid already after rather small code changes. We have also observed such a phenomenon during our research. So we can confirm the findings of Elbaum et al. Consequently, we usually re-create logs for tests that have been selected for re-execution.

Compared with the usual development workflow, parts of step ② (the instrumentation), parts of step ④ (the creation of logs) and the steps ⑥ to ⑧ are new. A decisive step is the comparison of P' with P (see ⑥). It enables us to detect all changes including those that affect client-side code. This is essential for selecting tests that check client-side functionality only. Of course, the comparison is applicable on both client-side code and on server-side code and gives us the information for doing the test selection. For this purpose, we generate our own CFG that is based on the source language. The next subsection explains how this is done.

Constructing the CFG

In order to be able to do a comparison of two program versions P and P' that are written in a specific source language, we represent each of the two versions by an *abstract syntax tree* (*AST*). As already outlined in Section 2.5, there are many tools available that create ASTs for different programming languages. Our approach is currently implemented for applications written in Java. Nevertheless, it is also applicable in other source languages with appropriate transcompilers. We discuss this in more detail in Section 4.9. In our implementation, we obtain the ASTs with the aid of the Eclipse Java Development Tools (JDT) [69] and the Eclipse built-in parser [62]. From this, we can construct an extended version of Harrold et al.'s JIG (see Section 4.3.1), which we call the *Extended Java Interclass Graph* (*EJIG*). The EJIG outperforms the JIG as it differs from the JIG in many details:

Supporting Today's Language Features and Handling Transcompiler Specifics: First of all, the EJIG is by far more up to date as it supports Java language elements up to Java version 7. (We explain later in this section in Paragraph "Preconditions" why we have decided to relinquish the implementation of Java 8 features in the EJIG.) Today for example, there are new syntactical elements such as Java Generics, Enums, or foreach-loops, and we have to cope with dependency injection as an additional way to instantiate referenced objects during initialization without passing pre-instantiated objects as arguments. Beyond that, transcompilers might bring along even more peculiarities. GWT for example introduces a special entry point (`onModuleLoad`, see Section 3.2) and it offers additional annotations (e.g. `@UiHandler`) in order to facilitate the binding of event handlers [99].

These language specific demands have induced us to introduce four artificial nodes in the EJIG similarly to the external code node (ECN), which has been introduced by Harrold et al. in their JIG in order to handle calls from external libraries (see Section 4.3.1). Our first artificial node is labeled *InjectedNode*. We connect each node that can be accessed via dependency injection to the InjectedNode via a call edge. For the modeling of code that is annotated with `@UiHandler`, we act in the same way and use a node *UiHandler*. As the entry method `onModuleLoad` invokes implicitly a (default) constructor, we add an extra *StartupNode*. Finally, the fourth node labeled *RestECN* connects all methods that cannot be reached. This is helpful to analyze code that is not referenced by productive code yet. We might think of code that implements new features. So, the RestECN node is another kind of entry node.

Precision of the Analysis and the Model: A major improvement of the EJIG embraces precision. The EJIG allows to drill down on the AST arbitrarily. We are not limited to solely methods (in contrast to e.g. Ren et al. [224]) or statements (e.g. Harrold et al. [126]). Instead, we can choose dynamically to represent classes, methods, statements, or even every single expressions in the EJIG. This has several advantages. First, it is possible to precisely model for example conditional expressions. (This is also important for solving the coverage identification problem, see Section 1.2.) Second, it is very useful to localize faults in statements consisting of many nested expressions. We might think of loops or nested method calls (in the JDT: method invocations). So we are able to isolate precisely the reason for a change, represented as node in the AST.

Technically, we use functionality provided via the JDT [62, 69] to parse the Java source code of each `ICompilationUnit` (see Section 2.5). Figure 4.5 shows the corresponding piece of code. We collect all the `ICompilationUnits` from the packages and put them in a list (see lines 2, 7-8). In line 13 we instantiate a new Parser that handles Java code according to the Java Language Specification `JLS4`. This corresponds to Java code up to Java SE 7 [67]. Afterwards (see line 17), we create an AST for each `ICompilationUnit` and collect all methods, constructors, and initializers. Starting from these elements, we apply the JDT's `ASTVisitor` [62] to traverse the corresponding ASTs.

```
1  private void createASTsForCompilationUnits(IJavaProject javaProject,
       IProgressMonitor monitor) throws JavaModelException {
2    List<ICompilationUnit> units = new LinkedList<ICompilationUnit>();
3    IPackageFragment[] packages = javaProject.getPackageFragments();
4    for (IPackageFragment p : packages) {
5      if (p.getKind() == IPackageFragmentRoot.K_SOURCE) {
6        // ...
7        for (ICompilationUnit unit : p.getCompilationUnits()) {
8          units.add(unit);
9        }
10     }
11   }
12   this.astRequestor = new EclipseASTRequestor();
13   ASTParser parser = ASTParser.newParser(AST.JLS4);
14   parser.setResolveBindings(true);
15   parser.setKind(ASTParser.K_COMPILATION_UNIT);
16   parser.setProject(javaProject);
17   parser.createASTs(units.toArray(new ICompilationUnit[units.size()])
       , new String[0], astRequestor, monitor);
18 }
```

Figure 4.5: Parsing Java source code.

For a better understanding, Figure 4.6 shows a piece of Java code that we have taken from Harrold et al. [126, Figure 3b]. Originally, they defined the method `bar` (see lines 14-17) without any surrounding class. In order to obtain valid Java code, we adapted the code slightly and put it into an extra Main class (see lines 10-18) with a `main` method (see lines 11-13).

```
1  // A is externally defined and has a public static method foo()
2  // and a public method m()
3  package testpackage;
4  class B extends A {
5    public void m() {...};
6  }
7  class C extends B {
8    public void m(){...};
9  }
10 class Main {
11   public static void main(String[] args) {
12     new Main().bar(new B());
13   }
14   void bar(A p) {
15     A.foo();
16     p.m();
17   }
18 }
```

Figure 4.6: Inheritance in Java code, adapted version taken from Harrold et al. [126, Figure 3b].

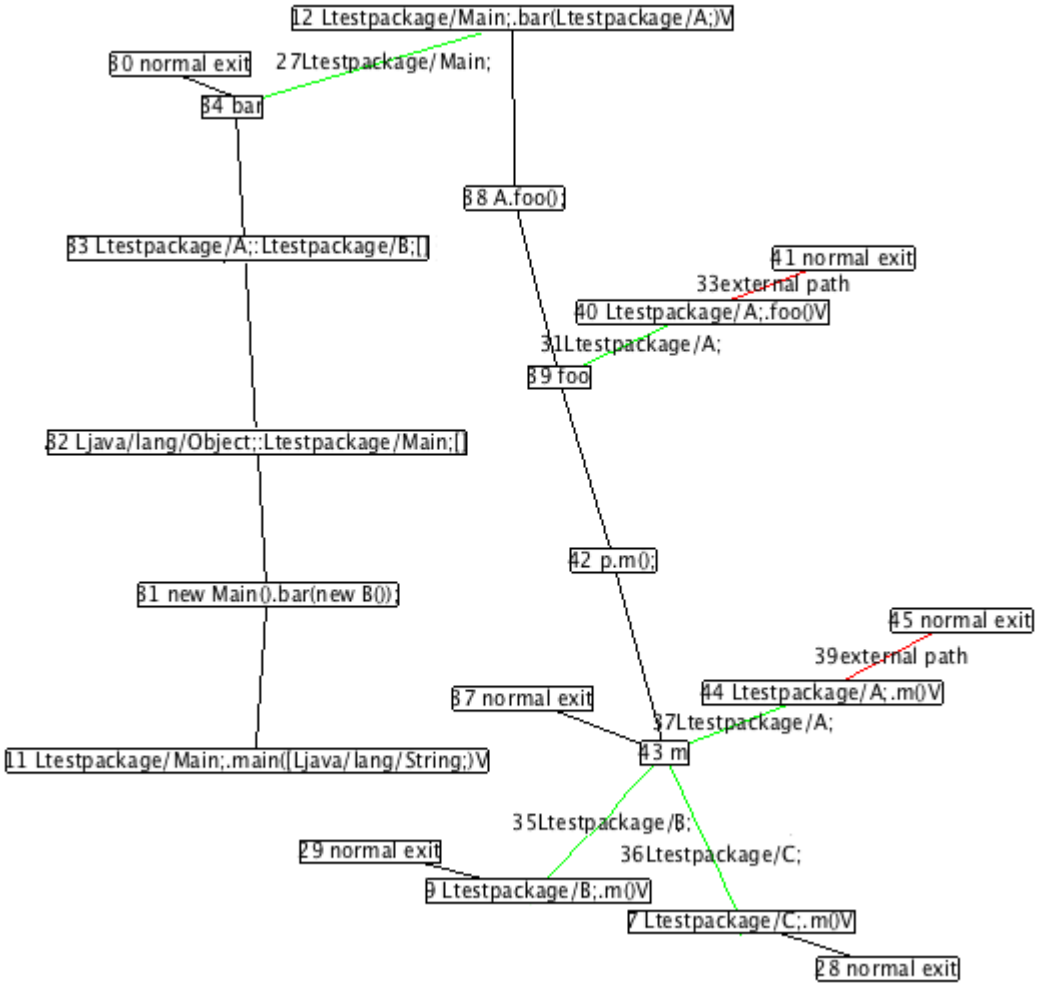


Figure 4.7: EJIG with bindings.

Figure 4.7 illustrates the relevant part of our resulting EJIG that we obtain when traversing the AST of the `main` method and its call to `bar` with the `ASTVisitor`. The figure is created with the aid of the GraphStream library [118]. As we can see, we create a node for every single element of interest that we want to represent in the EJIG. This includes the methods (e.g. node 12, `bar`), constructors, and initializers themselves that we collected before (see Figure 4.5), but we also create nodes for statements (e.g. node 38, `A.foo()` and node 42, `p.m()`), special expressions such as conditional expressions, and method calls (e.g. node 44, `A.m()`, node 9, `B.m()`, and node 7, `C.m()`). For representing calls to external methods, we adhere to the principle proposed by Harrold et al. [126] and solely model the call itself. As the external method might be part of a library whose code is not available, we just model the rest of the method as path edge to an exit node (see the red edges 33 and 39 labeled

as `external path`). Call edges are represented as green edges. In case of the invocation of method `m`, there are three possible methods in class `A`, `B`, and `C`.

Node Structure: Another detail worth considering is the node structure in the EJIGs. Conformable to the doctrine, Rothermel, Harrold, and colleagues [126, 233] model return nodes for each method even if there does not exist a corresponding statement in the code. Figure 4.8 shows Harrold et al.'s JIG [126] of the `bar` method presented in Figure 4.6. Please note the return node after node 7 (`A.foo();`). In contrast, we do not introduce return nodes artificially and model return nodes only if they are present in the code.

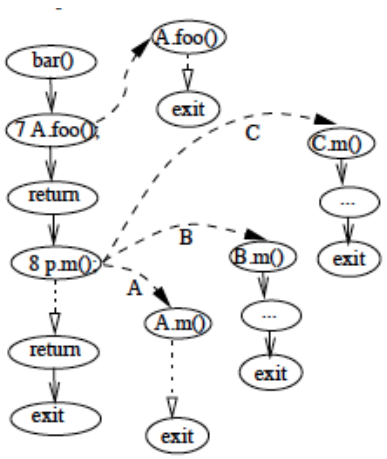


Figure 4.8: Artificial return nodes in Harrold et al.'s JIG [126, Figure 3b].

Dealing with Equally-Named Elements and Class Hierarchies: Yet another improvement in the EJIG includes the effort to do the whole analysis. In the EJIG, information about object types are crucial for an adequate representation of the control flow. The necessary identification of classes is done for us in the Eclipse AST by means of the Eclipse *bindings* mechanism [62, 64] which resolves all identifiers occurring in the parsed code. Bindings contain sophisticated information about references, types, super classes, and many more details. This enables us to create globally qualified class names to represent object types. This way, we are able to distinguish classes with the same name in different packages as well as to differentiate between classes in inheritance chains to represent polymorphism. Unlike Harrold et al.'s approach [126], there is no need of doing a separate class hierarchy analysis to determine all the methods that can be reached by a call due to polymorphism and dynamic binding. Bindings offer all the information we need to perform the necessary calculations in a fast and effective way when parsing the AST and creating the EJIGs. In Figure 4.7, the nodes and call edges are already labeled with an appropriate binding. If a new class is instantiated, we always use the globally qualified class name. For example,

the code in Figure 4.6, line 12 (`new Main().bar(new B());`) and node 32 or node 33 in Figure 4.7 illustrate this. Here, the label for the `Main` Object is `java/lang/Object:testpackage/Main`. The label of the class `B` is `testpackage/A:testpackage/B` and reflects the inheritance chain.

In Figure 4.10 and Figure 4.11, we give some more insights how to model statements and expressions. To this end, we use example source code of a program version P and its modified version P'. The code is depicted in Figure 4.9. Line 10 of Figure 4.9a contains a conditional expression. An excerpt of the corresponding EJIG is shown in Figure 4.10. We model each branch of the conditional expression explicitly (see nodes 26-30 in Figure 4.10). In more complicated conditional expressions, this helps us to precisely locate possible faults. Moreover, it helps us to calculate for example the branch coverage to solve the coverage identification problem (see Chapter 7).

```
1  package test;
2
3  public class A {
4    protected Object o =
5          new Object();
6  }
```

```
1  package test;
2
3  public class B extends A {
4    private int minVal = 0;
5
6
7
8    private void minVal(int b) {
9      int a = 1;
10     minVal = (a < b) ? a : b;
11     System.out.println(minVal);
12   }
13
14   private void m() {
15     o = 1;
16   }
17
18   public static void main(
         String[] args) {
19     B b = new B();
20     b.minVal(2);
21     b.m();
22   }
23 }
```

(a) Source code in P

```
1  package test;
2
3  public class A {
4    protected Object o =
5          new Object();
6  }
```

```
1  package test;
2
3  public class B extends A {
4    private double minVal = -1.0;
5    private Object o =
6          new Object();
7
8    private void minVal(int b) {
9      int a = 1;
10     minVal = (a < b) ? a : b;
11     System.out.println(minVal);
12   }
13
14   private void m() {
15     o = 1;
16   }
17
18   public static void main(
         String[] args) {
19     B b = new B();
20     b.minVal(2);
21     b.m();
22   }
23 }
```

(b) Source code in P'

Figure 4.9: Statements, conditional expressions, and fields in the EJIG.

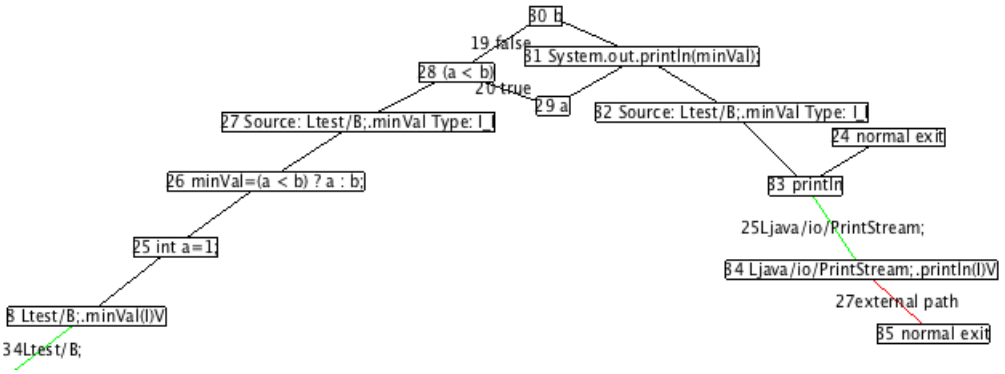


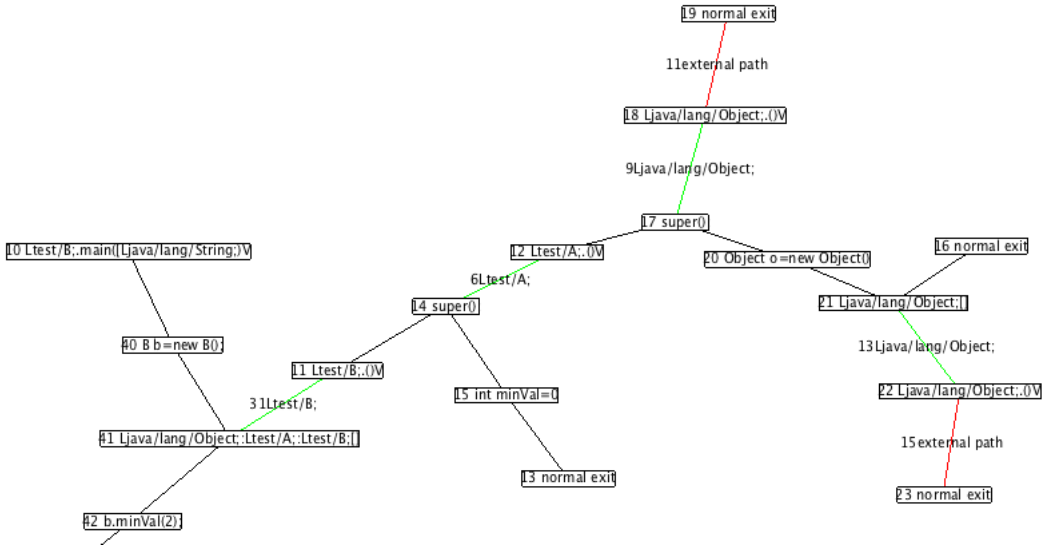
Figure 4.10: Conditional expression in the EJIG of P.

Modeling Fields: Another peculiarity affects global variables/fields and the direct initialization of global variables (in Java: field initialization, see e.g. class `A` or class `B`, line 4 in the example in Figure 4.9). To create the EJIGs, we traverse all initializers, constructors and methods as described above. But unfortunately, fields whose initialization is beyond a constructor do not show up in the EJIG. To reflect modifications in the field initialization, we model them in the EJIG explicitly as part of the constructor or – in case of static fields – as part of an initializer. If there is no constructor or initializer, we introduce the appropriate element artificially and model the fields therein. Figure 4.11 illustrates this. Instantiating a new object `B` invokes the default constructor (see node 11 in Figure 4.11a). It invokes the super constructor (node 12) which in turn initializes `Object o = new Object()` (node 20). Afterwards, the constructor of class `B` initializes `minVal`.

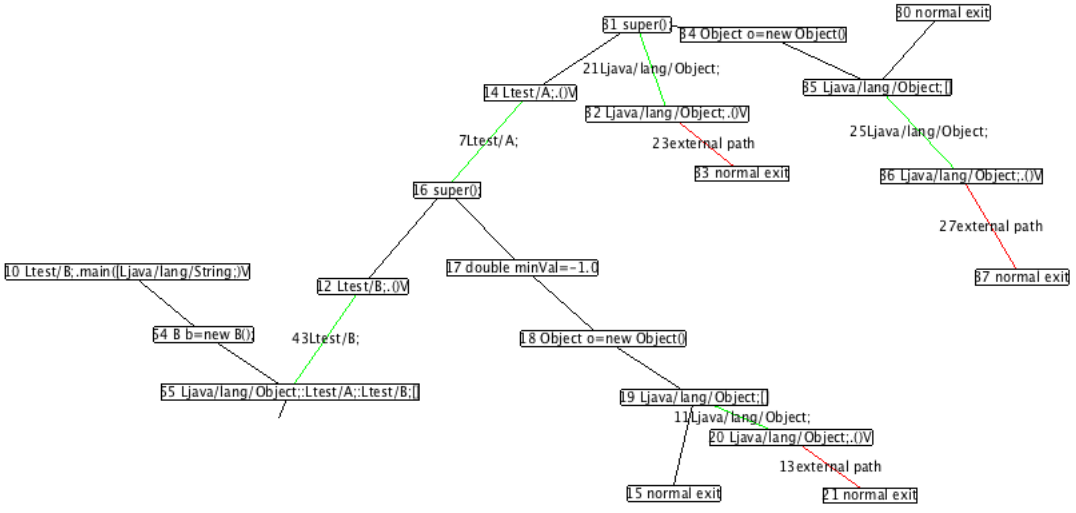
The label of a field integrates both the type and the location of a field in the source code. This is important to recognize type modifications or changes in the initialization. For example, we might think of a change from `int` to `double` in the field `minVal` and the assignment of another value (see Figure 4.9b, class `B`, line 4). This is reflected in the EJIG of P' in Figure 4.11b (compare node 17 with node 15 in Figure 4.11a).

Beyond that, we detect when a field declaration hides another field declared in a super class. The code in Figure 4.9b introduces a new field `o` (see class `B`, line 5 and node 18 in Figure 4.11b). The method `m` (see lines 14-16 in P', Figure 4.9b) re-initializes `B.o` rather than `A.o` as shown in the code of P. The EJIG is fully aware of this change. Please compare node 38 in Figure 4.12a with node 52 in Figure 4.12b and note the change in the type from `test/A` to `test/B`.

Handling Inner and Anonymous classes: The last adaptation affects the handling of inner classes and anonymous classes. In Java, top level classes may comprise anonymous classes and inner classes. Despite binding information, the usage of anonymous and inner classes brings along naming problems when



(a) Field `minVal` and `o` in P.



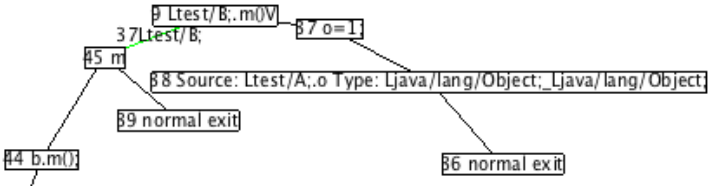
(b) Field `minVal` and `o` in P'.

Figure 4.11: Fields in the EJIGs.

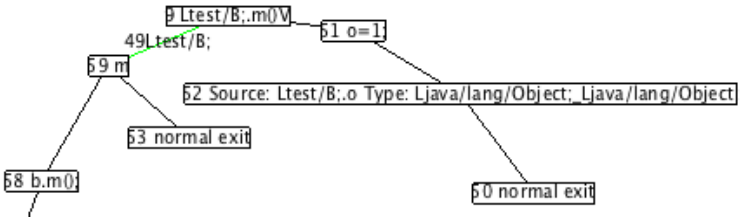
modeling their globally qualified class names. For example, already Ren et al. [224], stated:

> "In a JVM, anonymous classes are represented as `EnclosingClass-Name$<num>`, where the number assigned represents the lexical order of the anonymous class within its enclosing class." [224, page 438]

If another anonymous class has been added or one of the existing anonymous classes has been removed in a new program version, this might lead to changes



(a) Overridden field `o` in P.



(b) Overridden field `o` in P'.

Figure 4.12: Statements, expressions, and fields in the EJIG.

in the name representation of the remaining anonymous classes. As a result, it complicates a consistent representation in the EJIG, which is important for the subsequent comparison of the two program versions. (The comparison algorithm will be introduced in Section 4.3.2). Consequently, there might occur failures to reliably detect changes made in anonymous classes or it might happen that wrong anonymous classes are compared with each other. Figure 4.13 illustrates this. We explain it in the next paragraphs.

The code in Figure 4.13a has just one anonymous class (`KeyListener`, line 9). Using Eclipse's binding directly, this class would be labeled with number `$1`. Accordingly, the label of the method `keyPressed` would be `test/OuterClass;.()$1.keyPressed(java/awt/event/KeyEvent;)`. That is, it would consists of the package name `test`, the name of the top level class `OuterClass`, the default constructor `.()`, the number `$1`, the method name, and the globally qualified class name for `KeyEvent`.

The code in Figure 4.13b, has a second anonymous class (`ActionListener`, line 4). The rest of the code remains unchanged (see lines 9-16). Due to this change, the anonymous class `KeyListener` would be labeled with number `$2`. Naturally, the label of the method `keyPressed` would change to `test/OuterClass;.()$2.keyPressed(java/awt/event/KeyEvent;)`. For this reason, the method would be erroneously recognized as code change although it did not change at all.

So, there are some enhancements necessary to identify anonymous classes correctly. Ren et al. [224] use a combination of the top level class name, the name of the method containing the anonymous class, the class name of the anonymous class itself, and a number. We basically follow their strategy, but we propose to consider parent nodes in the AST (e.g. `button.addKeyListener`) in order to create a unique label. This solves a naming problem that has been

```
1  public class OuterClass {
2    private JButton button = new JButton("Press me!");
3      public OuterClass() {
4          // In P', there will be another anonymous class in lines 4–8
9          button.addKeyListener(new KeyListener() {
10             @Override
11             public void keyPressed(KeyEvent e) {
12                 press();
13             }
14         });
15     }
16 }
```

(a) Anonymous class in program version P.

```
1  public class OuterClass {
2    private JButton button = new JButton("Press me!");
3      public OuterClass() {
4          button.addActionListener(new ActionListener() {
5              @Override
6              public void actionPerformed(ActionEvent e) {}
7          });
8
9          button.addKeyListener(new KeyListener() {
10             @Override
11             public void keyPressed(KeyEvent e) {
12                 press();
13             }
14         });
15     }
16 }
```

(b) Additional anonymous class in version P'.

Figure 4.13: Problems when generating globally qualified class names for anonymous classes.

reported by Ren et al. themselves. According to this, their naming solution could fail "when two anonymous classes occur within the same scope and extend the same superclass" [224, page 438].

Comparing the EJIGs of two program versions

For traversing and comparing the EJIGs of two program versions, we apply the basic principle depicted in Section 4.3.1. However, we have adapted several details which we want to present in this subsection.

The first adaptation concerns the comparison and the expected results. The purpose of Rothermel, Harrold, and colleagues [126, 233] is to identify those parts of the code in the original program P that have been changed in the new

program version P'. The output of their `compare` algorithm ([126], Figure 7) is a set of dangerous edges for P. This is inadequate for our purposes as we want to localize faults in the *new, refined* program version P'. In doing so, we would like to have support from the IDE in use. That is, we wish to obtain a view that lists all the changes in P'. When selecting one of the changes, we expect to switch to the corresponding piece of code in P'. The old version P is not available any more. So, the output of our algorithm is a set of dangerous nodes of the *new* program version P'. With this information, we can map the affected test cases directly to statements in the code that might be responsible for the change (see Section 4.5 and Section 4.6, respectively). Algorithm 1 depicts the details of our comparison algorithm.

When comparing our algorithm with the one presented by Harrold et al. [126], one difference affects the entry nodes. In case of Harrold et al. [126], these include entry nodes of the `main` method, as well as entry nodes of methods declared as `static`. As described before, we additionally consider GWT-specific entry nodes and dependency injection. Furthermore, Harrold et al. seem not to take into account that such an entry node might have been modified. (At least, their algorithm does not indicate it.) For example, we might think of moving the `main` method to another package. For this reason, we at first check the labels of the nodes for equivalence (see line 1 in Algorithm 1). To do this, we follow Harrold et al. and compare the labels lexicographically. Due to the bindings mechanism, we can track any changes. If the labels do not coincide, we add the outgoing edge of the `main` method to list of dangerous edges. For this purpose, we use a helper method `getAnyEdge()` that determines all the outgoing edges of a node (see lines 2-3). In a subsequent analysis (more details follow in Section 4.3.3), we are able to identify in general which node (start node or target node) of the dangerous edge has been modified.

Unlike the algorithm of Harrold et al. [126], our algorithm dispenses with recursion. This avoids stack overflow problems in Java-like languages that rely on the JVM. So in line 5 (see Algorithm 1), we push the start nodes s and s' onto a `stack` and start a loop to traverse the corresponding EJIGs (see line 6).

For the moment, the rest of our `compare` algorithm follows the comparison algorithm of Harrold et al. [126] in its basic principle except for our modifications (operating on P' rather than on P and the adaptations explained in Section 4.3.2, "Constructing the CFG"). We first pop the topmost pair of nodes $n \in P$ and $n' \in P'$ from the stack (see lines 7-8) and check whether n' has been visited in a previous iteration of our `compare` algorithm (see line 9). If it has not been visited yet, we mark n' as visited (see line 12). Afterwards, for each edge leaving n, we try to find a corresponding edge that leaves n'. This is implemented in the helper method `match` (see lines 13-14). If we cannot find such an edge e' (see line 15), we continue the loop in line 12. Prior to that, we do another analysis that we introduce in Section 5.4.4. Otherwise, if we have found two corresponding edges e and e' in line 14, we check whether their target nodes t and t' (lines 19-20) coincide, too (see line 22). If this is not the case, we add e' to the list of dangerous edges (line 25) and perform – as opposed to

```
   input  : s: start node in P; s′: start node in P′
   output: d: List of dangerous edges
 1 if ¬nodeEquivalent(s′, s) then
 2 |   e′ ← s′.getAnyEdge();
 3 |   d ← d.add(e′);
 4 end
 5 stack.push(s, s′);
 6 while stack not empty do
 7 |   n ← stack.pop.oldNode;
 8 |   n′ ← stack.pop.newNode;
 9 |   if n′ is marked visited then
10 |   |   continue;
11 |   end
12 |   mark n′ visited;
13 |   foreach edge e leaving n do
14 |   |   e′ ← match(n′, e);
15 |   |   if e′ is empty then
16 |   |   |   // handle removed code; more details follow in Section 5.4.4;
17 |   |   |   continue;
18 |   |   end
19 |   |   t ← e.targetNode;
20 |   |   t′ ← e′.targetNode;
21 |   |   // handle special nodes, see Section 5.4.1;
22 |   |   if ¬nodeEquivalent(t′, t) then
23 |   |   |   // distinguish kind of modification;
24 |   |   |   // more details follow in Section 5.4.4;
25 |   |   |   d ← d.add(e′);
26 |   |   end
27 |   |   else if t′ is not marked visited then
28 |   |   |   stack.push(t, t′);
29 |   |   end
30 |   end
31 |   foreach edge e′ leaving n′ without counterpart do
32 |   |   // handle special cases; more details follow in Section 5.4.4;
33 |   |   d ← d.add(e′);
34 |   end
35 end
```

Algorithm 1: Comparing two EJIGs: `compare`.

Harrold et al. – an extra analysis to distinguish the kind of code modification (see lines 23-24). Besides, the analysis enables us to compare nodes of two program versions more in-depth even if changes have already been detected. We introduce our extra analysis once more in Section 5.4.4. If the target nodes

t and t' are unchanged, we push them on the `stack` (see lines 27-29). Finally, before restarting the outer loop (pop a node pair from the `stack`), we add all edges e' leaving n' without a counterpart $e \in P$ to the list of dangerous edges (see lines 31-34). Again, we also handle some special cases (see line 32).

Preconditions

We follow Rothermel and Harrold [233] that there should be no obsolete tests (see Section 2.1) in the test suite left. This way, they avoid test failures due to non code related factors like illegal test inputs. However, in contrast to Rothermel and Harrold, we do not consider this as a hard precondition. In practice, the implementation is fast and we need a quick feedback about which tests should be re-executed. It might happen that the developers do not recognize that a specific test in a huge test suite is obsolete. But this is no problem as the test would just fail. When localizing eventual faults in the source code, the developer will notice that the test does not fit the current implementation any more and that the test has to be adapted. Apart from that, research exist (e.g. [185]) on how to determine which tests are still usable after code modifications and which ones are not. Instead of just ignoring the obsolete tests, researches have come up with techniques (e.g. [185]) to automatically repair them.

In Section 4.3.1, Paragraph "Assumptions in the Approach", we have already mentioned that Harrold et al. [126] assume tests to be deterministic. Moreover, Fowler [86] stresses that non-deterministic tests that fail randomly are completely useless because the developer never knows whether the test failure is due to a bug or due to other factors that result in non-determinism. We follow their argumentation and take this as precondition, too. Nevertheless, especially in web applications, we are facing many factors that threaten this assumption. We discuss possible threats in Section 4.8.3 in more detail.

As classes in external libraries are usually available in bytecode only, GWT is not able to compile them. So, external classes are not analyzed by our tool. As a precondition, we consider external code as well-tested.

Some standard Java language elements are currently not supported by the implementation of our technique. This includes reflection and concurrency. However, this means no restriction. GWT does not support reflection in general [103, 105]. The same applies to concurrency [105]. So, if JavaScript is used as the target language, multi-threading does not exist. GWT just ignores concurrency-related keywords like `synchronized`. Of course, there are other target languages that support concurrency and multi-threading in order to utilize multi-core processors efficiently. For this purpose, our approach would have to be extended. We could imagine that the idea of Apiwattanapong et al. [12] to handle concurrency could be integrated.

At the time of our evaluation, the latest GWT version has been GWT 2.7. This version did not support Java 8 yet. For this reason, we decided to relinquish the implementation of Java 8 features in the EJIG. Consequently, we cannot guarantee that our prototype implementation will work correctly when

applied to code using Java 8 features like lambda expressions. Notwithstanding that, the support of these syntactical features is just a technical detail that is not decisive for our overall approach.

Currently, native JavaScript methods using JSNI [97] are not analyzed either because the AST does not represent them. Originally, we planned to support this in a later version of our tool implementation. In the meantime, though, a new concept has been introduced which is called JsInterop [94] (see Section 3.2). It is meant to be the replacement of JSNI. For this reason, we expect that the developer is aware of this fact. If the developer cannot cope without using JSNI, we expect that these parts of the code are tested with existing special tools like Selenium [252], UnitJS [271], or GwtMockito [169].

4.3.3 Localizing Changes in the Source Code

After the comparison of the EJIGs has been finished, we have a set of dangerous edges, consisting of a start node and a target node. Of course, we want to know exactly which of the two possible nodes has been changed because this is essential for the test selection and – provided that there are test failures – the subsequent fault localization. Moreover, we can provide a direct link to the underlying code in the source language. Technically, the changed node is represented by an `ASTNode` in the Eclipse AST. So we can highlight the appropriate piece of code in the Eclipse IDE. Besides, we can provide extra views with a list of all the code changes and we are able to bookmark the modifications in the code editor.

In order to decide whether the source node or the target node of the dangerous node has been changed, we have to consider many different cases. We discuss the most important ones:

Regular Case: In the easiest case, a dangerous node indicates that the start node coincides in P and in P', but the target node differs. So it is the target node that represents the code change and thus, we can highlight the corresponding ASTNode in the Eclipse view. This *standard rule* applies in case of simple statements within methods or constructors.

Artificial Nodes: As another example, let us imagine that the dangerous edge starts with an artificial node (i.e. InjectedNode, UiHandler, StartupNode, or RestECN). Here, the target node must have been changed as the artificial node has no real counterpart in the source code.

There are many more similar cases where the standard rule applies. However, there are also lots of exceptional cases:

External Methods: Let us imagine that the start node of the dangerous edge is an external method. Then, the call must have been changed. As we usually do not have access to the code of the external method, we mark the start node as modified to register that there is a change.

Exit Nodes: Similarly, when the CFG of P' suddenly ends with an exit node, some nodes have been deleted. As we cannot mark removed code in the Eclipse editor, we highlight the start node.

Main Methods: Finally, we want to revisit modified entry nodes. In the previous section, we briefly considered changes in the `main` method of P'. Here, the start nodes representing the `main` methods in P and in P' are not equivalent. So once more, we identify the start nodes as changed.

4.4 Basic Instrumentation Approach

In regression test selection, it is a prerequisite to know for each single test which code it traverses in an initial program version P. Only with this information, it is possible to determine these tests that cover modified code in a new program version P'. A very popular way to obtain this knowledge is to instrument the application's code and to run every test case in order to collect some kind of coverage information during the test run. As already discussed in Section 2.3, there are basically two possibilities where to insert instrumentation code: plain source code or any kind of target code such as bytecode. Again, some transcompilers directly take the source code as input and compile it to a target language. In case of GWT-based web applications for example, client-side Java code is compiled to JavaScript. Java bytecode is never considered during that process. Besides, developers usually only deal with the source language. Code of the target language might be obfuscated (e.g. GWT, see Figure 3.2 in Section 3.1, or NeoMad [202]). In this case, it is very difficult to comprehend what the code in the target language actually does, why a test case fails and which code in the source programming language is responsible for the failure. So analyzing the code of the target language is not helpful when trying to localize faults in the source code written by the developer. For these reasons, a generic solution requires instrumenting the source code.

4.4.1 Challenges when Instrumenting Source Code

Instrumenting source code always rises four main questions: (a) Which information do developers expect to obtain by instrumenting source code? (b) How should the instrumentation code look like in order to provide the required information? (c) How can we manage to access the data produced by the instrumentation code? (d) Where should the instrumentation code be added? Answers to these questions should be as generic as possible in order to be usable for many different transcompiler frameworks and in order to find solutions to the three problems depicted in Section 1.2. We therefore want to inspect the questions and their implications in more detail:

Question (a) addresses two different aspects. First, which purposes do developers have in mind when instrumenting source code? Which kind of information do they need? Is it enough to get information whether or not a

certain piece of code has been traversed by a test or do developers also want to know how intensively and thoroughly a piece of code is covered by a test case? Besides, what is actually a piece of code? Is it a function or should even more fine-grained components of the code be reflected? This leads to the second aspect: In which form should the information be provided? Possible forms start with simple result lists for every test case and might end with extensive datasets in databases.

In order to find an adequate solution to the two aspects imposed by question (a), we have to think about how to encode the required information. Besides, we have to find a general pattern to structure instrumentation code internally. This is the focus of question (b).

Question (c) asks for a solution how we can obtain the data that describe which UI/web tests have traversed which parts of a mobile application/web application. These data are the basis for deciding which tests have to be selected for re-execution and for determining a set of changes in the source code that might be responsible for a test failure. But in web tests for example, the data are only available in the browser and it is not possible to write them to disk directly. Besides, the size of the browser cache is limited. So, collecting these data in the browser hits the wall quickly.

Finally, question (d) considers both syntactic and semantic requirements on instrumenting source code. As already explained at the beginning of this section, we create instrumentation code for the source code. Following the usual procedure of source code instrumentation, the additional instructions are injected to the software's source code, thus must fulfill the syntax rules of the corresponding source language. In the context of transcompilers, both source code and instrumentation code will then be transcompiled into the target language. The additional instrumentation code is always semantically related with a piece of the source code. In the context of transcompilers, it is crucial that the semantic relation still holds after compiling the source code into the target language.

In the next four subsections 4.4.2 - 4.4.5, we give answers to the questions (a) - (d).

4.4.2 Purpose of Code Instrumentation and Expected Information

Purposes of Our Code Instrumentation: In order to solve our three main concerns (test effort reduction problem, fault localization problem, coverage identification problem, see Section 1.2), we need to gather information about which test traverses which parts of the source code. This enables developers to precisely select test cases that cover code modifications. The information has also to be usable to assist developers in identifying code changes that are responsible for test failures. Finally, the code instrumentation has to assist developers in judging which parts of the code need additional testing.

Required Kind and Precision of Information: Basically, it is a simple yes-no decision to answer the question whether or not a specific piece of code is ever traversed by a test. Encoding this information just requires a boolean variable. However, for estimating how thoroughly a specific piece of code is tested, it is better to use more sophisticated instrumentation code that provides information about which piece of code is traversed how often in total, maybe even per test case. For this reason, our future test infrastructure must have the following properties: It has to produce instrumentation code that 1) unambiguously identifies a piece of code and that 2) reports whenever this code has been traversed by a test. Based on this information, the infrastructure has to keep track on 3) how often a specific piece of code has been traversed (in total or even by a specific test).

Source code instrumentation can be imagined at different *granularity levels*, depending on the pieces of code we consider. For example, instrumentation code can represent source code at function level, at statement, or even at expression level. If we would just be interested in whether a function has been traversed, it would be enough to instrument functions. But as we want to determine as exactly as possible the location of a code change, this information is too imprecise. The same is true when judging the quality of test cases. With instrumentation code on function level, we of course can solely provide values about function coverage. Naturally, a finer instrumentation level needs to insert more instrumentation code to identify these syntactical elements. This increases the runtime, but offers execution information even for fine-grained elements like conditional expressions, which are supported by many C-based languages via a ternary operator. This enables us to check whether all possible branches or whether a critical statement is tested thoroughly. We offer and evaluate in our regression test selection approach various *instrumentation levels* which can be defined by the user. We also investigate two different ways to provide the data collected during instrumentation, a simple file-based approach and a database-based approach.

4.4.3 Our Code Instrumentation Structure

In this Section, we define how our instrumentation code has to look like in order to meet all requirements defined in Section 4.4.1.

Discussing Existing Ways of Code Instrumentation

When deciding how the structure of our instrumentation code should look like, it could be helpful to consider already existing approaches. CodeCover [44] is a tool for calculating a test suite's code coverage. In order to provide these data, they need to instrument code, too. We would like to explain their way of instrumenting source code in an example (see Figure 4.14) that is taken from a publication of Hanussek et al. [124, page 19] which describes the functional principle of CodeCover:

```
<statement1>

<statement2>

<statement3>
```

(a) Statements of a program.

```
counter1 := counter1 + 1
<statement1>
counter2 := counter2 + 1
<statement2>
counter3 := counter3 + 1
<statement3>
```

(b) Statements with added instrumentation code in a program.

Figure 4.14: Method for instrumenting code used by CodeCover; the example is taken from Hanussek et al. [124, page 19].

Figure 4.14 shows the pseudo-code of a program. Figure 4.14a shows three statements. In Figure 4.14b, they have been supplemented with instrumentation code. As we can see, CodeCover uses simple counters to do this. The authors realize the counters as serially numbered variables, but they emphasize that any other data structure (e.g. arrays) being able to hold a value is equally convenient. When running a test on application code instrumented like that, we obtain as output a set of variables for each test case. The variables are representatives of the elements of interest that have been executed.

While serially numbered variables are sufficient to just decide which parts of the code have been traversed by a test, they are not suitable for test selection. The main problem is that any kind of relative numbering requires us to always re-execute all test cases. Thus, we would have to stick to the retest-all approach. We would like to explain this in a modified version of the example taken from Hanussek et al. [124]. Figure 4.15 shows the altered code:

```
f() {
counterS1 := counterS1 + 1
<statement1>
counterS2 := counterS2 + 1
<statement2>
counterS3 := counterS3 + 1


<statement3>
}

g() {
counterS4 := counterS4 + 1
<statement4>
}
```

(a) Program version P

```
f() {
counterS1 := counterS1 + 1
<statement1>
counterS2 := counterS2 + 1
<statement2a>
counterS3 := counterS3 + 1
<statement2b>
counterS4 := counterS4 + 1
<statement3>
}

g() {
counterS5 := counterS5 + 1
<statement4>
}
```

(b) Program version P'

Figure 4.15: Instrumentation before statements in two consecutive versions.

Let us assume that a test t_1 runs the function `f()` in Figure 4.15a, whereas another test t_2 runs the function `g()`. Now, version P will be refined to version P' (see Figure 4.15b). `<statement2>` has been replaced by `<statement2a>` and a new statement `<statement2b>` has been inserted. In order to obtain data about the new statements, it is necessary to re-instrument the source code after a code modification. As we can see, this indispensable procedure moves the locations of `counterS2`, `counterS3`, and `counterS4`. `counterS2` and `counterS3` are still in the same function, but `counterS4` has become part of `f()`. Moreover, a new counter occurs in `g()`. So in version P', some counters have been relocated and others have been added compared to version P. This is not restricted to statements. The same applies when adding or removing branches, loops or functions. As a consequence, test t_1 covers `counterS1`, `counterS2`, `counterS3` and `counterS4` and t_2 covers `counterS5`.

So, despite the function `g()` has not been changed, the coverage information of t_2 differs (`counterS5` instead of `counterS4`) and must be determined again for P'. This implies that the entire coverage information is not usable any more because we do not know in advance which statements in other functions are affected in the same way. For this reason, all test cases would have to be re-executed in order determine the coverage information. Of course, this is completely contradictory to our target of re-executing only these tests that cover a code change in P'. We never want to rerun all tests in order to restore their execution history. Consequently, it should never be necessary to re-execute test cases that are not affected by a code modification. Besides, we do not want to re-instrument source code that has not been changed. As we can see, this relative kind of numbering code elements can be used only once to collect coverage information for a certain version of the application and becomes invalid as soon as the source code has been changed.

In the Related Work Section 4.2, we have discussed some more approaches and we have already stressed their deficiencies. Please remind that entry/exit events (see the approach of Geimer et al. [92]) or whole blocks (see Larus [171] or Ayers et al. [23]) are too imprecise for a precise fault localization and a test selection that re-executes as less tests as possible.

Another possible solution for instrumenting code has been presented by Rothermel, Harrold, and colleagues [126, 233]. Rothermel and Harrold instrument the application code to obtain a "*branch trace* that consists of the branches taken during this execution" [233, page 176]. Thereof, they determine edges covered by a test. Harrold et al. process these data further in order to get an *edge-coverage matrix* [126, page 315]. By using the dangerous edges that identify modified edges in the CFG, they are able to select those tests that are affected by code changes.

We are interested in a more straight-forward solution that goes beyond the consideration of edges in order to localize modifications in the source language. For this reason, we develop an own approach to instrument test cases that addresses all the challenges listed in Section 4.4.1. This way, we will also be able to check how thoroughly a particular piece of code is covered by tests.

Developing an Own Code Instrumentation Structure

We propose to use as instrumentation code unique identifiers representing *code entities* like functions, statements or expressions. The basic idea resembles the procedure of Larus [171] mentioned in Section 4.2 before. But most notably, we do not linger over basic blocks but handle statements or even expressions, too. Besides, we manage to realize the code instrumentation in transcompiled applications. In order to instrument the source code automatically, we require an abstract syntax tree (AST) for each source file. Because programs in most programming languages can be represented by their ASTs, our approach is universally usable. We start to traverse the AST from its root until we reach the nodes with a certain level (see Section 2.5, Paragraph "Structure of a Tree, Basic Terms and Definitions") that corresponds to the instrumentation level defined by the user. For each node in the AST traversed so far, we compute a unique *code identifier* (*CID*). Each CID consists of a sequence of characters (in Java: a String). For the CIDs, special rules are in place. We explain them in the next parts.

Technical Demands on Code Identifier: A CID has to fulfill two conditions: It must be *deterministically computable* and it has to be *unique*.

A unique CID means that each CID represents exactly one code entity. This is necessary in order to avoid confusions that could arise if for example a statement occurs several times in the source code. More specifically, unique CIDs enable us to identify exactly which source code entities have been traversed by a test.

A deterministic computation means that none of the CIDs will change during a recalculation as long as the program code has not changed. Once the program code has been modified, only those CIDs change that are connected to modified code entities. All other CIDs remain unchanged. This implies that newly introduced or removed body declarations in the AST (e.g. functions or global variables) do not affect the CIDs of already existing body declarations (provided that they have not been changed).

Deterministically computed CIDs are necessary for two reasons. First, for selecting tests and for localizing faults in a newly created version P', we need to reconstruct all the CIDs that have been used to instrument the original program version P in order to obtain test traces. Actually, when the original program version P is transcompiled for the first time into target code, any kind of ID would be fine as long as it reflects the code structure of P and if it is able to trace the code executed by a test. However, as soon as the code evolves, we obtain a new version P' and compare it with P (see Section 4.3.2). As a result, we get all the code changes in P'. At that point, it is essential to find out which test traverses the corresponding code change. All the existing CIDs refer to code in the original version P. Consequently, we need to reconstruct the CIDs that have been traced to the target code before. Afterwards, we transfer the CIDs to P' during the comparison. If there is a code change in P', we query

the CID that represents the code entity in P and associate it with the new code entity in P'.

The second reason for using deterministically computed CIDs concerns the effort to update a test suite. Tests that are affected by a change have to be adapted. This invalidates the former test trace containing the corresponding CIDs. A new test trace with the latest CIDs has to be created. But this actually affects only those tests that are affected by a change. All the other tests must not be executed and there should be no necessity to create a new test trace. Their CIDs must still be valid. Otherwise, all the effort to analyze code for selecting tests for retest would be destroyed.

Basically, we could store the CIDs calculated for P somehow (e.g. in a database). But this entails an annoying maintenance that involves updating changed CIDs as well as adding new/removing old CIDs. As the Eclipse JDT provides a fast traversal of the AST via the visitor pattern, we recalculate them instead.

Internal Structure of CIDs: To create a CID that meets all the described technical demands, one could basically come up with the idea to encode the whole CID as hash value or to reuse Eclipse's binding. However, the bindings mechanism of Eclipse only works with named entities (e.g. types or methods) [64], but not with statements. Hash code is used for example by Ruth and Tu [240] to encode statements for the comparison of two CFGs. Nevertheless, using simple hash values in code instrumentation is insufficient as it does not take into account that for example a statement might occur several times in the same function of a file. Consequently, we would obtain the same hash value which would violate the demand on CIDs to be unique. We refer to this problem as *identical statement problem*. In order to distinguish these statements, we need a more sophisticated encoding that reflects which statement a CID refers to.

Another idea could be to generate unique CIDs for all the code entities in a file of the source language. This would completely suffice if the transcompiler would keep CIDs separated that are contained in separate files of the source language. That is, the transcompiler should maintain the source code's file structure in the target application. This way, the transcompiler could even introduce new files in the target language in order to group code more than before. The only constraint is that the transcompiler would never be allowed to combine code in a file of the target language that has been separated in the source language because this could violate the uniqueness requirement of CIDs. But in fact, some transcompilers exactly do mix up code that has been defined in different source files in the source language. An example is the GWT compiler (see Section 3.2, "Google Web Toolkit"). It creates only one file that contains all the target code. As a result, CIDs could not be kept apart if we would apply our second idea. To avoid this problem, we encode in our CIDs the path to each node in the AST. This is done by prefixing all the CIDs with their globally qualified file name. This way, we are able to distinguish CIDs originating from different source files.

Having these preliminary considerations in mind, we can define the general structure of a CID that consists of a mixture of paths, hash values and hexadecimal numbers:

```
<path_to_fileName_fileExtension><HashValue(s)><HexNumber>
```

Part 1: The `path_to_fileName_fileExtension` represents the globally qualified file name. In Java, this corresponds to the globally qualified class name.

Part 2: The next part of the CID comprises one one several `HashValue`(s). It represents type declarations (e.g. classes) and body declarations (e.g. fields or methods). CIDs of unchanged nodes in the AST must never change when code has been added or removed. They have to be stable. For this reason, we cannot use relative numbers for the identification of types or body declarations. Depending on the AST and the parser in use, this could lead to the problem that we have described in Section 4.4.3 with respect to CodeCover. New or removed methods would shift the relative numbering. We illustrate this in more detail in Figure 4.16. In the original program version P, there are two methods `f()` and `h()`. `f()` is represented by the relative CID `<pathToClassName.java001>`, `h()` is represented by the relative CID `<pathToClassName.java002>`. Both `f()` and `h()` contain the same statement `s1` (see Figure 4.16a). In method `h()`, let `<pathToClassName.java002s1>` be the CID for `s1`. For `s1` in `f()`, let `<pathToClassName.java001s1>` be the CID. In Figure 4.16b, a new method `g()` has been added in order to provide a new feature. It has also a statement `s1`. Now, `s1` in `h()` is identified by the number `003` in the CID `<pathToClassName.java003s1>`.

```
1  void f() {
2    <pathToClassName.java001s1>
3    // s1
4  }
5
6
7
8
9
10
11 void h() {
12   <pathToClassName.java002s1>
13   // s1
14 }
```

(a) Program version P

```
1  void f() {
2    <pathToClassName.java001s1>
3    // s1
4  }
5
6  void g() {
7    <pathToClassName.java002s1>
8    // s1
9  }
10
11 void h() {
12   <pathToClassName.java003s1>
13   // s1
14 }
```

(b) Program version P'

Figure 4.16: Instrumentation with relative numbers in two consecutive versions.

In order to test the new feature, a developer implements a new test t. Actually, it would be enough to create a new test trace for t. So, the test trace

would show the CID `<pathToClassName.java002s1>`. However, a previously created test trace for method `h()` in P also shows exactly the same the CID `<pathToClassName.java002s1>`. So, the CID represents two different code entities. Or more general, unchanged code entities would be represented by other CIDs than before. In the end, we would have to update the CIDs for all test traces instead of updating only those CIDs that are affected by a code change. To circumvent the problem, we exploit that names of functions and global variables have to be unambiguous. With respect to Java, class names as well as method and field names have to be unique. In order to get a general representation for these names, hash values are perfectly suitable as they are stable.

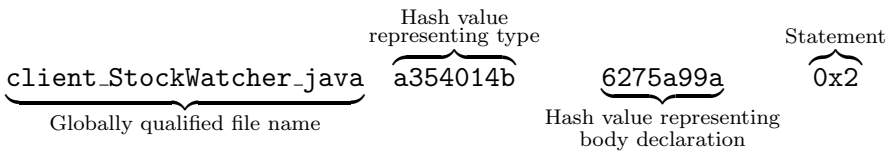
Part 3: As opposed to body declarations, statements can easily be represented relative to a method declaration. If a statement has been changed, the corresponding branch/path in the enclosing method has to be retested anyway. While doing this, a new test trace will be created for this test. We therefore have decided to encode the paths to statements relative to methods using a hexadecimal number `HexNumber`. This is the last part in our CID structure. Using a hexadecimal number usually requires less characters than using the statement as identifier and circumvents the identical statement problem. Besides, we can use a trie (see Section C in the Appendix) to persist the data in a memory-saving way.

The combination of globally qualified file name, hash value, and hexadecimal number is well-suited to serve as CID. An enhancement would have been to encode the entire combination (globally qualified file name plus hash value plus hexadecimal number) as a whole as hash value. However, we have decided against the additional encoding of the CIDs as single hash value to facilitate the debugging process. A simple hash value would make it very hard to check whether the instrumentation works as expected.

Examples: We want to put the structure of CIDs into practice. To this end, we give two code examples that use Java as source language. In the first one, we consider the initialization of a field that has been introduced manually in the constructor of the class `Stockwatcher` presented in Figure 4.2:

```
1 Button addStockButton = new Button("Add");
```

The corresponding CID is `client_StockWatcher_javaa354014b6275a99a0x2` and can be divided in four parts:



Finally, our second example (see Figure 4.17) shows code of a class that contains a method with several statements.

```
1  package test;
2
3  public class Branch {
4
5    public static void main(String[] args) {
6      if(true) {
7        ;
8      }
9      else {
10       ;
11     }
12   }
13 }
```

Figure 4.17: Simple Java example code.

ASTNode	Start	End	CID
Method-Declaration	5	12	test_Branch_java1627510bfa690ec7
IfStatement	6	11	test_Branch_java1627510bfa690ec70x40x0
Then_Statement	6	8	test_Branch_java6e016941fa690ec70x40x00x0
Empty-Statement	7	7	test_Branch_java1627510bfa690ec70x40x00x00x0
Else_Statement	9	11	test_Branch_java1627510bfa690ec70x40x00x1
Empty-Statement	10	10	test_Branch_java1627510bfa690ec70x40x00x10x0

Table 4.1: Examples of CIDs.

Table 4.1 shows the corresponding ASTNodes, their location in the code and the corresponding CID. For example, the MethodDeclaration starts in line 5 and ends in line 12.

Algorithm to Create CIDs for a Software Project: After we have specified the structure of a CID, we want to explain its calculation. In general, we use an algorithm called `calcCIDs` (see Algorithm 2) that relies on the underlying AST of the source code. It takes two arguments: the AST of a source file `f` and a set of node types that have to be represented by CIDs. These node types have to be defined by the user. All other node types will not be reflected by CIDs. In our implementation, we offer menus with predefined options. In the initial step, `calcCIDs` starts with the root node of the AST and pushes it on a stack (see line 1 in Algorithm 2). The next steps are processed in a loop. In the first place, `calcCIDs` takes the topmost node `n`

(line 3) and checks whether this node type should be represented by a CID (line 4). At the moment, this is true as `n` is the root node of the AST representing the whole file (in Java: a class). So `calcCIDs` calculates for `n` the globally qualified file name. To this end, `calcCIDs` invokes the helper function `createGloballyQualifiedFileNameOf(f)` (see line 6). Afterwards, `calcCIDs` stores the node `n` and the corresponding CID as key/value pair in a Map `s` (see line 10). Finally, our algorithm determines all the child nodes of `n` and pushes them on the stack (see line 11).

```
   input  : AST of the source file f; Set t of node types that should be
            represented by CIDs
   output: Map s that assigns CIDs to nodes n in the AST
 1 stack.push(root element of AST);
 2 while stack not empty do
 3 |   n ← stack.pop();
 4 |   if typeOf(n) in t then
 5 |   |   if n has no parent node then
 6 |   |   |   cid ← createGloballyQualifiedFileNameOf(f);
 7 |   |   else
 8 |   |   |   cid ← concatenate(getCIDofParentNode(s, n),
   |   |   |   getIdentifierOf(n));
 9 |   |   end
10 |   |   s.put(n, cid);
11 |   |   stack.push(childNodesOf(n));
12 |   end
13 end
```

Algorithm 2: Calculating CIDs: `calcCIDs`.

Now, the loop restarts in order to calculate CIDs for all other nodes in the AST. If the stack is not empty and if the type of the topmost node matters, `calcCIDs` pops a new node `n` and calculates its CID as concatenation of the CID of `n`'s parent node and the identifier of the current node `n` (see line 8). For this purpose, `calcCIDs` calls `getCIDofParentNode` which simply fetches the appropriate CID of `n`'s parent node from the Map `s`. Another function `getIdentifierOf(n)` returns the new identifier `i`. If `n` represents a function or a global variable, `i` is a hash value (see Section 4.4.3, Paragraph "Internal Structure of CIDs"). Otherwise, `i` is a relative number that is encoded as hexadecimal number. This way, we can build up the CID incrementally.

In case of programs written in Java, we exploit the Eclipse JDT to traverse the AST via the visitor pattern rapidly in order to calculate the CIDs. Of course, there are similar tools available for other programming languages such as the Eclipse PHP Development Tools (PDT) [63, 68].

4.4.4 Processing Data Generated by Instrumentation Code

Of course, solely adding instrumentation code is not enough. We also need a way to get the information which parts of a program have been traversed. As we have seen in Paragraph "Discussing Existing Ways of Code Instrumentation" in the previous subsection, relevant data about program execution can be collected for example in variables, by creating branch traces or some other kind of log file. Basically, we could imagine to trace the program execution in a similar technical way. The main question however is always, how can we access the data collected in variables or in any similar way, shape or form?

As the tests may be of any kind (unit tests, integration tests, UI/web tests), we need a mechanism that is universally able to provide the data gathered when executing instrumentation code. Especially web tests impose restrictions as the data are only available within a web browser in the first place. It is not possible to directly persist data on the local hard drive due to security reasons. Nevertheless, we need a possibility that allows us to transfer the data efficiently. AJAX is a pervasive concept to transfer data between client and server. For this reason, we have initially investigated XMLHttpRequests to pass the data to an own *http-logging server module* whose main task is to persist the data collected by the instrumentation code. The main disadvantage of XMLHttpRequests however is that the connection has to be re-established every time when data are about to be transmitted to the logging server. This has always required extra time while deteriorating the throughput. Besides, it has been difficult to notify the client automatically when it should interrupt, stop or delay collecting instrumentation data. This has for example been necessary when the application under test had to perform some initial setup steps that did not belong to the actual test case. For these reasons we have investigated a solution employing the WebSocket protocol. It is described in the official standard [148] in the following way:

> "The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g., using XMLHttpRequest or `<iframe>`s and long polling)." *The WebSocket protocol, Version 13* [148]

Especially the possibility to have a two-way communication that remains open is a big advantage and meets our requirements. Consequently, we have decided to realize the data processing with the aid of the Web Socket protocol in combination with a logging server. More details about the logging server and its functionality follow in Section 4.7.

4.4.5 Syntactic and Semantic Requirements on Instrumenting Source Code

Especially in the area of code coverage, the problem of relating instrumentation code with source code arises. In existing tools like for example CodeCover [44], instrumentation code is associated with a statement via its position in the source code [124]. There, the rule is just to insert the instrumentation before the statement it represents. But of course, adding instrumentation code into source code must obey the usual syntactical rules of the source language. Moreover, the instrumentation code is always semantically related with a piece of source code. This semantic relation depends usually on the exact location of the instrumentation code and must be preserved after any kind of code optimization and even after transcompilation. That is, in the target code, the semantic relation must still hold.

Based on these insights, we have investigated two different approaches to instrument code that incorporate the special demands of transcompilers. The first approach follows in a large part the standard approach of instrumenting source code: We add instrumentation code next to the corresponding code in the source language and take special precautions to not break any semantic relations between instrumentation and source code. Details follow in the next Section 4.5.

The second approach completely avoids adding instrumentation code into the source code. Instead, the approach directly adds instrumentation code in the target code. This way, we do not risk to break during transcompilation any semantic relations between source code and its corresponding instrumentation code. Unfortunately, this freedom is not for free and has its own disadvantages. We explain the entire approach in Section 4.6.

4.5 Compiler-Independent Instrumentation

Our compiler-independent approach injects code identifiers as instrumentation code that will be transcompiled to the target programming language just as the normal source code. In the target language, we log which identifiers are executed by a test. The entire instrumentation process requires three steps. It aims at injecting information into the source code in such a way that it is still usable after transcompilation in order to collect information about which test executes which parts of the target code. This information is used to select tests for re-execution and to identify code changes that might be responsible for a test failure.

Assumptions

Our approach is based on the following assumptions: First, we presume that the toolkit used to compile an application to a target language works correctly. In the same sense, we also assume that the compiler maintains the logic of the

code. That is, a specific sequence of function calls will be maintained by the transcompiler. This implies that a common function call declared in front of a certain statement s will still be executed before s when it has been compiled to another language. More generally, function calls used as instrumentation code are still usable for determining code coverage after transcompilation. In the next subsections, we explain the three steps required for our instrumentation method in detail.

Step 1 – Code Instrumentation

We start by determining CIDs for all code entities by traversing the AST in the manner explained in Section 4.4.3. Please remind that the CIDs consist of simple sequences of characters. We inject these CIDs into the program source code as instrumentation code. This is done in a pre-processing step before transcompilation. In order to avoid polluting the local working copy with instrumentation code, we use a copy of the source code of P. During transcompilation, both the source code and the instrumentation code are transferred into the target language. When the transcompiler obfuscates the target code, it will look completely different than before except for the CIDs: As they are normal sequences of characters (in Java: Strings), they are maintained during transcompilation and show up unaltered in the target code.

In contrast to other techniques (e.g. CodeCover, see Section 4.4.3), we cannot use variables in instrumentation code because they could be obfuscated during transcompilation. In this case, it could be hard to identify the variables and thus, it could be impossible to retrieve their values. Besides, our injected instrumentation code should be usable regardless of the kind of application. Our solution for this problem looks as follows: Instrumentation code always consists of two parts: one (or several concatenated) CID(s) and a simple function call that takes the CID(s) as argument. Figure 4.18 illustrates this principle. The instrumentation code – i.e. the corresponding function call `instrument(...)` – is highlighted with gray color. It refers to an interface of a separate module. This module will always be injected during source code instrumentation. More details on its functionality follow in the explanation of the next step. In most languages, function calls have to be inserted in the code as part of a statement. So as a *rule of thumb*, we inject instrumentation code as statement in front of the code entity they identify.

However, some programming languages impose syntactical restrictions which prevent this straightforward solution. Depending on the source language in use, we have to consider individual exceptional cases. To provide a better understanding, we discuss different exceptional cases for Java:

Global Variables, Fields, Classes: Global variables are of great interest for us for multiple reasons. For the test selection and fault localization, we need to know whether the type of a global variable has been modified or which global variables have been added or removed. For calculating the code coverage

```
1  function f() {
2    instrument(<unique id representing int x = a * b;>);
3    int x = a * b;
4    instrument(<unique id representing g(x)>);
5    g(x);
6  }
7
8  function g(number x) {
9    // ...
10 }
```

Figure 4.18: Rule of thumb: Function calls as instrumentation code in front of code entities.

of UI/web tests (see Chapter 7), we want to reveal whether global variables are never used or whether they are never initialized. Consequently, we have to add instrumentation code for these syntactical elements. Nevertheless, in languages like Java, injecting instrumentation code outside methods results in syntax errors. (Please remind that instrumentation code consists of simple function calls.) For this reason, we exploit the language's rules which define when the global variables (= Java fields) will be initialized. In Java, field initialization is performed by the Java compiler within the (default) constructor. Consequently, we insert instrumentation code within the constructor after possibly present (super-) constructor calls. Figure 4.19 gives an example (see `instrument(<unique id representing i>)`). If the class misses a constructor, we insert a default constructor manually and borrow the visibility modifier declared by the class.

Considering class variables, an instance of the class is not stringently required. So we add instrumentation code in the (default) constructor as usual and additionally, we insert instrumentation code in a static initializer that might have been inserted into the AST if it did not exist yet (see `instrument(<unique id representing j>)` and the static initializer in Figure 4.19).

With regard to classes, we follow the same strategy and add instrumentation code in the constructor in order to keep track of which classes are executed by a test case (see `instrument(<unique class-id>)`).

Functions, Initializers, Blocks: In order to track that a test executes a function, an initializer, or a(n) (anonymous) block, we inject instrumentation code at the top of the body (see the `function-id` in Figure 4.20). For the statement in `m()`, the general rule of thumb applies. That is, the instrumentation code is injected in front of the corresponding statement (see `return 1` in Figure 4.20). So, even if an exception in thrown by the web application, it is ensured that the corresponding instrumentation code is executed before.

```
1  class C {
2    private int i;
3    private static int j;
4
5    C() {
6      instrument(<unique class-id>);
7      instrument(<unique id representing j>);
8      instrument(<unique id representing i>);
9    }
10
11   static {
12     instrument(<unique id representing j>);
13   }
14 }
```

Figure 4.19: Field instrumentation, class variables instrumentation, and class instrumentation.

```
1  public int m() {
2    instrument(<unique function-id representing m()>);
3    instrument(<unique statement-id>);
4    return 1;
5  }
```

Figure 4.20: Standard and function instrumentation.

Step 2 – Test Execution and CID Logging

The objective of this step is to gather data about which test executes which source code. In general, our approach requires tests to be executed on the instrumented, transcompiled application in order to find out which parts of the code/ which CIDs are traversed by tests. Then, the collected data have to be passed to the tool that performs the test selection and the mapping on the source language in order to assist developers in localizing faults. We refer to this tool as *analysis tool*. So, after code instrumentation has been finished, we transcompile the application and run the tests. During test execution, instrumentation code is executed just as other application code in the target language. Please note that the instrumentation code still consists of (transcompiled) function calls that take a sequence of characters as argument.

If the code under test belongs to a desktop application or to server-side code of a web application, the callee (that is part of a module M) directly connects to a built-in database of our analysis tool and inserts all the CIDs obtained as arguments into this database.

If the code under instrumentation belongs to a web application or a mobile applications, our analysis tool acts as logging server. The callee sends the CIDs via the WebSocket protocol to the logging server, which inserts them into the

database. Figure 4.21 shows the relevant code for GWT-based web applications. Here, the callee (`instrument(String nodeIdentifier)`, line 1) passes CIDs (see line 3) to another piece of code that implements the WebSocket API. Having a closer look at the whole method `instrument(String nodeIdentifier)`, it becomes evident that this is no regular Java code (please note the keyword `native` and the special syntax that resembles Java comments (`/*- ... -*/`)). In fact, we exploit that GWT allows us via the JavaScript Native Interface (JSNI) [97] to write native JavaScript code within regular Java code (see lines 2-4). This enables us to write Java instrumentation code consisting of simple method calls to native JavaScript code. Here, the native JavaScript function `sendToLoggingServer(nodeIdentifier)` refers to code that implements the WebSocket API (see lines 10-17). It is injected in the web application via the GWT ScriptInjector [114] which adds JavaScript code to the Document Object Model.

```
1  public native void instrument(String nodeIdentifier) /*-{
2    if(nodeIdentifier !== '') {
3      \$wnd.sendToLoggingServer(nodeIdentifier);
4    }
5  }-*/;
6
7  // Entry point for GWT applications, similar to the main()-method in
       regular Java
8  public void onModuleLoad() {
9    ScriptInjector.fromString("
10     var loggingServer = new WebSocket('ws://localhost:8090/
           loggingServer');
11     loggingServer.onopen = function() {};
12     loggingServer.onmessage = function(evt) { ... };
13     loggingServer.onclose = function() { ... };
14     function sendToLoggingServer(nodeIdentifier) {
15       ...
16       loggingServer.send('testName='+testCaseName+'&id='+
             nodeIdentifier);
17     }
18   ");
19 }
```

Figure 4.21: Sending CIDs to the analysis tool via WebSockets.

Finally, the function `sendToLoggingServer(nodeIdentifier)` (see line 14) passes the CIDs to the logging server which in turn persists them in a database. Figure 4.22 illustrates the code of the logging server.

The database holds information on which CIDs have been traversed by a specific test. So all the test traces are persisted in a database. There, we also store how often a CID has been executed by a specific test. So, if a CID is already present in the database, a counter is incremented instead of adding a new database entry. This enables us to easily provide further statistics on the test thoroughness a piece of code is tested with. We call this the *execution frequency*

```
1 @OnWebSocketConnect
2 public void onConnect(Session session) { ... }
3
4 @OnWebSocketMessage
5 public void onMessage(String msg) {
6   writeToDatabase(msg);
7 }
```

Figure 4.22: Persisting CIDs passed to the logging server.

of a code entity. (This is especially important in Chapter 6). Optionally, we are able to assign the execution frequency to individual test cases.

Step 3 – Selecting Test Cases and Localizing Emerging Faults in the Source Code

We want to select test cases that execute code changes. As a result of step 2, we know which CIDs have been executed by the test cases. But the code changes are still unknown. We determine them with the aid of our method for calculating changes made in the source code (see Section 4.3.2), which is based on a comparison of the nodes in the EJIG of P and P'. As output, we obtain the code changes and lots of meta data that describe these changes. We call them *code change meta data*. They encompass the node in the AST that corresponds to the code change, as well as the affected source file, the line number, the kind of change (added, modified, or removed code), and how the code looked like in the previous program version P. This enables us to mark the code changes within an IDE.

All the code change meta data are available in our database. Now, we have to decide which test cases cover the code changes. To achieve this, we join the table containing information on code change meta data and the table containing data about test traces. This way, we can depict which test case is affected by a code change and conversely, which code change(s) might be the reason if a test case fails. This highly enhances the fault localization and eases the debugging process. Nevertheless, before joining the databases, we need some additional information.

For the test selection, we need to know which code change corresponds to which CID in the test traces. For this purpose, we recalculate the CIDs for the old version P and link the code changes in P' to the CIDs obtained from the test execution of P. We base the linking process on the set of dangerous edges obtained during the comparison of the EJIGs. Similarly to the fault localization presented in Section 4.3.3, several different cases have to be considered. Usually, we can borrow the CID from the node in P that corresponds to the code change in P'. That is, if the target node of a dangerous edge in P' has been modified (this is the Regular Case in Section 4.3.3), we use the CID obtained for the target node in P. This is also true if a node has been removed at the end of

a method in P'. We borrow the CID c from the target node in P. With this information, we are able to search in the test traces for c. All tests that have executed c in P are selected for re-execution. Please note that during fault localization, we need to highlight the *start node* of the dangerous edge in the source code as the modified node (i.e. the target node) does not exist any more. So the highlighted node in the Eclipse view might differ from the node that has been changed and whose CID is used determine the test selection. Of course, there are also special cases in which we have to use the CID belonging to the start node in P. For example, this applies to exit nodes (see Section 4.3.3).

4.6 Compiler-Dependent Instrumentation

The basic idea of the compiler-dependent instrumentation approach is to avoid inserting CIDs as instrumentation code in the source programming language. Instead, we keep the CIDs apart from the source code in either an XML file or a database. In the end of course, we still need to know which tests execute which code entities. The CIDs have to show up in the target programming language so that tests can traverse them. So, we have to add the CIDs as instrumentation code in the target language. The core challenge for a successful implementation of our technique is the question on how the mapping of CIDs to the target code generated by a transcompiler could be accomplished. Some transcompilers support source maps (see Section 3.1). But even if not, each transcompiler provides at least some functionality to parse and transfer the code from a source language into a target language. This is our starting point for injecting instrumentation code.

In general, the main steps of our approach to map code identifiers to target code are the same for all transcompilers. But of course, each transcompiler could have some peculiarities depending on its general abilities and its implementation. Besides, programming languages usually have different syntactical rules. Thus, it is difficult to define a universal, for many programming languages valid solution for adding instrumentation code to the source code of the target language. This is why the details of our approach will usually differ and why we expect to have always some exceptional cases per source programming language and its corresponding transcompiler.

Our compiler-dependent approach consists of four main steps. Three out of four of these steps are similar to the compiler-independent approach. However, as the transcompiler does not automatically add instrumentation code any longer while translating the code of the source language into the target language, we have to take care ourselves of the CIDs not getting separated from the code entity they identify. This mapping is part of the additional step. Again, we explain the steps of our approach in general. In each step, we additionally explain more specific details by taking the example of GWT's transcompiler.

Assumptions

Our approach is based on the following basic assumptions: We presume that the code of the transcompiler is open source as we need to do some adaptations. This implies that we can build a new version of the transcompiler ourselves.

Step 1 – Code Instrumentation

The first step is completely independent from a specific transcompiler. The only requirement is that an abstract syntax tree is available. We start by determining CIDs for all code entities in the way explained in Section 4.4.3 by traversing the AST representing the original program version P. In addition, we collect data on the line number of the corresponding code entity and on the globally qualified file name. For brevity, we refer to the combination of these data and the CIDs as *source code meta data*. Usually, the transcompiler is an external program. To provide the source code meta data for all kinds of transcompilers, we would have to create a universal interface. This is difficult as the transcompiler might differ completely in their architecture. So, in order to ensure that the data are available for all transcompilers, we persist the data either in an XML file or in a database.

Step 2 – Mapping Code Identifiers to Target Code

We exploit the transcompiler's abilities and some basics of source maps (see Section 3.1) to take on the task of injecting the CIDs as instrumentation code in the target code. Especially the name of the original source file as well as the line number of the corresponding code entity within the source file are indispensable to make our approach work.

With respect to GWT, it has turned out during an extensive analysis of its open source code [106, 116] that at its heart, GWT also uses the AST provided by the Eclipse JDT to represent a Java application. Based on this AST, GWT maps every single node to a node of another AST representing the JavaScript application. So, there exists a mapping between the nodes of the Java AST and the nodes of the JavaScript AST. Besides, GWT collects source code information for each node representing Java code. We refer to these data as *GWT meta data*. They include the globally qualified class name as well as the line number[2]. So in the end, GWT gathers and maintains essentially the same data as we do. We utilize this for our purposes.

Now, the main question is how to make the transcompiler add the CIDs into the target code. Here, we had to keep in mind that the transcompiler is still undergoing a process of extensions and improvements and that there are new GWT versions to appear. This is of course not specific to GWT but also applies to other transcompilers. Consequently, we have tried to find an easy solution that reduces the necessary changes made in the GWT compiler to

[2]Details can be found in `com.google.gwt.dev.jjs.SourceOrigin.java` of the `dev/core` module in the GWT Git repository [117].

an absolute minimum. However, the GWT compiler does not provide special hooks for this purpose. At the end of the day, we have managed to implement our approach with three different kinds of changes.

Our first adaptation of the GWT compiler extends the valid compiler options [95] by a new parameter that accepts either a path pointing to an XML file or data to connect to a database. We refer to this file/database as *node identifiers data source*. It contains all the CIDs together with the line number of the code entity that is represented by a CID and the globally qualified class name of the Java class a CID belongs to. If several CIDs are connected to code entities that occur in the same line of code, the CIDs are concatenated in a special way.

Furthermore, we combine the GWT meta data and our own source code meta data that contain the CIDs. For this purpose, we expand the GWT meta data by an own object that encapsulates the corresponding CIDs. In order to find the CIDs that fit the data gathered by the GWT compiler, we match the file names and line numbers provided in the GWT meta data against our source code meta data. This way, the GWT compiler knows the CID for each source code entity that has to be transcompiled.

Another required change of the transcompiler concerns the output of the CIDs in the JavaScript code. For this, we have adapted the GWT compiler in our initial version in such a way that the CIDs are arguments of `log`-statements. As a result, as soon as a test case traverses the CIDs, they are logged to the console. Another, vastly better possibility is to use WebSockets just as already explained in Section 4.4.4 before. So, we directly inject the same code in the JavaScript code of the web application that we have illustrated in the lines 10-17 in Figure 4.21. But in either case, adapting the GWT compiler involves of course decisions at which position the CIDs should be added in the target language. For JavaScript as target language of GWT, similar rules apply as already described in step 1 of Section 4.5. In particular, we reapply our *rule of thumb* according to which we inject the CIDs as instrumentation code in front of the code entity they identify. Again, there are some special cases that have to be incorporated:

Omitted brackets: Notoriously, the GWT compiler extremely optimizes the JavaScript code. This includes the exclusion of brackets where they are not necessary. This affects for example if- or loop-statements. Naturally, adding instrumentation code within this syntactic simplifications changes the semantics. Consequently, we have to manually add brackets.

Constants: Another optimization affects constant propagation (see Paragraph "Code Optimization" in Section 3.1) which is rather delicate. For example, the GWT compiler replaces Java constants (fields declared as `static final`) by their value. So, the CID might get separated from the code entity it identifies and consequently, assigning a different value to the constant in a new program version P' might remain undetected. Solving this difficulty requires a deeper

intervention in the transcompiler code. But the only remaining alternative would be to turn off code optimization in the transcompiler.

Please note that in the compiler-independent approach, this problem does not arise as the constant initialization is performed within the constructor (see step 1 of Section 4.5). So the transcompiler always transfers the CID representing the constant into the target language.

Step 3 – Test Execution and CID Logging

In general, this step does not differ from step 2 in our compiler-independent approach (see Section 4.5). The objective is to gather data which test executes which source code. To this end, we run UI/web tests that produce test traces that are persisted in a file or, in later versions of our approach, in a database. We can rely on the same WebSocket-based interface as we did in the compiler-independent approach.

Step 4 – Selecting Test Cases and Localizing Emerging Faults in the Source Code

Just as the previous step, selecting test cases and localizing emerging faults in the source code follows the same procedure as described in step 3 in Section 4.5. So we want to refer the reader to this section to recall the details.

4.7 Prototype Tool Implementation: Compiler-Dependent Approach

We have implemented a prototype of the compiler-dependent approach as an Eclipse plug-in called GWTTestSelection. As the name indicates, the prototype has been tailored for transcompiled cross-platform applications based on GWT. The plug-in consists of several modules: One of them calculates the EJIG with the aid of the Eclipse JDT. Another one performs the calculation of code changes (see Section 4.3.2). The tasks of the other modules comprise the calculation of the CIDs (see Section 4.4.3), their mapping to JavaScript (see "Step 2 – Mapping Code Identifiers to Target Code" in Section 4.6) and the calculation of code entities that are suspected of being responsible for a test selection (see "Step 4 – Selecting Test Cases and Localizing Emerging Faults in the Source Code" in Section 4.6). The results are available in a separate analysis report file and in Eclipse's code editor. In our initial prototype tool, the logging server has still been implemented using AJAX. Details about the improved version that relies on WebSockets follow in Section 5.5.

GWTTestSelection expects the existence of an Eclipse project in the current Eclipse workspace. It is the current program version P. For this version, GWTTestSelection calculates CIDs. Besides, during test execution, it creates test traces. So initially, P is the program under test. Beginning with additional code enhancements or bug fixes, P becomes the new version P'. In

order to be able to decide which tests have to be re-executed, GWTTestSelection requires a previous version P for the comparison. This version can be provided in two different ways. First, P can exist locally as a additional (renamed) Eclipse-Project in the workspace. The Eclipse plug-in offers a wizard to choose the appropriate projects. Second, our Eclipse plug-in includes functionality to checkout P from a version control system and to import this preceding project version temporarily. In order to avoid name clashes, our plug-in appends a time stamp.

Figure 4.23 shows a screenshot of our initial prototype tool. The Eclipse code editor in the upper region displays a class of a modified program version and the localized changes. The lower part contains the control panel of our Eclipse plug-in. It enables the creation of the CIDs for an initial or a changed program version and it is able to launch our adapted GWT compiler to create an instrumented version of the web application. At the time of the evaluation, the instrumentation level has been set to instrument all the statements and even some kinds of expressions (e.g. conditional expressions). In later versions, this is adaptable according to the users needs. Additionally, the control panel lets the user start calculations to localize changes and to determine affected web tests. Besides, the control panel offers some additional analysis options such as investigating special methods. Here, we might think of methods that are only reachable via dependency injection. If our technique would be applied for example in web applications created with JavaServer Faces we could analyze methods that are only called from XHTML-templates that are used to create the user interface of the web application.

To ensure that the plug-in works as expected, our Eclipse plug-in offers an additional feature. We can turn on a unit testing mode in which the analysis results are compared to our expectations. To this end, we define for each class in P and/or P' how the expected analysis result should look like and we insert differences into the source code (see comments in Figure 4.23).

In subsequent chapters, we show extended versions of the plug-in.

4.8 Evaluation: Compiler-Dependent Approach

We have evaluated our Eclipse plug-in GWTTestSelection with five research questions in mind:

RQ1 Is our EJIG suitable to model GWT-based web applications in such a way to localize code changes correctly?

RQ2 Does our approach select all the tests for re-execution that are indeed affected by a code change? That is, is the test selection safe after tracing the mapping of Java source code to JavaScript code?

RQ3 Have web tests been selected unnecessarily?

RQ4 How efficiently does our tool work?

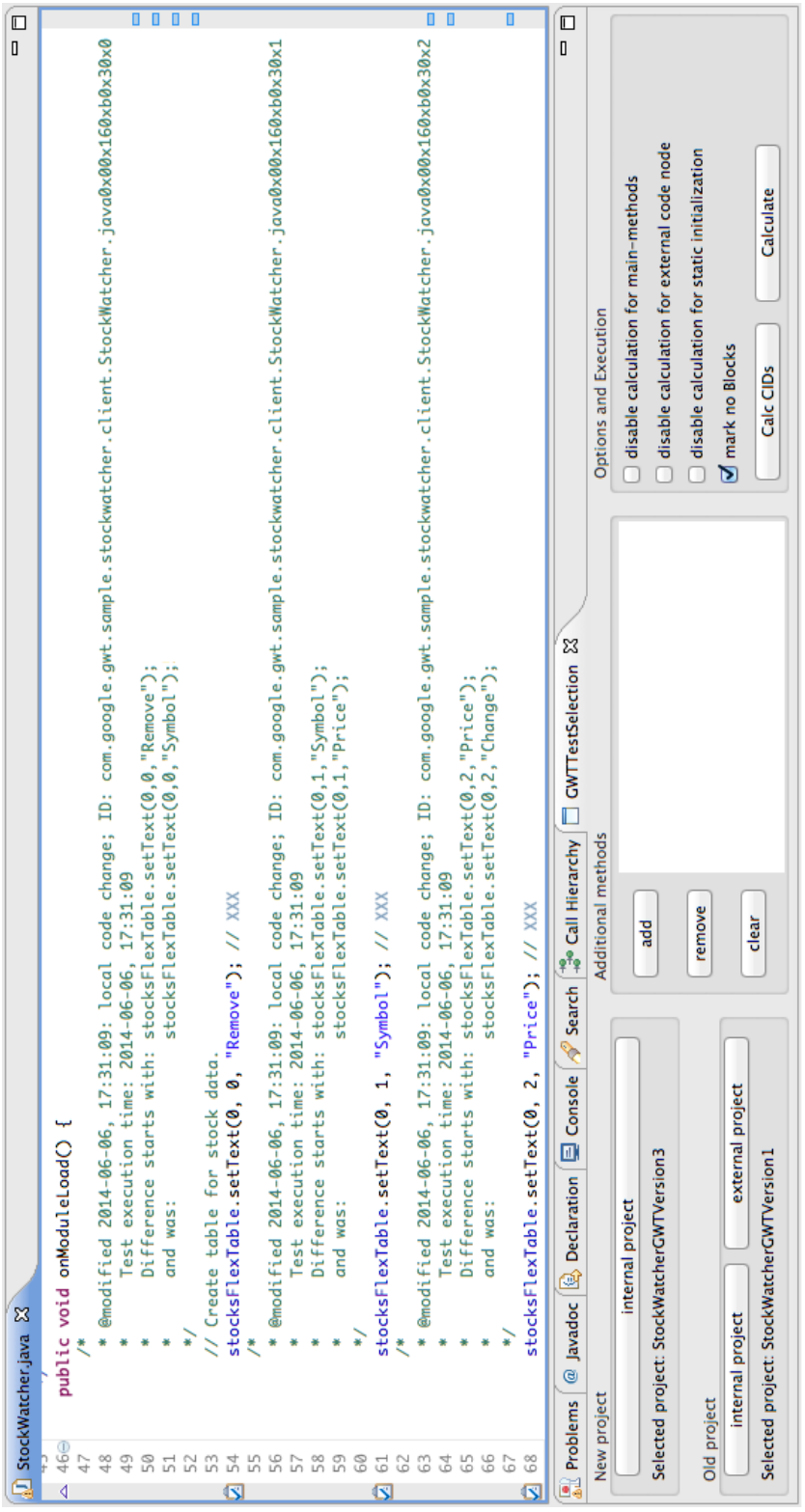


Figure 4.23: Fault localization.

4.8.1 Software under Evaluation

For our evaluation, we use two Java-based web applications that have been transcompiled to JavaScript by using the GWT transcompiler. These are StockWatcher and Hupa. As already mentioned in Section 3.1, StockWatcher is a small example taken from the GWT tutorials [100] that displays self-defined stocks and their value fluctuations. The tutorial introduces different versions of StockWatcher. To test our comparison algorithm, our change localization, and our test selection thoroughly, we use the version that generates and maintains all the data on client side instead of generating the data on server side. This version consists of almost 300 non-empty lines of code in two classes. Although the code basis of is very small, it has been suitable to investigate the functionality of GWTTestSelection towards correctness.

In order to gain more confidence and to show the applicability of our our compiler-dependent approach in a real web application, we have additionally chosen Hupa [144] as software under test in our study. Hupa is a mid-sized open source GWT-based mail client that provides all the basic functionality of modern mail clients. This includes receiving, displaying, sending and organizing mails. For retrieving mails, Hupa uses the IMAP protocol. We have checked out the source code from the public repository in the revision 1580208 which was the most recent version at the time of the investigation. It consists of approximately 40.000 non-empty lines of code in 484 classes and interfaces. Initially, we have planned to compare this version with older revisions from the repository. Unfortunately, these revisions always contain a large number of changes. This is contrary to the usual way of doing small increments that are regression tested afterwards. Besides, several of the older revisions are corrupted due to merge conflicts. For this reason, we have created own versions introducing typical changes.

4.8.2 Experimental Setup

For the evaluation of StockWatcher, we need several program versions and a set of test cases. To this end, we have created 11 versions that contain different kinds of changes, including behavioral, logical, and optical changes to simulate modifications as they occur in a standard development process. For example, we have altered case distinctions, modified conditions in loops, or injected faults in the calculation of the relative performance of the stocks. Optical changes have been possible as StockWatcher makes no use of GWT's UIBinder [99]. So, the entire layout is defined in plain Java. To simulate optical modifications, we have swapped the columns in the UI and we have altered CSS class names used to layout the web application.

Unfortunately, there are no web tests available for StockWatcher. For this reason, we have created web tests ourselves. To each of the versions, we have applied 12 web tests representing user stories such as adding and removing stocks as well as handling illegal inputs.

The evaluation of Hupa is based on four versions, emerged from the original

revision. As already explained, previous revisions of HUPA were buggy. So we had to re-create these revisions by extracting some of the original changes from the HUPA repository, ignoring the merge conflicts. This way, we have tried to guarantee real conditions. In the end, each of the versions contains modifications concerning the behavior, the code structure, or the client-side logic. For example, we have restored a previous algorithm used to attach files to a mail. To simulate changes in the structure, we have refactored and renamed the code. As opposed to our previous web application under evaluation, the mail client employs the UIBinder to define the user interface. So, we did not move widgets on the screen. However, HUPA uses CSS for some widgets. Here, we have made modifications in CSS names.

As the developers of HUPA do not provide any web tests, we have created 40 simple web tests with SELENIUM. The tests implement user stories such as sending, responding, or deleting a mail.

All measurement in this evaluation have been performed on an Intel Core i5 at 2.4 GHz with 8 GB RAM. The Eclipse settings have been left untouched.

4.8.3 Threats to Validity

External Threats to Validity

The GWT compiler has several compiler options (`OBFUSCATED`, `PRETTY`, and `DETAILED`) [112] that determine how the output should look like. We have to ensure that our tool works independently from the code style applied by the GWT compiler. Technically, we have encapsulated and attached the instrumentation code to the code entity that has to be transcompiled. The transcompiler extracts the instrumentation code not before the obfuscation has been finished. So at least, to the best of our knowledge, we can be sure that the instrumentation code and the corresponding code entities are never separated. For each transcompiled code entity, a piece of instrumentation code is added in the target code, notwithstanding code obfuscation. In order to confirm this in practice, we have conducted our evaluations twice. First, we have used the compiler option `PRETTY` to obtain code that is still similar to the original Java code. Afterwards we have employed standard compiler options (`OBFUSCATED`). In both test runs, the results concerning fault localization and regression test selection have been identical. So, obfuscation had no effects. Of course, we cannot conclude from this observation that there are no bugs in our code. But the test did not reveal any faults so far.

Fowler [86] discusses several problems that might lead to unpredictable nondeterministic test behavior: In particular, asynchronous calls with the aid of AJAX might lead to missing or wrong data that corrupt the state of an application and thus the whole test. Fowler also points out possible solutions such as callbacks. According to him, pure wait-commands that always wait for a fixed time are not enough because sometimes, the predefined wait time is not enough or, conversely, the result is already available but the test execution cannot proceed as the predefined wait time is not up yet. We want to mention

that of course, developers of test tools are aware of these difficulties. Accordingly, test tools provide functionality to cope with asynchronous client-server interactions to avoid faults as a result of e.g. pending server responses. For example, SELENIUM provides special `waitFor()`-commands [251] that succeed as soon as a condition becomes true. Otherwise, after a predefined timeout, the command returns false and the test fails.

Nevertheless, Fowler [86] stresses that there exist situations in which this kind of dealing with non-determinism is still not enough. He points out that whenever a call c does not expect a response, commands like `waitFor()` do not help any more. Here, additional tests are necessary to ensure that the call c worked as expected.

Another kind of problem emerges according to Fowler [86] from resource leaks. As examples, he mentions memory leaks and database connections. However, we consider this problem to be less difficult to solve. As a possible solution, virtualization (e.g. with Docker container [59]) could be used. For example, each test could run in an own container with an extra database that is based on a predefined image.

Referring to Sprenkle et al. [263], another potential error source concerns web page faults. The authors divide this kind of faults into "form faults", "appearance faults", and "link faults" [263, page 225]. With regard to our approach, naturally, everything is fine if the user interface and its entire appearance is completely written in the source language. In this case, every single change would be detected by our technique. In practice however, plain HTML and CSS is frequently used to define the layout of a web application. Besides, there are also frameworks that use a different way to define the layout. GWT for example provides two possibilities of creating the user interface of a web application. The first one relies almost completely on Java. Solely for the layout of widgets, GWT applies CSS [107]. According to this, each and every modification concerning the functionality or the structural layout can be detected by our comparison algorithm. This includes even changes in the layout of a widget due to modifications in the assignment of CSS class names or CSS ids. Solely changes within the CSS style sheet cannot be detected. They need to be analyzed with standard, DIFF-based tools. But this is no problem as styles defined with CSS usually have no effect on the behavior of web tests. This is also true for changes in e.g. margins of UI-widgets. Although this affects the position of a widget, it is immaterial as modern test tools refer to widgets within tests via ids or name attributes. Thus, we do not consider changes within style sheets.

The second possibility to define the user interface in GWT is based on the *UIBinder* framework [99], a declarative way of programming user interfaces. Instead of using Java to define the layout and the position of each widget, XML and HTML is applied. The widgets are bound to a Java owner class. So, the layout and the position of widgets are defined using the document object model. However, the Java owner class is still responsible for the entire processing of user interactions in the client as the UIBinder does not offer loops, conditionals, or if statements [99]; and it cannot display data dynamically.

Therefore, nothing changes in compiling Java code to JavaScript. When using UIBinder, modifications in the logic can still be found with our algorithm. In addition, modifications or the removal of widgets can also be detected. As opposed to this, because all the layout definitions are outsourced in a HTML file, modifications in the page structure of the web application or broken hyperlinks cannot be detected with our algorithm. But as already explained above, a shifted widget is no problem when identifying widgets by means of ids or name attributes. Besides, we can detect changes in HTML easily with DIFF-based tools. The same is applicable to issues with CSS as explained before. For this reason, we do not pay attention to changes within HTML files.

Internal Threats to Validity

In order to judge the ability to identify code changes correctly, it is not enough to apply our approach on two web applications. This is because we could miss special cases in which our tool GWTTESTSELECTION would actually fail. But as these special cases do not show up by accident in the software under evaluation, they remain undetected. For this reason, we have created many pairs of small example Eclipse projects whose modified version P' contains many different kinds of syntactical changes. For each pair, we created unit tests by defining exactly where a code change has been made. We additionally define the CID that identifies the code modification, and how the code looked like before if this is possible.

Threats to Construct Validity

We assume that the compiler works correctly when transferring the application's source code into the target language. Besides, we expect that the code's internal logic is fully retained. This is important as instrumentation code is always implicitly connected to a code entity. Following the rule of thumb explained in Section 4.6, "Step 2 – Mapping Code Identifiers to Target Code", instrumentation code has always to be executed right before the corresponding code entity. Potential code optimization performed by the compiler must not destroy this implicit connection in order to be able to determine the code coverage. However, we can trust in this expectation in the same way as other approaches do when adding instrumentation code in source code, in bytecode, or binaries respectively. Moreover, in case of GWT, compiler optimization can be turned off in case of doubt [108]. Other transcompilers like Haxe offer similar settings [128, 129].

4.8.4 Results

In the next subsections, we present the results of our research questions which we obtained during our evaluations.

RQ1: EJIG – Suitability to Model GWT-Based Web Applications in Such a Way to Localize Changes Correctly

When analyzing StockWatcher and Hupa, our tool has identified all code modifications correctly. It has provided information about the CID and a hint how the code has been modified. When turning on the unit testing mode, GWTTestSelection additionally inserts this information as comments in front of these lines of code that contain the code modification(s) (see Figure 4.23). If code has been removed, the comment is attached after the previous line of code. When no such line exists, the change is displayed after the beginning of the next enclosing block.

The EJIGs representing P and P' have succeeded in modeling all parts of the program versions that are necessary to identify the changes introduced in P'. So, when adding our experience made with the small test projects mentioned in Paragraph "Internal Threats to Validity" of Section 4.8.3, the EJIG is suitable to model GWT-based web applications.

RQ2: Is the Test Selection Safe?

We have thoroughly analyzed which code modifications should affect which web test case. Starting from this analysis, we have a set of test cases S expected to be selected for re-execution. Afterwards, GWTTestSelection has determined the test selection. We have compared the expected set of test cases S with the results calculated by our tool. For both StockWatcher and Hupa, GWTTestSelection did not miss a web test. Moreover, GWTTestSelection has listed for each web test due to which code change(s) it has been selected for re-execution. Consequently, the test selection has been safe after tracing the mapping of Java source code to JavaScript code. That is, GWTTestSelection selects at least all the tests affected by a code change for re-execution.

RQ3: Unnecessarily Selected Web Tests

Apart from a safe test selection, another goal has been to reduce the total number of selected test cases to a minimum. That is, we desire the number of unnecessarily selected tests to be as low as possible in order to obtain a preferably low test case selection. The results of the test selection are illustrated in Figure 4.24. On the left, it shows the overall percentage of tests that need to be rerun due to changes made in the code of StockWatcher and Hupa. On the right, it displays the percentage of tests actually selected by our tool. In case of StockWatcher, we have expected that 80% of the tests should be selected for re-execution. This rather high value is due to the timer employed in the application to update and reload the stock prices in a fixed time interval. Reloading and updating the data requires the traversal of a major part of the application and therefore, some of our code modifications affected many web

tests. When considering the tests selected by GWTTestSelection, 89% of the tests have been selected without missing one of the expected tests.

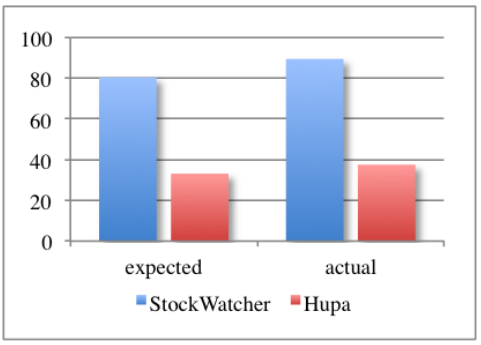


Figure 4.24: Percentage of expected and actually selected tests in StockWatcher and in Hupa.

Wen applying GWTTestSelection to Hupa, it has turned out that the total number of tests selected for retest is considerably lower (see Figure 4.24). GWTTestSelection has selected 37% of the tests for retest. Actually, we have expected 33% of the tests to be re-executed.

Earlier studies (e.g. [126]) have also observed such variations in the number of tests selected for retest. According to them [126], this is due to the kind of modifications. If there are only small changes, it is more likely that only a few tests are selected for retesting. This is consistent with our observations we have made in the evaluations. Whenever we have replaced big parts of the code, the number of selected tests has increased.

RQ4: Efficiency of Our Tool

The application of our tool to a web application like StockWatcher provides not much information on efficiency as the EJIGs are small and fast to calculate. Nevertheless, we have combined all the web tests to a test suite for re-execution with the Selenium IDE. As the web application uses a timer to update the stocks in predefined intervals, the total time required for executing the web tests depends heavily on the time interval size. But even if we omit those tests that depend on the timer, the test suite has taken more time than our tool. Beyond that, if we restart our tool to repeat an analysis, we observe a significant time reduction by approximately more than a half. This is because the EJIG(s) have already been calculated and thus, the data are still available in memory. Even if we repeat the analysis using one new version and a version whose EJIG has already been calculated, we can still observe a time reduction compared to the time consumption that is necessary to calculate two EJIGs for the first time.

In order to assess the efficiency of GWTTestSelection, Hupa is more suitable as the EJIGs for both the old and the changed program version are

large. So, the comparison is more complex and the analysis of test traces is more time-consuming. Again, we have used the SELENIUM IDE to run our tests. Despite the huge amount of logged CIDs, our tool has finished the analysis after approximately 90 seconds, whereas the test suite has taken twice the time. So, our tool works efficiently. Apart from the pure time reduction, our tool has the additional benefit that it is able to localize faults. Naturally, the traditional retest-all approach does not support this.

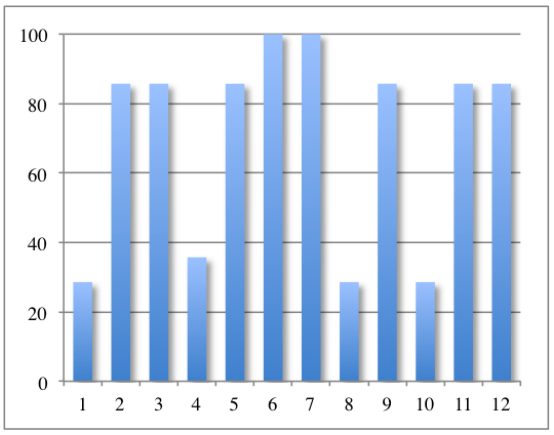


Figure 4.25: Code changes responsible for a test selection in STOCKWATCHER.

During our analysis, we have observed many cases in which only a subset of the overall number of changes has been responsible for a test selection. At best, a test case can be referred back to a single code change. Figure 4.25 shows the results we have obtained from the analysis of two of our STOCKWATCHER versions (original versus one of the modified versions). Figure 4.26 shows the results for HUPA. The horizontal axis shows the different test cases. The bar displays how much code modifications are responsible for a test selection. Some of these tests are not affected at all by the changes (see the test cases with number 1, 4, 24-26, 29-31, 34-37, 40 in Figure 4.26) and therefore do not exhibit a bar in the chart (0% of the changes, i.e. no test selection). In all other cases, the developer gets precise information about a definite set of modifications that should be investigated if a test fails. So, the bug fixing process benefits enormously from the fault localization and the references between code modifications and test cases.

4.9 Discussion

Code Instrumentation: We have investigated in Section 4.5 and in Section 4.6 two different code instrumentation approaches. In the previous two sections, we have investigated the compiler-dependent approach. During the implementation, we have discovered several advantages and disadvantages in comparison to the compiler-independent approach. Table 4.2 gives an overview:

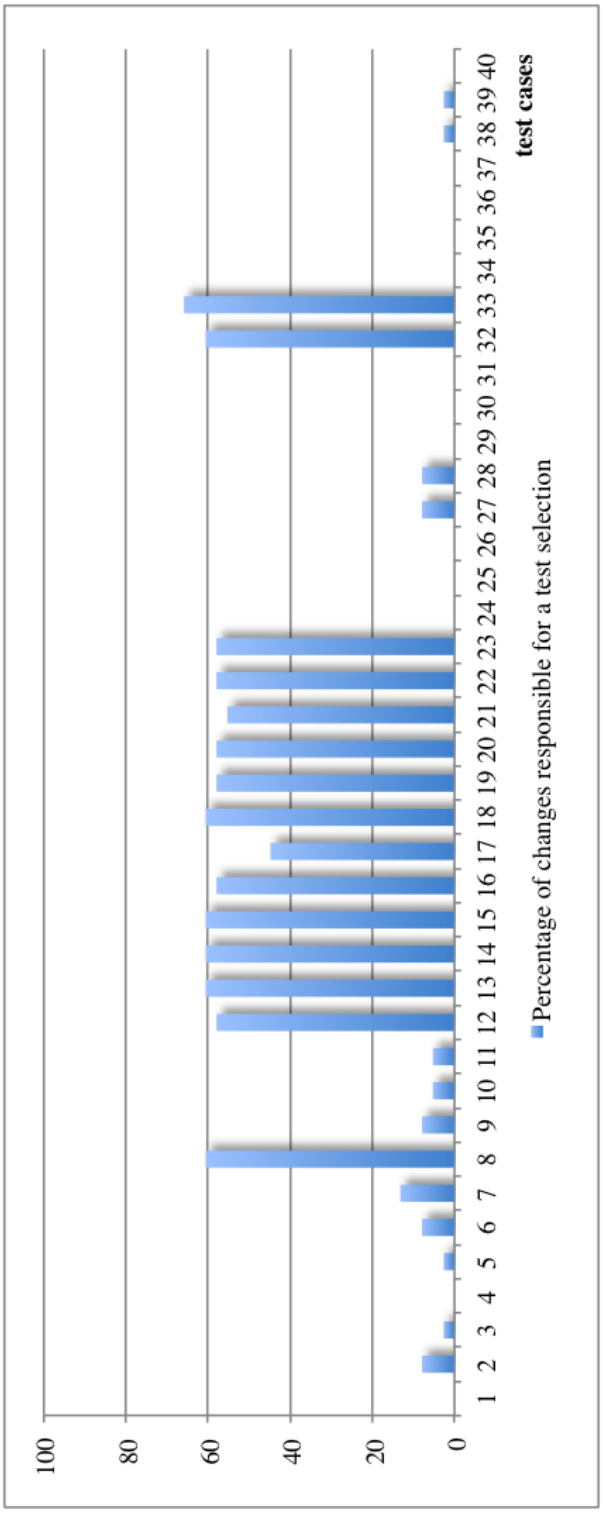


Figure 4.26: Code changes responsible for a test selection in HUPA.

	Code Instrumentation Variant	
	Compiler-Independent	Compiler-Dependent
Duplicate original source code	• desirable	✓ unnecessary
Instrumenting code of source language	✗ necessary	✓ unnecessary
Compiler modifications	✓ unnecessary	✗ necessary
Risk to break relation between code entity and CID	✗ yes, during transcompilation	✓ no
Reusable for other services	✓ yes	✓ yes

Table 4.2: Overview of advantages and disadvantages of code instrumentation approaches.

When instrumenting code in our compiler-independent variant, developers might feel uncomfortable if a tool starts injecting code directly into the production code. Indeed, if the source code contains modifications that are not under revision control yet, a bug in the instrumentation tool might destroy them. So it is desirable to create a temporary copy of the source code that will be used to do the instrumentation. (In fact, our compiler-independent variant creates a copy before doing the instrumentation.) Of course, this takes some extra time, depending on the size of the software project. Our compiler-dependent variant avoids these upcoming problems completely.

With regard to instrumenting the target code, both versions have to calculate the same CIDs. Nevertheless, the versions differ in the way they deal with these CIDs. Our compiler-dependent version uses a database whereas the compiler-independent version has to insert the instrumentation code within the source code. Especially when trying to write with several threads in the same source file, we have to take care of data integrity and consistency. This is harder than just adding CIDs and their associated meta-data into the database, which is designed for persisting many data entries in a fast way while ensuring *ACID* properties[3]. Beyond that, we can use the database data for statistical calculations. For example, in order to solve the coverage identification problem (see Section 1.2.3), we query the database and perform several calculations (see Chapter 7).

To make our compiler-dependent variant work, we have to extend the transcompiler's source code to incorporate CIDs while transfering code from the source to the target language. This implies that the source code of the transcompiler is open source and that there is a minimal documentation available that supports external developers in realizing these extensions. Naturally, compilers differ both in their implementation and in the programming language they are written in. Unfortunately, they often do not offer hooks where

[3] *ACID* stands for *A*tomicity, *C*onsistency, *I*solation, and *D*urability.

additional functionality can easily be added to the compiler. (For example, we might think of the GWT compiler.) Thus, the extensions are individual and highly specialized to every transcompiler. In the end, adaptations of the transcompiler have to be redone as soon as the original vendor decides to offer a new version. This implies a deep analysis of the underlying changes, which requires lots of time. (The naming of this variant harks back to this issue.) As a whole, our compiler-dependent variant requires much additional work that has to be done for every transcompiler and any of its versions. In contrast, our compiler-independent variant is generic. It can be used for any transcompiler, even for non-open source ones. (We might think for example of Codename One's transcompiler.)

Our compiler-independent variant relies heavily on the relation between source code and its instrumentation code. It is mandatory that the transcompiler leaves this relation intact. As opposed to that, our compiler-dependent version attaches instrumentation code to every code entity that has to be transcompiled. Both parts are encapsulated and remain together until the transcompiler writes the target code into a file. So we have a better control over the compiler and the insertion of CIDs into the target langauge. If the transcompiler supports source maps, they can even be extended to provide information about CIDs.

Finally, if we are interested in offering additional services such as calculating the code coverage, both variants are suitable even though we expect the compiler-dependent variant to be faster as already discussed above.

General Applicability: In order to decide which approach is more useful, we have to consider some more aspects that go beyond the pure code instrumentation. In general, both approaches explained in Section 4.5 ("Compiler-Independent Instrumentation") and in Section 4.6 ("Compiler-Dependent Instrumentation") are completely generic as they can be applied to many transcompilers and because they are not dependent on a specific programming language. Moreover, our method for calculating code changes also works for non-transcompiled desktop or web applications. In this case, instrumentation code is executed for example as bytecode as usual. The principle of inserting CIDs into a database in order to send queries remains valid. We enlarge upon these aspects in more detail in the next two paragraphs.

Effort to Support Other Transcompilers: Our Eclipse plug-in supports transcompiled web applications created with GWT and it is also able to handle standard Java desktop applications. Beyond that, both approaches can basically be used to support other transcompilers. Of course, this is by far more easier for our compiler-independent instrumentation as discussed before. Ignoring the additional effort in the compiler-dependent instrumentation to map code identifiers to the target code, there are only two crucial requirements in both approaches.

First, we require an AST representing the application in its source language;

or more general: we require that the source programming language is based on an abstract syntax tree. This is sufficient to generate unique CIDs for all the code entities of the source programming language that should be investigated for code changes and that could be the reason for a test selection, respectively.

As second demand, it must be possible to insert a module M in the application's target language in order to pass CIDs to the Eclipse plug-in (see Figure 4.21). If the application is cross-platform, the ability of M to establish a bi-directional persistent connection via WebSockets meets exactly this requirement. When regarding mobile applications, this means no restriction. Transcompilers like Codename One also support WebSockets [265]. But even if not, it is just a technical question to implement this. Otherwise, if it is a standard desktop application, a common connection to a database is sufficient.

In case of our compiler-independent approach, both the instrumentation code (calling the module M) as well as the module M itself may be transcompiled to the target programming language. Solely the CIDs may not be changed due to transcompilation or obfuscation. Our own code instrumentation structure (see Section 4.4.3) takes this into account. We relinquish to follow other approaches that use variables for code instrumentation because these variables could be obfuscated during transcompilation (see for example GWT or NeoMad). Consequently, it is difficult to localize code changes in the source language. To avoid this, we use method calls and pass CIDs as arguments (type String) to the module M. These strings always have a meaning that must be preserved. They will neither be modified during transcompilation nor during code obfuscation. In addition, the order of method calls has to be maintained by the transcompiler because the logic may never be changed. This ensures that the association established between the CID and the statement it represents will not break during transcompilation. So, assigning CIDs to the elements of an AST has the big advantage that we can trace the translation of code entities from the source language to the target language and back.

In our compiler-dependent approach, too, CIDs may not change. But as the instrumentation code is directly injected into the final source code of the target language, we neither have to cope directly with transcompilation nor with obfuscation. The injected functions call the module M and pass the CIDs as arguments as described above. The basic principle, the advantages of our code instrumentation, and the effort to send the CIDs to the Eclipse plug-in is the same as in the compiler-independent approach. Nevertheless, we cannot ignore the additional effort to support other transcompilers. The necessity to extend the implementation of the transcompiler by ourselves can be very troublesome and is a clear downside.

Effort to Support Other Programming Languages: The approaches in Section 4.5 and in Section 4.6 explain how we obtain test traces by running tests in a target language and how the information can be mapped to the source language in order to localize possible faults. In our compiler-independent approach, the general rule of thumb (inserting instrumentation code as function

call in front of the code they identify) is applicable to all AST-based programming languages. The exceptional cases depicted in step 1 of Section 4.5 might have to be adapted or extended for other programming languages. Our current rule set of exceptional cases is valid for Java. However, there are many other languages with similar syntactical rules like C++, C#, PHP, JavaScript, or Haxe. We expect the effort to fully support these languages to be minor. Moreover, such an adaptation is possible without loss of generality of the overall approach as the three steps depicted in Section 4.5 are still valid. This is even true for passing CIDs to the Eclipse plug-in with the aid of WebSockets as this protocol is widely supported by other languages. In addition, as long as there is no fundamental change in the concept of a programming language, our rule set and the exceptional cases remain valid even in new versions of the programming language.

In our compiler-dependent approach, first of all, the four steps depicted in Section 4.6 remain valid when other languages should be supported. Considering the effort, it could be roughly the same as in the compiler-independent approach, but it might be higher. The reason is that we have to distinguish between source and target languages. In the compiler-independent approach, we only consider the source programming language of the transcompiler. If the rule of thumb and the exceptional cases are fixed, we can support arbitrary target languages. In our compiler-dependent approach, this is different. As we have explained in step 2 of Section 4.6, we have to decide at which position the CIDs should be added in the target language. If there is only one target language, the effort is roughly the same as in our compiler-independent approach. However, some transcompilers such as Haxe [131] offer multiple target languages. So, we might have to implement step 2 several times.

4.10 Conclusion and Future Work

We have presented an enhanced regression test selection technique for transcompiled cross-platform applications. It builds on a technique that has originally been created for pure desktop applications. The enhancements in our technique are a result of the challenges we have been faced to. First, we have improved the way to compare two program versions in order to select test cases for re-execution. Besides, we have refined the fault localization. In order to address any kind of application, we have managed to design an own basic instrumentation approach as an refinement of former principles. It focuses in particular on how to transfer data collected during the execution of instrumentation code from an arbitrary application to a system on a different platform. Here, we have to deal with applications that have been transcompiled from a source language (the language they have been written in) to a different target language. Besides, the application might run on different platforms or even in the cloud. This constellation has resulted in a transcompiler-independent and a transcompiler-dependent instrumentation approach.

In our evaluation, we have shown the feasibility and the functional principle

of our compiler-dependent approach. As we have seen, it is able to reduce the test effort and to localize code modifications in the code of the source programming language. However, when comparing the compiler-dependent approach with the compiler-independent approach, the first-mentioned has more advantages. Notwithstanding this, it also lists the biggest disadvantages (see the discussion in Section 4.9 and consider especially Table 4.2). The significant additional work to provide and maintain this variant and the exclusive applicability to open source transcompilers reduces the chance to be used in the field. For these reasons, we investigate and evaluate the compiler-independent approach in the next chapter in more detail.

Furthermore, we believe that the performance and the efficiency of our RTS technique can still be optimized and that we need additional insights in order to provide an adequate solution for reducing the test effort and for localizing code modifications. We additionally incorporate these purposes in the next chapter. Beyond that, the coverage identification problem (see Section 1.2.3) has not been handled yet.

Chapter 5

Efficiency of Code Analysis and Fault Localization

5.1 Introduction

Analyzing code by means of control flow graphs requires time and a powerful system with enough RAM. These resources are limited though. Especially in large applications, generating two control flow graphs for the old and the new program version and the subsequent analysis might demand very much RAM and processing power. This deteriorates the performance, or even worse, makes the analysis unfeasible. Thus, it is very important to perform the analysis in a resource-saving way.

Ideally, the analysis would always be fast and precise in order to select only these tests that are in fact affected by a code change. This is extremely important in the area of UI/web tests, which take much time for test execution and which stress performance particularly. In the best case, only a small number of tests is selected for re-execution so that test selection clearly outperforms the retest-all approach. But in practice, analysis effort and analysis precision are negatively correlated. A fast analysis is always borne by the analysis precision and the ability to locate faults precisely in the source code, and vice versa. For this reason, it is important to optimize the test selection procedure to achieve better efficiency.

We start in Section 5.2 with related work. In Section 5.3, we discuss the challenges to apply a CFG-based RTS technique in a cost-efficient way on transcompiled applications. As it is difficult to find large mobile open source applications, we focus on testing GWT-based web applications. However, our findings are also applicable to other transcompilers because the underlying procedure is always the same as explained in the previous chapter. In Section 5.4, we present solutions to the challenges discussed before. We investigate several analysis levels at various precision. They serve as basis for a proposal of a dynamically customizable analysis level based on a heuristics that improves the performance so that even the analysis of large-scale transcompiled applications on customary computers is possible. Additionally, we investigate how the ef-

fort to do regression testing can be reduced in general. We pick up an idea proposed by Apiwattanapong et al. [12] that uses lookaheads and advance it according to our needs. This way, we are able to provide additional information about the kind of code modification (added, modified, or removed code) which in turn supports the fault localization. The findings of all our investigations have been the starting point for an extension of our Eclipse plug-in. It seeks to overcome the common nightly build and test cycle towards a fast executable and repeatable cycle of code changing, test determining, test case execution, and bug localization/test case fixing that resembles continuous integration [84]. Details can be found in Section 5.5. Then, in Section 5.6, we apply our tool with different parameter settings in an evaluation in order to find an efficient trade-off between a low memory consumption and the ability to detect code changes as exact as possible. We also investigate the runtime and the ability to reduce the analysis overhead with the aid of lookaheads. We discuss the results in detail and conclude in Section 5.7.

This chapter is partially based on two of our publications [133, 135].

5.2 Related Work

Cost Model: For judging the cost efficiency of RTS techniques, Leung and White have proposed one of the first "cost model to compare regression test strategies" [178, page 201] in an eponymous paper. In their cost model, they incorporate the costs for the system analysis (C_a), the test selection (C_s), the test execution (C_e), and the result analysis (C_r). C_a consists of costs incurred because of test engineers who have to accustom themselves to the application under test. The latter costs C_r include costs "C_u for understanding the program and specification in order to judge whether the program behavior or output is correct, and the cost C_c for comparing each test output to the expected output" [178, page 204]. Besides, they distinguish old tests (T_o) and new tests (T_n). The sum of all these components gives their definition of the costs C for the retest-all approach:

$$\begin{aligned} C(retest-all) = &[C_a(T_o) + C_s(T_o) + C_e(T_o) + C_u(T_o) + C_c(T_o)] + \\ &[C_a(T_n) + C_s(T_n) + C_e(T_n) + C_u(T_n) + C_c(T_n)] \end{aligned}$$

In the same way, they define the costs for techniques that use test selection

$$\begin{aligned} C(testselection) = &[C_a(T_s) + C_s(T_s) + C_e(T_s) + C_u(T_s) + C_c(T_s)] + \\ &[C_a(T_n) + C_s(T_n) + C_e(T_n) + C_u(T_n) + C_c(T_n)]\,, \end{aligned}$$

where T_s represents tests selected by the corresponding RTS technique in use. Naturally, the RTS technique is more cost-efficient when $C(testselection) < C(retest-all)$. The most important conversion of these equation is:

$$\begin{aligned} &C(testselection) < C(retest-all) \text{ if} \\ &\qquad C_s(T_s) < [C_e(T_o) - C_e(T_s)] + [C_c(T_o) - C_c(T_s)] \end{aligned}$$

which means that "the selective strategy is more economical than the retest-all strategy if the cost for selecting a subset of the previous tests is less than the cost for executing and checking the extra previous tests needed for the retest-all strategy" [178, page 205]. For more coherences, we refer the reader to the paper of Leung and White [178].

Another cost model has been proposed by Do and Rothermel [54]. They consider nine different kinds of costs such as costs for "test setup", "identifying obsolete test cases", or costs for resolving faults that have been detected late [54, page 143f.]. Some of these costs are very hard to determine in practice. For example, it has not been possible to determine costs for identifying obsolete test cases in the industrial application we have used in our evaluation. For this reason, we take the cost model of Leung and White as a basis for judging the efficiency of our technique.

Cost Efficiency: There are many empirical evaluations that report on the cost efficiency of RTS techniques [78, 119, 229, 232, 234, 238]. Rothermel and Harrold [232] have compared many different techniques, including the graph walk technique. An important observation has been that graph walk techniques are more precise, but have higher analysis costs.

In a more recent review, Engström et al. [78] have compared 28 RTS techniques. They have ascertained that all techniques (including the one that has served as starting point for our own approach) showed to be less costly compared with the retest-all approach. When considering the fault detection effectiveness, the authors have found that safe RTS techniques are superior.

Graves et al. [119] have compared the cost-effectiveness of several techniques, including a safe technique and the rest-all approach. As representative of the safe technique, they have investigated the safe graph walk-based RTS technique proposed by Rothermel and Harrold [233] which has served as one of the starting points of our own RTS technique. Graves et al. [119] have solely studied (non-transcompiled) C programs. Their results have been promising. Admittedly, the evaluated safe regression technique could sometimes merely reduce the test suite by 1%. But in contrast, their technique achieved sometimes a test reduction of 95%. When considering the whole picture, Graves et al. have observed that merely 60% of the test cases have been selected for re-execution.

Compared with the findings of Graves et al., Rothermel and Harrold [234] have observed similar results for several small applications that consist of 138 to 516 lines of code. For these programs, their RTS technique has selected 54,3% of the tests for re-execution on average. Nevertheless, Rothermel and Harrold have also found cases in the small programs where the test reduction was rather low. On the other hand, the results of the analysis of the largest application in their study are again very promising. The application consists of almost 50 000 lines of code and belongs to a real software applications that has evolved over a long time. The test reduction has been more than 95%. Of course, the test suites in all these studies did not contain UI/web tests and the authors have

not incorporated the extra effort that is necessary to deal with transcompiled applications. We investigate the impact of these new circumstances in today's applications and whether the promising numbers of test suite reductions still hold in our environment.

In another paper, Rothermel et al. [238] have found that the structure of the test suite is relevant for the cost-effectiveness of RTS techniques. They refer to this as *test suite granularity* [238, page 130]. According to them, a fine-grained test suite consisting of many small tests rather than a few big tests requires more execution time and detects less faults. Nevertheless, the authors also remark that a more coarse-grained test suite involves the danger that there is (almost) no test suite reduction. Naturally, this is – as we want to emphasize – counterproductive for applying a RTS technique efficiently.

Kim et al. [164] have detected that the number of tests selected for re-execution is related to the test interval. According to them, the bigger the test interval, the more tests are selected for re-execution. However, as explained in Section 2.1 and in Section 2.2, we want to remind that code changes should anyway be integrated in a repository frequently. More details can also be found in Fowler's article on *Continuous Integration* (*CI*) [84]. The commits are the trigger for new builds and test runs. So, as the intervals are small, we do not expect the concern of Kim et al. to be the reason for a potentially high number of selected tests.

Analysis Effort, Analysis Levels, and Heuristics: Apiwattanapong et al. [12] incorporate *lookaheads* [12, page 6] in their tool JDiff (see Section 4.2, Paragraph "Code Analysis and Fault Localization") that is able to detect code changes in the object-oriented language Java. As soon as the analysis of P and P' detects that the current nodes $n \in P$ and $n' \in P'$ do not correspond to each other, they try to find matching nodes by using a lookahead in a Breadth-First Search. They always compare n to the successor nodes m'_i ($i \in \mathbb{N}$) of n' and they compare n' with the successor nodes m_j ($j \in \mathbb{N}$) of n. If the search could not find a corresponding node, they continue the comparison of n (n') with the successor(s) of m'_i and the comparison of n' with the successor(s) of m_j, respectively. Apiwattanapong et al. continue this procedure until they find a corresponding node or until the lookahead value is reached. If they detect matching nodes $p \in P$ and $p' \in P'$, they check the edges starting from p and p' for a match and – if successful – continue the analysis with the successor nodes. So, Apiwattanapong et al. use the lookahead to find for the nodes in P corresponding nodes in P'. If the analysis fails to find matching nodes before the maximum lookahead values is reached, they consider the nodes $n \in P$ and $n' \in P$ as "modified" [12, page 16]. All remaining nodes in P are considered as deleted, remaining nodes in P' are considered as added. We also use a lookahead, but in order to find a point in the code where the comparison can be resumed in order to analyze the rest of the code for additional code modifications. For this purpose, we use the lookahead in a two-stage analysis process. Apiwattanapong et al. do not analyze code modifications in detail. Besides, in

some cases at least, their algorithm takes more time to find coinciding nodes than ours. Let us assume that the current nodes n and n' do not match due to code modifications. Let us also assume that the successor node of n in P is q, which in turn has a successor node r. r is succeeded by u. Likewise, let us assume that n' in P' has a successor node s, which in turn has a successor node t. t is succeeded by u', which matches u. Apiwattanapong et al. have to perform three analyses with lookaheads in order to detect that u and u' are matching nodes. In contrast, with our two-stage algorithm, we perform a single lookahead analysis and check directly whether nodes have been modified, added or removed. Consequently, we are faster in detecting code modifications. Further details on our two-stage analysis follow in Section 5.4.4.

A different technique focusing on the reduction of analysis overhead has been presented by Orso et al. [215]. They investigate the code at two stages. They perform a high-level analysis (stage 1) narrowing the choice of code that should be analyzed in a subsequent code investigation (stage 2). In stage 1, they use a partitioning algorithm that is based on a special *Interclass Relation Graph* [215, page 244]. This way, they are able to identify changes of statements or declarative changes. In stage 2, they compare only these parts of the code that are affected by changes. Although we share the idea of reducing the analysis overhead via a precedent investigation, we use a simplified approach to check quickly which parts of the code require a precise check for changes. Unlike Orso et al., we do not exclude parts of the code from the analysis for security reasons. Instead, we reduce the analysis granularity dynamically as needed and argue that our approach is not insecure.

The approach of Bible et al. [29] is close to our suggested technique. They report on advantages and problems of a coarse-grained safe regression testing technique and another safe technique that analyzes the code on statement level. On this basis, they develop a prototype for a hybrid technique that tries to combine the best properties of both approaches. However, we want to remark that they have no facility to adjust the analysis level as needed. Besides, they do not employ project related data to decide on which analysis level might fit the best.

Gligoric et al. [93] analyze code very coarse-grained at file level. They avoid the creation of CFGs. Instead, they compare the checksums of files in order to detect changes. For the test selection, they merely model the dependency of files that are traversed by a test class or by a test case. This approach speeds up the analysis but is of course less precise. The authors admit that their technique – implemented as extension of JUnit in a tool called Ekstazi – usually selects more tests than more fine-grained techniques. Nevertheless, they state that Ekstazi is faster than the retest-all approach. Furthermore, they have compared their tool with another recent RTS tool called FaultTracer [293] (see Section 4.2, Paragraph "Code Analysis and Fault Localization") which is, according to their findings, even slower than the retest-all approach. With respect to the findings of other authors (see the beginning of this section), we are concerned that changes affect many or even all tests. The number of tests

selected for re-execution is decisive. The fastest analysis cannot outperform the retest-all approach if the RTS technique is not able to reduce the number of selected tests. We expect this concern to be especially true in the context of UI/web tests as these kinds of tests do not necessarily check mostly distinct functions like JUnit tests usually do, but rather execute lots of classes/files. Gligoric et al. have solely focused on projects with standard JUnit tests, but have not considered transcompiled (web/mobile) applications. Beyond that, the very coarse-grained analysis at file level contradicts our target to precisely locate changes/possible faults in the source code (see the fault localization problem, Section 1.2.2). The time for locating and fixing bugs has also to be taken into account. Gligoric et al. do not consider this aspect. For this reason, we introduce a heuristics as trade-off between a fast analysis and the lowest possible number of tests selected for re-execution.

5.3 Motivation and Challenges

Every RTS technique should achieve two goals: a) an exact localization of code changes in order to select only these tests that are actually affected, and b) the benefit of the technique has to outweigh the overhead of a retest-all approach. In case of web tests that run in a web browser, runtime is limited by factors like the client-server communication or the data loading from databases. For this reason, web tests are known to be time intensive. For example, in the company that allowed us to investigate our approach, the test suite consists of 105 tests and takes 9 hours when using the common restest-all approach. So in fact, running a few tests more or less makes a huge difference. Therefore, techniques that aim at reducing the test effort have potential to boost the test selection.

Basically, the test suite could be split into several parts in order to run them in parallel and to reduce the time consumption. Nevertheless, this is accompanied by an increase of costs. On the one hand, there have to be enough powerful virtual machines available that have to be maintained and kept up to date. On the other hand, every machine requires a license of the testing platform. Especially the costs for licenses and support are considerable[1] and usually cannot be provided in a sufficiently large number to support continuous integration.

With regard to the choice of technique, we want to remind that a safe RTS technique is preferable as it does not miss test cases that reveal a bug (see Section 2.2). The findings of Graves et al. [119] encourage us. According to them, the safe RTS technique has performed well in terms of cost-effectiveness. Despite the fact that there has been some cases in which their technique was merely able to reduce the test suite by 1%, there have also been cases in which 95% fewer tests had to be executed than in a retest-all approach. On the median, a test reduction of 40% has been achieved while revealing all faults. In

[1] Standard business testing tools are priced at more than 2000 € (see e.g. `http://smartbear.com/product/testcomplete/pricing/`)

contrast, Orso et al. [215] (see also Section 5.2) report that safe selective regression testing techniques are less cost-efficient compared to unsafe techniques in particular when applied to big software systems. According to them, the reason is that the safe technique takes more time than the retest-all approach. As the article is more than ten years old and as today's computers have significantly more internal memory and power, these results seem not to be crucial any more. Instead, the efficiency of our technique might be compromised by the additional time needed for:

- instrumenting the Java code,
- executing the instrumented application with its transmission of CIDs to the logging server and their insertion into a database for further processing,
- creation of EJIGs for the old program version P and the new program version P',
- comparing the two graphs,
- selecting tests cases by querying the database to find these ones that traverse the CID of a changed node in the EJIG.

Especially the nature of UI/web tests can have a significant impact on the number of selected tests for re-execution. Distinct UI/web tests do not test mostly disjoint functions as unit tests usually do, but might execute the same UI code. Therefore, modifications in the code may affect easily many UI/web tests, which makes a test suite reduction harder. This concern is also confirmed by Rothermel et al. [238] who have observed that it is more effective when the test suite contains many small tests rather than a few big tests.

All these findings and reflections make obvious that it is beneficial to do further research on a technique that is highly optimized for test selection and fault location.

5.4 Approach

A main factor that influences the time exposure is the precision used to perform the analysis of the Java code. Here, we can distinguish various levels of precision. For example, the code could be analyzed for code changes rather coarse-grained by comparing method declarations. A more fine-grained analysis could perform this comparison on statement or even on expression level. The precision level impacts the instrumentation of the Java code. By logging the execution of every entity (methods, statements, expressions), the level of precision has a high impact on the performance overhead introduced by instrumentation. Queries to select the tests that have to be re-executed therefore take longer. Besides, when doing a fine-grained analysis of the code, the EJIGs contain more nodes. So any comparison of the two graphs potentially takes

more time and additionally, it leads to increased memory consumption. However, a fine-grained analysis results in a better fault localization and therefore in a reduced test selection, which is one of our main targets. For this reason, we introduce two levels of precision for our analysis which we call *Body Declaration* and *Expression Star*. They will serve as starting point to define a heuristics for finding a trade-off.

The runtime of the analysis is additionally affected by the completeness of the analysis. Harrold et al. [126] stop comparing the CFGs of P and P' as soon as a code modification has been detected. If a test case t covers this code modification, they just add t to the set of test cases that have to be re-executed on P'. This approach is of course completely valid and does not impair a safe test selection. However, we add for consideration that there might be more code changes throughout the remaining program execution that affect other test cases. Without a continuing analysis, bugs in these code changes might not be detected before a future analysis gets started. Moreover, in the approach of Harrold et al., it might happen that a test case is selected for re-execution due to a code modification m_i, but the test case fails due to another code modification m_j that appears later in the code. Using m_i for searching for bugs would be clearly misleading. In the approach of Harrold et al. however, code modification m_j could only be detected by a subsequent analysis. Consequently, it would be helpful to know all possible code changes that might be the cause for a test failure. As already explained in Section 5.2, Apiwattanapong et al. [12] have proposed a *lookahead* [12, page 6] to find matching nodes within hammock nodes. We take this idea up to do a more in-depth analysis using two different algorithms. There, the lookahead defines an upper limit of how many levels of successor nodes will be investigated to find additional changes. More details follow in Section 5.4.4.

In this section, we describe the details of our approach to deal with the mentioned factors.

5.4.1 Analysis Levels at Various Precision

We introduce the analysis precision level *Expression Star* (*E**) which calculates CIDs and generates nodes in the EJIG on expression level with some exceptions. For example, all literals have been excluded as they would increase the logging amount enormously without providing any benefit for fault localization.

In the analysis precision level *Body Declaration* (*BD*), nodes represent body declarations in the code. In Java, we might think of methods, types or fields. So this level is less precise and is not able to distinguish modifications in different statements within a method. As a consequence, it risks selecting too much tests. The example code in Figure 5.1a shows a method of an initial program version P. In P', there is a code modification in the `else`-branch (see Figure 5.1b, where the call `bar()` has been changed to `bazz()`). Figure 5.1c shows for each test case the list of CIDs it has traversed during the last test run. To keep the example simple, we just name the CIDs `cid1`, `cid2`, and `cid3`. Solely test 2

traverses the changed code (see Figure 5.1d). When comparing P' with the old version P, E* considers the CIDs in the case distinction and selects only test 2. In contrast, BD only considers the CID representing the method declaration (`cid1`) and will select both tests.

```
1  private void m(boolean mycase) {
2    InstrumentationLoggerProvider.get().instrument("cid1");
3    if(mycase) {
4      InstrumentationLoggerProvider.get().instrument("cid2");
5      foo();
6    } else {
7      InstrumentationLoggerProvider.get().instrument("cid3");
8      bar();
9    }
10 }
```

(a) Original version P with a code modification in the `else`-branch.

```
1  private void m(boolean mycase) {
2    InstrumentationLoggerProvider.get().instrument("cid1");
3    if(mycase) {
4      InstrumentationLoggerProvider.get().instrument("cid2");
5      foo();
6    } else {
7      InstrumentationLoggerProvider.get().instrument("cid3");
8      bazz();
9    }
10 }
```

(b) Version P' with a code modification in the `else`-branch.

CIDs traversed by	
test 1	test 2
cid1	cid1
cid2	cid3

(c) CIDs in distinct test cases.

	Test selection in	
Test	E*	BD
Test 1	×	×
Test 2		×

(d) Test selection results per precision level.

Figure 5.1: Test selection in various precision levels.

The EJIG created by the BD-level usually contains less nodes than the EJIG created by E* which leads to a reduced memory consumption. Furthermore, we would expect an improvement of the runtime as there are less nodes to compare. In the creation of the EJIG itself via the BD-precision level, we do not expect a significant speedup. Technically, the EJIGs are created by traversing the AST provided by the Eclipse Java Development Tools (JDT) as already mentioned in Section 2.5. We drill down the Eclipse AST and stop creating nodes for the EJIG as soon as the current node in the Eclipse AST does not match the analysis precision level any longer. Due to the fact that method invocations are expressions, we have to continue walking through the AST even if the BD-level

is selected for precision in order to model calls of methods in the control flow. So, even in an EJIG that is based on a BD-precision level, we need to create nodes that represent method invocations. We call this kind of nodes *glue nodes* because without them, the control flow gets interrupted.

Although we need to traverse method invocations even if the BD-level is selected for precision, we solely want to have nodes in the final set of code modifications that fit the BD-level. This way, we expect to gain an edge over the fine-grained precision level. Of course, this affects the Algorithm 1 in Section 4.3.2. Algorithm 3 shows the version of the original algorithm with an update in line 21. Here, we call a helper method `handleGlueNodes`. It just checks whether the target nodes t' is not marked visited yet and whether t and t' fit the chosen precision level. If so, the algorithm continues as usual with line 22. Otherwise, we directly push the node pair t and t' on the `stack` and continue with the loop in line 13.

5.4.2 Dynamically Customizable Analysis Level Based on a Heuristics

We want to find a reasonable trade-off that considers the strengths and disadvantages of the E*- and BD-precision level in order to optimize runtime performance and memory consumption, but at the same time guaranteeing both a precise selection of test cases and the identification of code changes in the underlying Java code.

The key idea of a dynamically adaptable analysis level is that there might be cases (e.g. when parts of the code usually never change at all or only to a limited extent), in which a detailed analysis does not really provide more information but requires more memory and potentially loses time in preparing or inspecting code. For this reason, we propose a hybrid form of analysis in order to reduce the gap between precision and performance. In this context, it is crucial to find a competitive decider. Especially in the area of test prioritization, heuristics are frequently used to decide on which test should be selected preferably. More details can be found for example in the survey of Yoo and Harman [290]. We modify this strategy to decide which parts of the code might be investigated less thoroughly.

In order to find a suitable decider at which precision level a `CompilationUnit` in the Eclipse AST (i.e. a class, interface or enum) should be analyzed, we have focused on the potential of single source files to contain code that cause a test to fail. In a first approach, we have mediated using the change frequency of `java` files as decider. Alternatively, we have reflected on analyzing those `CompilationUnit`s on E*-precision level that have been responsible for an increased test selection in a previous analysis. Kim and Porter [163] have also investigated different kinds of data obtained during previous test runs. Their most similar idea has been to check which tests revealed faults the most often in the past. (More details follow in Section 6.2). The major difference to our first plans is that they focus on the ability of a test to reveal faults rather than

```
   input  : s: start node in P; s': start node in P'
   output: d: List of dangerous edges
 1 if ¬nodeEquivalent(s', s) then
 2 |   e' ← n'.getAnyEdge();
 3 |   d ← d.add(e');
 4 end
 5 stack.push(s, s');
 6 while stack not empty do
 7 |   n ← stack.pop.oldNode;
 8 |   n' ← stack.pop.newNode;
 9 |   if n' is marked visited then
10 |   |   continue;
11 |   end
12 |   mark n' visited;
13 |   foreach edge e leaving n do
14 |   |   e' ← match(n', e);
15 |   |   if e' is empty then
16 |   |   |   // handle removed code; more details follow in Section 5.4.4;
17 |   |   |   continue;
18 |   |   end
19 |   |   t ← e.targetNode;
20 |   |   t' ← e'.targetNode;
21 |   |   handleGlueNodes(t, t');
22 |   |   if ¬nodeEquivalent(t', t) then
23 |   |   |   // distinguish kind of modification;
24 |   |   |   // more details follow in Section 5.4.4;
25 |   |   |   d ← d.add(e');
26 |   |   end
27 |   |   else if t' is not marked visited then
28 |   |   |   stack.push(t, t');
29 |   |   end
30 |   end
31 |   foreach edge e' leaving n' without counterpart do
32 |   |   // handle special cases; more details follow in Section 5.4.4;
33 |   |   d ← d.add(e');
34 |   end
35 end
```

Algorithm 3: Comparing two EJIGs - Handle glue nodes.

on source files. For our purposes, we would have to go one step further in order to analyze the potential of single source files to let tests fail. Elbaum et al. [71] have pursued a strategy that partially fits our demands. They have determined functions that showed to be faulty in previous program versions.

However, when analyzing the industrial software MEISTERPLAN (see Section 5.6.1) for our evaluation, we have detected that the number of changes are neither Gaussian distributed, nor do the test selection prone code changes in P correlate significantly with changes in P'. Of course, this may be different from software to software but obviously, the change frequency and the likelihood of `CompilationUnits` being responsible for a high test selection in the past are not suitable criteria for all kinds of applications. For these reasons, we had to refine out strategy.

Our final heuristics does not rely on data from previous test runs any more. Instead, the heuristics uses a check to decide which source files have been changed. Here, a change can be an addition, a modification, or a removal of a file in P'. (For simplicity, we refer to them as *changed files*.) This is done by querying the code repository before the creation of the EJIGs starts. Of course, irrelevant changes (e.g. white spaces or blank lines) are ignored. The heuristics takes the list of changed files as input and influences directly the number of nodes both in P and in P'. When traversing the ASTs of P and P', the heuristics checks for each `CompilationUnit` whether it is affected by a code change. If there is a match, the heuristics creates corresponding nodes until the E*-precision level is reached. Otherwise, the EJIGs contain only nodes that represent body declarations or more coarse-grained elements plus – as explained before – glues nodes to handle method invocations that connect method declarations.

Our heuristics is similar to the one proposed by Orso et al. [215] (see also Section 5.2). In favor of a quick and easy computation, it is less precise as we do not analyze any relationships among classes. Orso et al. consider the class hierarchy or relations between classes. They argue that a heuristics depending solely on modifications in the source files fails to identify and treat declarative changes in the context of inheritance precisely. Besides, they state that such a test selection is not safe. This is true in their specific approach for a new RTS technique that is based on partitioning and selection. We also agree that this is a valid remark in the context of choosing whether a compilation unit should be analyzed at all. In our case however, this claim does not apply since we only use the heuristics for adapting granularity between BD and E*. We still analyze the whole code, but represent the code in some `CompilationUnits` more coarse-grained. So our approach is still safe. Their argumentation that a heuristics based solely on modifications is not able to identify declarative changes does also not apply in our case. To illustrate this, we use the relevant part of the same example as Orso et al. employ in their argumentation (see Figure 5.2) and extend it by an additional class `HyperA`. We deliberately omit a code change (Orso et al. have changed the statement in line 10 of P' from `i--` to `i++`) as the authors just use this as an example for a declarative code change that is easy to recognize. Of course, we are also able to recognize this declarative code change.

In the example in Figure 5.2, `LibClass` represents a library that returns an instance of type `A` or `SuperA`. `A.foo()` has been added in P' (see Figure 5.2b in

```java
public class SuperA {
  int i=0;
  public void foo() {
    System.out.println(i);
  }
}

public class A extends SuperA {
  public void dummy() {
    i--;
    System.out.println(-i);
  }




}

public class B {
  public void bar() {
    SuperA a = LibClass.
        getAnyA();
    a.foo();
  }
}

// Library that returns either
    an instance of SuperA or A:
public class LibClass {
  public static SuperA
      getAnyA() {
    return new A();
  }
}


public class HyperA {
  public void dummy() {
    // do something
  }
}
```

(a) Version P

```java
public class SuperA {
  int i=0;
  public void foo() {
    System.out.println(i) ;
  }
}

public class A extends SuperA {
  public void dummy() {
    i--;
    System.out.println(-i);
  }
  public void foo() {
    System.out.println(i+1);
  }
}

public class B {
  public void bar() {
    SuperA a = LibClass.
        getAnyA();
    a.foo();
  }
}

// Library that returns either
    an instance of SuperA or A:
public class LibClass {
  public static SuperA
      getAnyA() {
    return new A();
  }
}


public class HyperA {
  public void dummy() {
    // do something
  }
}
```

(b) Version P'

Figure 5.2: Declarative code change taken from Orso et al. [215], extended by class `HyperA`.

comparison to Figure 5.2a) and this is why `SuperA.foo()` is not traversed any more in P' if the library returns an instance of type `A`. In the EJIG, this modification is represented by a modified call edge. When looking at the EJIG of P, there are two call edges pointing from `a.foo()` in `B.bar()` to `SuperA.foo()`. The call edges represent the possible return types of `LibClass.getAnyA()` (i.e. `SuperA` and `A`). In P', the EJIG has a redirected call edge (dangerous edge)

whose start node is `a.foo()` in `B.bar()` which points to `A.foo()`. (The other call edge is left untouched.) Our heuristics will consider `A` in P' as changed and it will do a fine-grained analysis on any code within `A`. In terms of our heuristics, this is correct and meets our expectations. With respect to test selection, our algorithm would select any test for re-execution that traverses the two call edges starting from `a.foo()` in `B.bar()`. It does not matter whether the library returns an instance of type `SuperA` or `A`. The selection remains safe. Only tests that traverse one of these call edges will be selected for re-execution, which is absolutely precise and exactly what we would expect. Other tests instantiating `A` that just call `A.dummy()` will not be selected for re-execution. This is different from the approach of Orso et al. who have to deal with the problem that all tests instantiating `A` have to be selected for re-execution, regardless of whether they traverse `A.foo()` or not.

Now we would like to extend the example of Orso et al. to discuss another declarative change. Imagine that in P', `A` just extends from an already existing, unchanged class `HyperA` instead of overriding `SuperA.foo()`. (So please ignore lines 13-15 in Figure 5.2b.) For the sake of type correctness, also assume that the (external) library `LibClass` returns an instance that fits our code (i.e., it has to return an instance of `HyperA` instead of `SuperA`, see line 28). In this case, there is a single declarative change in the inheritance chain (`A extends HyperA` instead of `A extends SuperA`). Our heuristics will again analyze `A` in detail, but actually, the code in `A` did not change. Instead, a test case might now execute the `dummy()`-method of `A` rather than the one of `HyperA`. Nevertheless, this does not affect the safety of our approach as we just represent `A` more fine-grained as necessary. Again, the call edge changed and any test that traverses the caller of the `dummy`-method will be selected for re-execution. The same is true when considering class `B`. It calls a library that now always returns either an instance of `HyperA` or `A`. Tests traversing `a.foo()` in `B` will be selected as the call edges in P' differ from those in P.

So on the one hand, we run the risk that that our heuristics analyzes code more fine-grained as necessary. But on the other hand, the heuristics is easy to compute and does not require much additional time. Moreover, we analyze the code to a high probability at a high precision level when it is actually necessary. Besides, with our heuristics, we reduce the memory consumption whenever there is no doubt that a fine-grained analysis is unnecessary.

As already mentioned in Section 5.2, our heuristics has the basic idea of using a hybrid form of analysis in common with the approach proposed by Bible et al. [29]. They have also tried to improve the efficiency by reducing the analysis overhead. However, they have applied this strategy on the whole software under test. This differs from our approach in such a way, that we assign each class the analysis granularity and the lookahead individually. We expect that a coarse-grained analysis of the entire source code would lead to a big loss in precision and an increasing number of test cases selected for re-execution. Here, we want to remind the example in Figure 5.1 in Section 5.4.1 where we cannot distinguish tests that execute specific branches.

5.4.3 Trace Collection Costs and Analysis Costs

The total costs C_{traces} for collecting test traces depend on the costs for the code instrumentation C_{instr} plus the costs C_{log} for traversing and sending the CIDs to the logging server plus the costs $C_{transcomp_\Delta}$ for transcompiling instrumentation code. $C_{transcomp_\Delta}$ is the difference between the costs $C_{transcomp_{instr}}$ for transcompiling an instrumented code version P and the costs $C_{transcomp}$ for transcompiling the source code of P without any instrumentation code. So:

$$C_{traces} = C_{instr} + C_{transcomp_\Delta} + C_{log}$$
$$C_{transcomp_\Delta} = C_{transcomp_{instr}} - C_{transcomp}$$

All parts in C_{traces} increase linear with the number of instrumentations. We use the number of nodes representing a code entity in E* and BD, respectively, to compare the total costs for collecting traces on our precision levels. In real programs, the set of nodes represented by BD is smaller than the set of nodes represented by E*. The BD-precision level is therefore cheaper in terms of test tracing.

The costs for our heuristics $C_{H_{traces}}$ basically depend on the number of `CompilationUnits` represented on E*-level and are bounded by the costs for collecting traces on BD-level and E*-level, respectively. So, theoretically, it is

$$C_{BD_{traces}} \leq C_{H_{traces}} \text{ and } C_{H_{traces}} \leq C_{E*_{traces}}$$

($C_{BD_{traces}}$ is only equals to $C_{E*_{traces}}$ if the source code consists of body declarations solely. In real programs however, there are always statements, too. So $C_{BD_{traces}} = C_{E*_{traces}}$ will never occur in practice.) As we do not know in advance which `CompilationUnit`s will change in P', the entire trace collection for P has to be done on E*-precision level. Hence, the costs $C_{H_{traces}}$ and $C_{E*_{traces}}$ are the same. So, in terms of cost for collecting traces, using a heuristics does not pay off yet.

Our heuristics takes effect when investigating the costs for analyzing and comparing the EJIGs. Some parts of the EJIGs are represented fine-grained on E*-level, the rest is represented coarse-grained on BD-level. So:

$$C_{BD_{analysis}} \leq C_{H_{analysis}} \text{ and } C_{H_{analysis}} \leq C_{E*_{analysis}}$$

When there are only a few changes, most nodes in the EJIGs represent body declarations. Then it is:

$$C_{H_{analysis}} = c \cdot C_{E*_{analysis}} + (1 - c) \cdot C_{BD_{anaylsis}}$$

where $c \in [0; 1]$ is the percentage of nodes that have to be represented fine-grained. So in the best case, it is:

$$C_{H_{analysis}} = \lim_{c \to 0} \left(c \cdot C_{E*_{analysis}} + (1 - c) \cdot C_{BD_{anaylsis}} \right) = C_{BD_{anaylsis}}$$

We can link our insights to the cost model presented by Leung and White [178, page 205] (see Section 5.2). To this end, we transpose their equation:

$$C_s(T_s) < [C_e(T_o) - C_e(T_s)] + [C_c(T_o) - C_c(T_s)]$$
$$C_s(T_s) + C_e(T_s) + C_c(T_s) < C_e(T_o) + C_c(T_o)$$

In this equation, $C_{traces} = C_{instr} + C_{transcomp_\Delta} + C_{log}$ is equivalent to $C_e(T_s)$ (the test execution costs). $C_s(T_s)$ includes the checkout of P, the comparison of P and P', and the check for tests that are affected by changes. So $C_s(T_s)$ corresponds in our case to $C_{H_{analysis}}$. That is, the costs for test selection plus the costs for executing the selected tests plus the costs for checking the output of these tests has to be smaller than the costs for executing all the old tests plus the costs for checking the output of all the old tests. In more detail, $C_e(T_s)$ and $C_c(T_s)$ have to outweigh the extra costs $C_s(T_s)$ although $C_e(T_s)$ itself includes additional costs for instrumentation, transcompilation, and logging.

5.4.4 Recognizing More Code Changes with Lookaheads

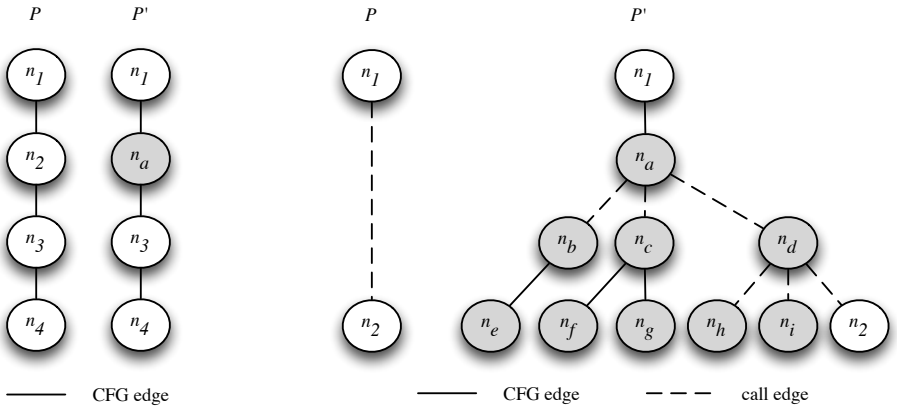
Code modifications are often not local to one particular position in the code. For instance, a refactoring that changes an instance variable name may cause multiple method bodies to change at several positions. However, many RTS techniques are unable to localize these positions exactly as they only detect changes at file or method/function level (e.g. [16, 38, 93, 142, 143]). Graph-based RTS techniques (e.g. [126, 228, 230, 233, 236, 272]) usually compare the program versions P and P' only up to the first occurrence of a code modification. Other changes that occur in the CFG later are not examined by these techniques any more. Their identification requires other techniques like change impact analysis (see Section 2.6) or manual inspection. Both possibilities of course require additional time. Missed impacts of code changes emerge not before the RTS technique is re-executed.

In order to reduce this overhead to find additional modifications, we adopt a feature of the approach presented by Apiwattanapong et al. [12] that uses *lookaheads* [12, page 6] to detect more changes. In contrast to Apiwattanapong et al., we employ a two-stage algorithm which is applied directly on the different nodes of the EJIG. It is implemented in a helper method `nextNodeOracle` which is located in line 24 in Algorithm 3 of Section 5.4.1. Algorithm 4 shows the updated part (lines 22-26). The two-stage algorithm uses as input the last matching nodes in P and P', and a reference to the `stack` that enables pushing of new nodes. Besides, we use a user defined lookahead value that defines an upper limit how many levels of successor nodes (see Section 2.5) have to be analyzed in order to find a coinciding node where the comparison of nodes can be resumed. Figure 5.3a and Figure 5.3b sketch the situation. In both cases, there are two CFGs representing P and P'. The node n_1 is the last matching node. In a *Parallel Search* (first stage), we try to find whether a successor node has just been *modified* in P' (see node n_a in Figure 5.3a) and whether there is a common node in P and in P' from where the program execution coincides again.

In the example, this applies to n_3. In more detail, we consider all the successor edges of the differing nodes in P and P' and check the kind of edges, the label of the edges, as well as the labels of their target nodes t_{i_P} and $t_{i_{P'}}, i \in \mathbb{N}$ against each other. The comparison of the kind of edge is important to distinguish call edges from CFG edges (see Section 4.3.1, Paragraph "Specialized CFGs for Java Software") and to distinguish overridden methods in class hierarchies. If we find a coinciding node c in P and in P' (i.e. a node with the same label that can be reached via the same kind of edge that has the same edge label), we stop the algorithm and continue our CFG analysis from c as usual. Otherwise, we consider the successor edges and their corresponding target nodes of t_{i_P} and $t_{i_{P'}}$. This procedure continues until we find coinciding nodes or until we reach a maximum number of levels of successor nodes defined by a lookahead parameter.

```
   input  : s: start node in P; s': start node in P'
   output: d: List of dangerous edges
22 if ¬nodeEquivalent(t', t) then
23 |  // distinguish kind of modification;
24 |  nextNodeOracle(e, e', stack);
25 |  d ← d.add(e');
26 end
```

Algorithm 4: Comparing two EJIGs: `compare` - Distinguish kind of modification.



(a) Parallel Search (b) Breadth-First Search

Figure 5.3: Two-stage algorithm to find matching nodes after modification.

In the example in Figure 5.3a, we start the search from node n_a. Here, it would be sufficient to set the lookahead value to 1 in order to find the next matching node n_3 in P and P'. If the Parallel Search algorithm is successful, the comparison of the EJIGs representing P and P' continues normally. In the

example, we continue with the successor of n_3. This way, we are able to detect additional changes that cannot be found by standard approaches as explained above.

If the Parallel Search-algorithm does not succeed, we continue with a second stage. Therein, we use a *Breadth-First Search* to determine whether nodes have been *added* or *removed* in P'. In Figure 5.3b, this applies for the nodes $(n_a - n_i)$. Between n_1 and n_2, the nodes n_a and n_d are new. To detect this, we start our Breadth-First Search from the node n_P in P that differs from $n_{P'}$ in P' with three parameters: the label of the actually expected node n_P, and the edge e_P that has to be traversed to reach this node n_P. The third parameter is the differing node $n_{P'}$ in P'. (In our example, it is $n_P = n_2$, $e_P =$ call edge and $n_{P'} = n_a$.) The algorithm starts to compare n_P against the target nodes $t_{i_{P'}}, i \in \mathbb{N}$ of $n_{P'}$ and e_P against the edges that have to be traversed in order to reach $t_{i_{P'}}$. In the example in Figure 5.3b, $n_P = n_2$ is compared against the set of target nodes $\{n_b, n_c, n_d\}$. This comparison continues until the coinciding node has been found or until we reach the maximum looakahead value. In the example, nodes with level 1 (i.e. $\{n_b, n_c, n_d\}$) do not contain the node n_2. So we consider the set of successor nodes of $t_{i_{P'}}$. This is $\{n_e, n_f, n_g, n_h, n_i, n_2\}$. Here, we find node n_2. The algorithm exits and the comparison of P and P' continues from n_2 as usual.

If the Breadth-First Search does not succeed, we repeat the Breadth-First Search with changed input. We restart the algorithm with swapped input from the node n'_P in P' that differs from n_P in P with three parameters: the label of the actually expected node n'_P, and the edge e'_P that has to be traversed to reach this node n'_P. The third parameter is the differing node n_P in P. The workings remains the same as described before.

If one of the two algorithms succeeds, we are able to determine the kind of code change. A success of the Parallel Search algorithm indicates *modified* code. In the example in Figure 5.3a, we consider the edge with start node n_1 and target node n_a as *modified edge* and add it to the list of dangerous edges. If the Breadth-First Search succeeds with the first input variant (see example in Figure 5.3b), code has been *added*. A success with the second input variant indicates *removed* code. Consequently, we consider the edge with start node n_1 and target node n_a as *added edge* and add it to the list of dangerous edges. If both algorithms fail, the non-coinciding node $n_{P'}$ is simply marked as modified. In general, if we cannot find a counterpart for a specific node $n_{P'}$ in a method m and if subsequent comparisons also fail to detect corresponding nodes before the maximum lookahead value is reached, we stop the analysis and mark $n_{P'}$ as modified. The analysis continues with the comparison of other methods that have not been analyzed yet. The remaining nodes in m stay without analysis. This is no problem as these nodes cannot be reached without traversing the already detected change of node $n_{P'}$. So we cannot miss to select tests for re-execution that are affected by a change.

Apart from the Parallel Search and the Breadth-First Search algorithms, we can additionally infer on the kind of modification by means of logic and

combinatorics. In line 32 for example, we handle such – rather technical and maybe secondary – special cases for edges e' in P' that have no counterpart (provided that the target node of the edge has not been marked as visited yet and that the target node is not a glue node). As these edges only exist in the new program version, we can consider them as added. That is, either the target node or the start node of the edge have been added. We analyze this in a helper method `handleSpecialCases` (see the updated Algorithm 5 that shows the lines 31-34 of the original Algorithm 1). Similarly, if we cannot find a counterpart of the edge e in P (see line 16), we can consider it as removed and we add the edge to the set of dangerous edges.

```
   input  : s: start node in P; s': start node in P'
   output: d: List of dangerous edges
15 if e' is empty then
16 |  handleMissingEdges(e);
17 |  continue;
18 end

31 foreach edge e' leaving n' without counterpart do
32 |  handleSpecialCases(e');
33 |  d ← d.add(e');
34 end
```

Algorithm 5: Comparing two EJIGs - Handle special cases.

Our main reason for employing a Breadth-First Search rather than any other search algorithm is that we expect just a few nodes to be inserted or removed from an EJIG. In general, we do not assume the solution of our search to be far from the last coinciding node in the EJIGs. Of course, our assumption might be wrong in some cases. But in these cases, an analysis of a large number of successor nodes requires overly much time which deteriorates the effectiveness of our approach compared with a retest-all approach. For this reason, it is better to stop the search for a coinciding node when reaching a certain lookahead value.

Our RTS technique does not necessarily need a lookahead. If the lookahead value is set to 0, it does not analyze additional subsequent parts of the code. As soon as the lookahead is applied, our algorithm implies extra time which results in a longer runtime of the entire analysis compared to standard approaches without a lookahead. The additional runtime rises with the lookahead value. Especially a big lookahead leads to an increasing complexity. In particular, this affects the BD-precision levels because a method has usually many possible successor nodes as there are various calls to other methods. In Figure 5.3b for example, all the outgoing edges from the grey nodes could be method calls. In our evaluation, we therefore use various lookaheads to investigate their impact on memory consumption and to find a useful configuration.

5.5 Tool Implementation

We have implemented our RTS technique as extension of our Eclipse plug-in GWTTestSelection to support an easy and quick usage in the daily development process.

The version of GWTTestSelection used for the evaluation of the compiler-independent approach has some additional modules. One of them performs the Java source code instrumentation. To do this, the Eclipse JDT is used to parse the Java code and to insert CIDs as instrumentation code into the Java source code. Another module implements the functionality for the built-in logging server. It offers several interfaces. One of them enables test cases to activate/deactivate the logging server automatically. Optionally, the logging server can be started/stopped manually in the Eclipse plug-in or by calling a script. In order to call these interfaces, additional commands are inserted into the source code of test cases. These commands might differ depending on the kind of test and the test framework in use, but the interface of the logging server always remains the same. For efficiency reasons, we buffer a certain amount of CIDs on client side while running the web tests. The buffered data are transformed into JSON in order to send them to the logging server. The server itself buffers the CIDs received by the instrumented web application once more for efficiency reasons and writes them to a database[2]. Our tool is completely independent from any tools suitable to create UI/web tests (e.g. Selenium [251] or TestComplete [258]). The extended Eclipse plug-in offers a big settings menu that can be invoked directly via the settings button, or alternatively via the Eclipse Preferences menu. Here, the user can choose between two static precision levels and our dynamic heuristics in order to define code instrumentation granularity as well as the analysis granularity. All analyses can be combined with arbitrary settings for lookaheads (see Figure 5.4). The result of the analysis is displayed in two tables in the Eclipse plug-in (see Figure 5.5). In the table on the left, the code changes are listed. For an easy access, the user can double click on the table entries in order to jump directly to the corresponding code in the Eclipse code editor view. Additionally, we have introduced a new bookmark. That is, the user obtains an overview of the code changes via the Eclipse Bookmark menu. The table on the right shows for each code change in the table on the left the corresponding test cases that might be affected and that should be re-executed. So, GWTTestSelection is a great assistance in the bug localization and bug fixing process.

5.6 Evaluation: Compiler-Independent Approach

In order to assess our solutions to provide an efficient selective regression testing technique for transcompiled cross-platform applications, we discuss our approach in terms of six research questions:

[2]The database is not built-in and has to be setup by the user.

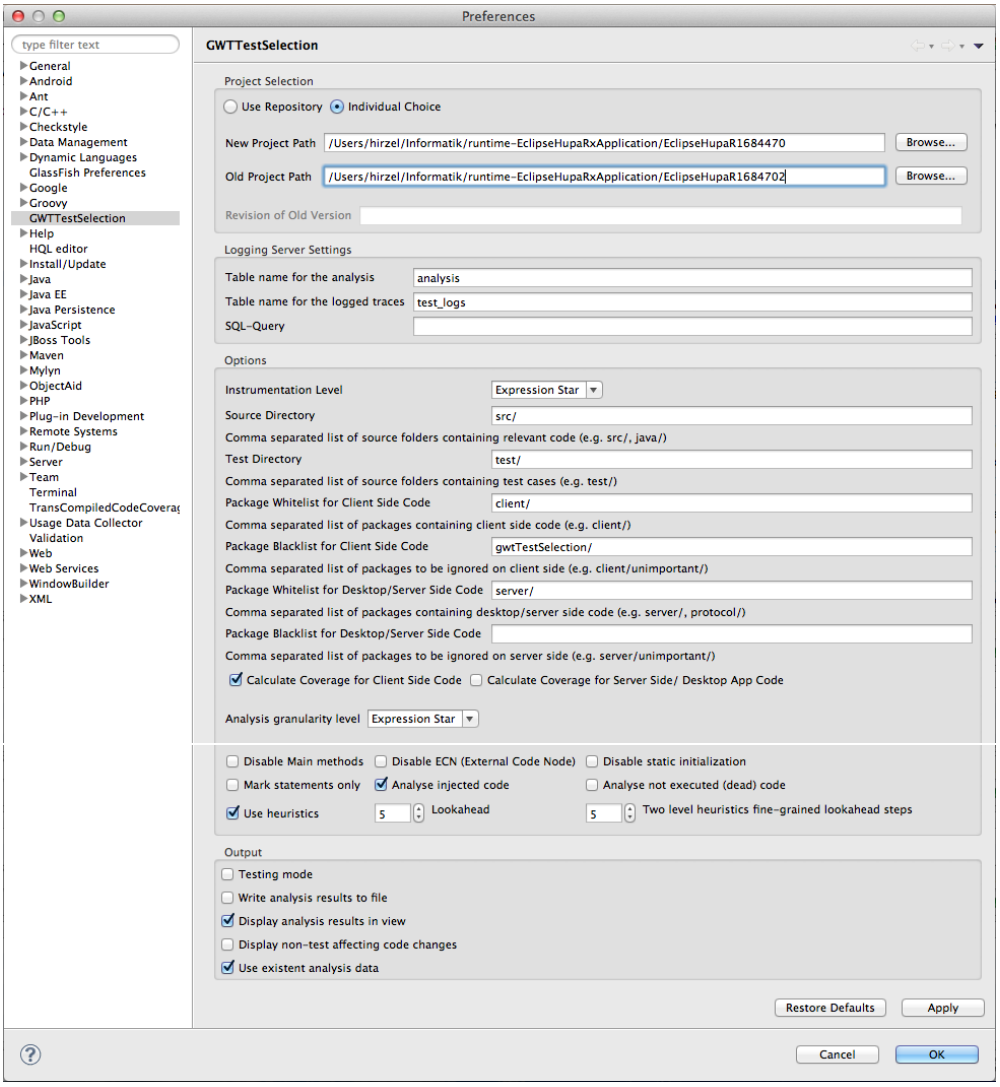


Figure 5.4: Options of Eclipse plug-in GWTTestSelection.

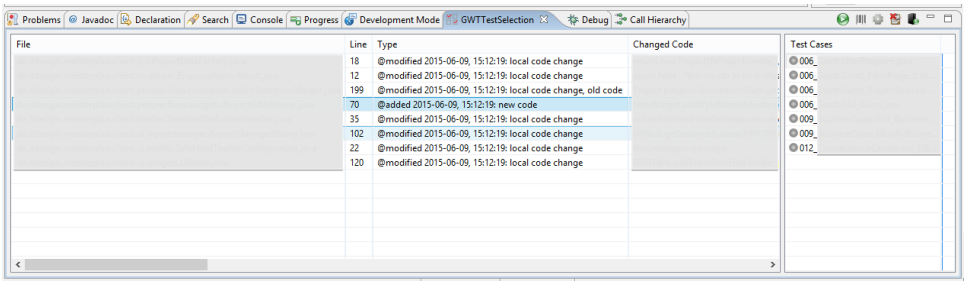


Figure 5.5: Eclipse plug-in GWTTestSelection.

RQ1 How accurate can code changes be categorized as added, modified and removed?

RQ2 To which extent will a fine-grained source code instrumentation and analysis take more time compared to a coarse-grained one? What does it mean for memory consumption?

RQ3 In which sense will lookaheads affect the analysis runtime and the detection of code changes?

RQ4 Can a dynamically adaptable, on a heuristics based analysis level outperform a static, user-predefined analysis level?

RQ5 How many tests are selected with our approach and how much differ the results from a retest-all approach?

RQ6 Is our technique cost-effective compared to a retest-all approach?

5.6.1 Software under Evaluation

In our study, we focus on two web applications created with GWT and its transcompiler. We reuse the open source mail client Hupa [144] (see Section 4.8.1). For the analysis of Hupa, we have checked out the source code of a series of revisions from the public repository. At the time of the investigation, the latest version was revision number 1684702. It consists of approximately 40.000 non-empty lines of code (NLOC) in 979 classes and interfaces. (Please note that the revision numbers are not consecutive.)

In order to apply and assess our approach thoroughly, we have additionally chosen an industrial application as second experimental object. Meisterplan [150] is a highly dynamic and interactive resource and project portfolio planning software for executives. Projects are visualized as Gantt diagrams. To each project, data like the number of employees, their hourly rates and their spent time may be assigned. Meisterplan accumulates allocation data from the different projects and creates histograms. Additionally, the existing capacity is intersected with the allocation data. This way, bottlenecks in capacity become visible. It enables the user to optimize the resource planing by either delaying a project or redistributing resources. Changes in capacity, project prioritization or strategy can be simulated by drag and drop. Dependencies between projects are visualized with the aid of arrows. To enhance project and cost analyses, views and filters are provided.

The last version of the source code of Meisterplan that we have investigated consists of approximately 170.000 non-empty lines of code (without imports) in roughly 2300 classes and interfaces. The test suite comprises 105 web tests. The software is built and deployed using Maven. This process and the entire testing is part of continuous integration using Jenkins[3]. All web

[3] `https://jenkins-ci.org/`

tests are created with the aid of TestComplete [258], an automated testing platform.

To evaluate these two applications, we have considered 13 pairs of versions of Hupa and 21 pairs of Meisterplan. Each of these pairs have been evaluated with various settings. In total, we have conducted 272 analyses.

5.6.2 Experimental Setup

As already explained in Section 4.8.2, ancient revisions of Hupa contain a large number of changes. This is contrary to the usual way of doing small increments that are regression tested afterwards. Besides, there are merge conflicts in many revisions. The choice of our start revision respects these obstacles. In total, we have selected six revisions of the Hupa repository including the most recent one to do this evaluation. In order to get more reliable data, we have asked a college to do error seeding. This way, four additional versions have been created. (As they do not compile any more, we use them for localizing faults.) Another four versions have been implemented by ourselves. In order to guarantee real conditions, we have extracted some changes from ancient Hupa revisions.

Our Hupa web test suite comprises 32 web tests created with Selenium. Unfortunately, the developers of Hupa do not provide any own web tests. For this reason, we have asked another college to create web tests. He has never seen Hupa or its source code so far. Again, we have created some additional ones.

The developers of Meisterplan maintain an own web test suite. We have selected revisions used for the nightly retest-all approach and the corresponding web tests to do our evaluation. As we would like to integrate our approach in the continuous integration process, we have additionally selected revisions committed during the day to investigate how our approach performs in this situation.

The evaluation of Meisterplan has been performed on an Intel Xenon 3.2 GHz with 8 GB RAM. The Eclipse settings allowed a maximum heap size of 6 GB. For Hupa, we have used an Intel Core i5 2.4 GHz with 8 GB RAM.

5.6.3 Threats to Validity

For getting reliable and representative results, our evaluation has to take several conditions into account. We discuss these conditions and how we have tried to heed them.

External Threats to Validity

In order to be able to judge the performance and the efficiency of our approach, it is essential to investigate large applications created with a transcompiler. However, it is difficult to find open source applications of a suitable size. In addition, open source applications usually do not provide any UI tests. For this

reason, we are grateful having the opportunity to investigate Meisterplan. It is a big industrial application based on a transcompiler. We had access to real revisions and a big web test suite. In addition, we investigate Hupa as representative of a mid-sized application. When combining these tools, they fulfill our demands to be able to judge the performance and the efficiency of our approach appropriately.

When considering the performance and the efficiency of our approach, the comparability could be threatened by the amount of (background) processes performed by the testing machine. As already mentioned, running all web tests in the Meisterplan test suite takes 9 hours. So we had to run the tests for the 22 revisions on several days after rebooting the test system. For this reason, we always tried to ensure that the same processes were running and that only relevant programs were executing.

Internal Threats to Validity

The implementation of our approach could maybe contain a fault. Of course, we have checked the behavior by applying the tool to small example projects and by comparing the results to our expectations. Nevertheless, in large projects such as Meisterplan, there might arise special cases due to the interaction of many complex program states. For this reason, we have always inspected our results for abnormalities manually.

Threats to Construct Validity

Our approach introduces a lookahead value in order to find additional faults in the source code. To this end, the approach employs two algorithms: Parallel Search and Breadth-First Search. Especially when using rather large lookahead values, it could happen that one of the algorithms detects a node that corresponds to the searched one accidentally. Such a scenario might arise when an expression appears several times in a method for some reason. For this reason, we have investigated our approach on the one hand with various lookahead values. On the other hand, we have always manually inspected whether the faults identified by our tool contain false positives. Besides, this procedure has enabled us to observe whether our tool missed a fault.

Programming languages frequently offer syntactical simplifications like omitted brackets in if- or loop-statements. This impairs the ability to insert instrumentation code without changing the semantics of the source code. For this reason, we require the usage of blocks in if- and loop-statements. However, this is no hard restriction as modern IDEs offer to automatically add missing blocks on save.

5.6.4 Results

In the following subsections, we discuss the results of our evaluation in terms of our research questions.

RQ1: Accurate Categorization of Code Changes as Added, Modified, or Removed

Our tool has categorized all the code changes as added, modified, or removed. We have observed some cases where the changes have been labeled more generally as modified instead of as added or removed. This is due to the combination of the lookahead value and the number of code entities affected by a modification. For example, if very much additional statements have been inserted in a method, the Breadth-First Search fails to find a location in the code where the common comparison can be resumed. So, as explained in Section 5.4.4, we mark the code as modified. However, it has never happened that added code has been labeled completely wrong as removed or vice versa. So in general, the combination of code change categorization and extra meta information about how the code looked like before/after it has been changed helps the developer to understand modifications without an additional DIFF-based tool.

RQ2: Time and Memory Consumption of a Fine-Grained Instrumentation/Analysis Compared with a Coarse-Grained Variant

Figure 5.6 and Figure 5.7 show the runtime needed to analyze the version pairs of HUPA and MEISTERPLAN, respectively. As we have 13 pairs for HUPA and 21 pairs for MEISTERPLAN, we use box plots to report on the results. The horizontal axis represents the parameter settings used in the different analysis methods. We have used the same analysis methods for both applications. The precision level E* has been investigated with lookahead values 20, 10, 5, and 1. We denote this precision level with the different lookahead values as E* L20, E* L10, E* L5, and E* L1, respectively. The BD precision level has been tested with lookahead values 5 and 1. We use as abbreviations BD L5 and BD L1. Apart from this, we have considered 2 heuristics. The first one is called E*-BD L5-5. It tries to balance the lookaheads in the E* and the BD level and sets both values to 5. The second one – E*-BD L10-1 – considers the extremes and defines lookahead = 10 for E* and lookahead = 1 for BD. Due to lack of space in tables or figures, we will also use the term H L5-5 instead of E*-BD L5-5 and H L10-1 instead of E*-BD L10-1, where "H" is a shortcut for heuristics.

The horizontal line of the box plots represents the median. The boxes below/above the median contain the lower/upper 25% of the values. The vertical lines (whiskers) at both ends of a box show the other values ignoring possible outliers.

Our results show that there are only little differences in runtimes. Due to its medium size, HUPA shows the same median in almost all settings except for E* L5 and BD L1. In these precision levels, HUPA shows slightly better median values. When looking at the runtime of MEISTERPLAN, E* L10 and BD L5 perform slightly better than the other static precision levels. When setting the outliers to the median, we have learned from a variance analysis that there are significant differences ($p = 5\%, X^2 = 56.06, df = 27$). However, a subsequent

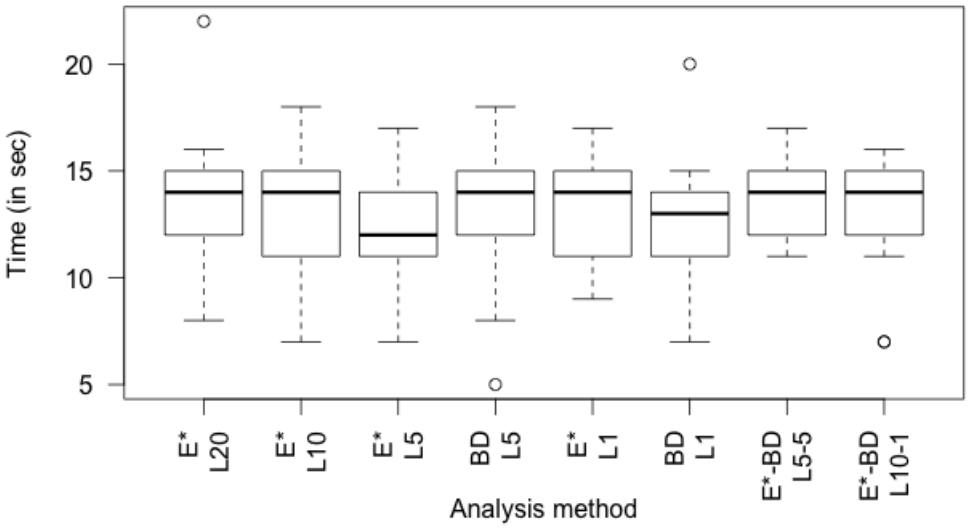


Figure 5.6: Test duration Hupa.

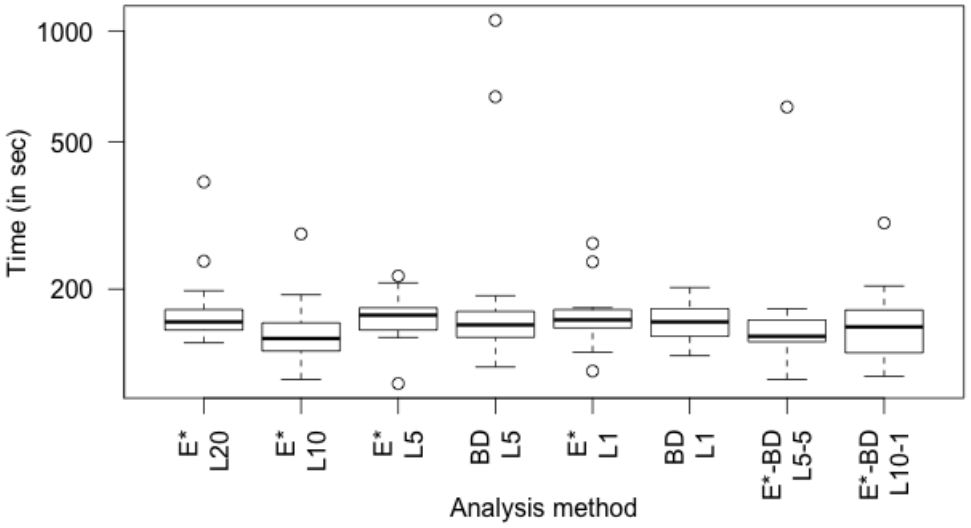


Figure 5.7: Test duration Meisterplan.

t-test has shown that only E*-BD L5-5 shows a significant better runtime compared to E* L5. This is somewhat contradictory to our expectations. When there is enough internal memory, a fine-grained analysis does not provide any disadvantages. Nevertheless, when analyzing Meisterplan on E*-level, each EJIG requires 5,5 times more nodes than at the BD-level in the latest version. Considering Hupa, there are still 3,6 times more nodes in the E*-level.

As far as RQ2 is concerend, a more detailed analysis is no problem as long as there is enough memory available. The similar runtimes are a result of the necessity to traverse the Eclipse AST even on BD-level as described in

	Settings (Precision level and Lookaheads)			
Versions	E* L1	E* L5	E* L10	E* L20
v2v1	28	28	28	28
v3v2	1	1	1	1
v4v3	16	20	20	21
v5v4	6	12	15	15
v6v5	1	3	3	3
v7v6	9	9	9	9
v8v6	11	11	11	11
v9v6	16	16	22	25
v10v6	10	11	11	11
v11v2	22	25	25	25
v12v2	16	17	17	20
v13v2	46	85	85	88
v14v2	10	10	10	10

Table 5.1: Number of CIDs affected by code modifications in HUPA for E* precision level.

Section 5.4.1. Nodes with many call edges slow down the analysis as many edges have to be checked in order to chose the appropriate path to continue the analysis. A more coarse-grained analysis takes effect not before considering the memory consumption. Here, large numbers of nodes can be handled more easily.

RQ3: Affect of Lookaheads on Analysis Runtime and Code Change Detection

Table 5.1 shows the number of CIDs that are affected by code modifications in HUPA for the E* precision level with different lookahead values. The first column shows the versions that have been compared. For example, v2v1 denotes the comparison of version 2 with version 1. The remaining columns show the settings used in the comparisons. Our findings indicate that the number of detected code modifications rises with the lookahead, which is desirable in order to reduce the effort for subsequent analyses.

In addition, we have noticed that the lookahead value should not be selected to large as it might happen that our algorithm detects nodes which coincide accidentally. This might for example happen when an expression re-emerges in the CFG, but it is also true for lookaheads used in an analysis on BD-level. Here, we might for example think of helper methods that are called from multiple methods. In these cases, it seems that two coinciding nodes n_P and $n_{P'}$ have been found and that the analysis can continue from these nodes. In actual fact, the successor nodes of n_P and $n_{P'}$ already do not coincide any more. So, if there is a fundamental code change, a large lookahead value

might result in recovering nodes although there is no common node in P and P' available. Consequently, we could stop the comparison earlier in order to reduce the runtime, but recovering coinciding nodes prevents the algorithm to detect that. We have repeatedly observed situations like this during our experiments to find a suitable lookahead for the BD-level when using big values (lookahead ≥ 10).

Apart from that, during our analysis, we have observed outliers in the time required to analyze the EJIGs when the lookahead value has been set to a big value. Examples are E* L20 or BD L5 (see Figure 5.7). Repeating the same analyses have confirmed that it did not happen by accident. Consequently, memory was at a critical point. The reason is that in cases with big lookaheads, the number of nodes we have to keep on the stack might increase exponentially. Therefore, the memory usage rises. Let us consider the example in Figure 5.3b again. If we would set the lookahead to the value 1, we would have to keep three nodes (n_b, n_c, n_d) on the stack when analyzing the successor nodes of n_a. (Please note that n_a has already been removed from the stack.) Setting the lookahead value to 2, we would also have to analyze the nodes $n_e, \ldots, n_i, n_2$. That is, after adding the successor of n_d on the stack, it would contain six nodes. In general, if a node represents a method declaration, it might have more outgoing edges than a statement. This is because method declarations often have many call edges. For this reason, the memory usage for an analysis on BD-level usually rises faster with an increasing lookahead value than for an analysis on statement level. Due to this fact, we have used in our evaluation a lower maximum lookahead for analyses on BD-level than on E*-level.

Thus, with respect to RQ3, setting the lookahead value has to be done with deliberation. However, a higher lookahead can be beneficial for the detection of code changes.

RQ4: Benefits of a Dynamically Adaptable Analysis Level Based on a Heuristics

When choosing the settings for our two heuristics, we have tried to take into account the results of our static precision levels and the experiences we made with large lookahead values. As already described, HUPA has the best median runtime values for the settings E* L5 and BD L1 (see Figure 5.6). MEISTERPLAN performs the best for E* L10 and BD L5 (see Figure 5.7). So, a mixture of these settings have been the most promising.

When comparing our heuristics against each other, the balanced one (E*-BD L5-5) performs a bit better than E*-BD L10-1. Besides, it is clearly better than the static variant E* L5. So, the settings E*-BD L5-5 could be the heuristics of choice. In general however, we can see that our heuristics provide no significant runtime improvement compared to static analysis levels. At first sight, it does not matter whether a static precision level or a heuristics is applied. But on second sight, it becomes evident that especially E*-BD L5-5 unifies in many cases the best results when looking at the test selection (see e.g. v7v5 in Table 5.3) or at the runtime. Beyond that, it offers

Versions	Settings (Precision levels and Lookaheads)								
	E* L10	E* L5	BD L5	E* L1	BD L1	E* L20	H L5-5	H L10-1	Exp
v2v1	0%	0%	0%	0%	0%	0%	0%	0%	0%
v3v2	0%	0%	47%	0%	47%	0%	0%	0%	0%
v4v3	97%	97%	97%	97%	97%	97%	97%	97%	6%
v5v4	97%	97%	97%	97%	97%	97%	97%	59%	0%
v6v5	9%	9%	13%	9%	13%	9%	9%	9%	0%
v7v6	0%	0%	0%	0%	0%	0%	0%	0%	fault loc.
v8v6	97%	97%	47%	97%	47%	97%	97%	97%	fault loc.
v9v6	44%	44%	44%	44%	44%	44%	44%	44%	fault loc.
v10v6	97%	97%	97%	97%	97%	97%	97%	97%	fault loc.
v11v2	9%	9%	97%	9%	97%	9%	9%	9%	9%
v12v2	97%	97%	97%	97%	97%	97%	97%	97%	3%
v13v2	97%	97%	97%	97%	97%	97%	97%	97%	0%
v14v2	97%	97%	97%	97%	97%	97%	97%	97%	0%

Table 5.2: Test selection HUPA.

a tradeoff in memory consumption. Considering the latest version of MEISTERPLAN, it is theoretically necessary to keep approximately 145 000 nodes per EJIG in the internal memory. As described in Section 5.4.3, the BD-level gains a lot from a lower analysis overhead. This leads to a significant reduction of nodes. In case of MEISTERPLAN, it is a factor of up to 5,5. In case of HUPA, it is still a factor of up to 3,6. Our heuristics benefits from this as it guarantees a precise analysis, but reduces in the best case the costs for the analysis to $C_{H_{analysis}} = \frac{1}{5,5} \cdot C_{E*_{analysis}}$ in case of MEISTERPLAN, and to $C_{H_{analysis}} = \frac{1}{3,6} \cdot C_{E*_{analysis}}$ in case of HUPA.

Thus, the main advantage of our heuristics with respect to RQ4 is the reduction of nodes during the analysis which in turn reduces the memory consumption. So we can conclude that a heuristics outperforms a static analysis level.

RQ5: Ability of Test Selection to Reduce the Test Effort

Table 5.2 and Table 5.3 show how many tests are selected for re-execution. Each row represents a pair of versions with eight possible settings. The first column shows which versions have been compared. (For example, v2v1 denotes the comparison of version 2 with version 1.) The value in the last column (Exp) indicates our expectations how many tests should be selected for re-execution. We have obtained these values from real test executions by looking up from the corresponding test reports which tests actually failed. (Please note that in Table 5.2, four versions have been used for fault localization only as mentioned above in Section 5.6.2.)

Versions	Settings (Precision levels and Lookaheads)								
	E* L10	E* L5	BD L5	E* L1	BD L1	E* L20	H L5-5	H L10-1	Exp
v2v1	100%	100%	100%	100%	100%	100%	100%	100%	21%
v3v2	100%	100%	100%	100%	100%	100%	100%	100%	4%
v4v3	6%	6%	6%	6%	6%	6%	10%	6%	3%
v5v4	100%	100%	100%	100%	100%	100%	100%	100%	0%
v6v5	62%	0%	0%	0%	0%	0%	0%	9%	0%
v7v5	9%	12%	9%	9%	9%	29%	9%	9%	1%
v8v3	100%	100%	100%	100%	100%	100%	100%	100%	6%
v9v8	99%	19%	19%	25%	19%	28%	19%	99%	0%
v10v8	99%	99%	99%	19%	99%	99%	99%	99%	0%
v11v10	100%	100%	100%	55%	55%	100%	100%	60%	0%
v12v10	100%	100%	100%	100%	100%	100%	100%	100%	0%
v13v12	51%	51%	100%	51%	100%	100%	100%	100%	0%
v14v8	99%	99%	99%	99%	99%	99%	99%	99%	0%
v15v14	100%	100%	0%	100%	0%	100%	0%	0%	1%
v16v15	100%	100%	100%	99%	100%	99%	99%	99%	0%
v17v15	100%	100%	100%	100%	100%	100%	100%	0%	1%
v18v17	100%	100%	96%	100%	96%	100%	96%	100%	0%
v19v18	100%	100%	100%	100%	100%	100%	100%	100%	0%
v20v19	100%	100%	100%	100%	100%	100%	100%	100%	0%
v21v20	97%	97%	97%	100%	97%	97%	97%	100%	0%
v22v21	99%	99%	99%	99%	99%	99%	99%	99%	0%

Table 5.3: Test selection Meisterplan.

Our findings show that the BD-level sometimes select more tests than the E*-level as expected (see e.g. Hupa, v3v2, BD L5 and BD L1 or Meisterplan, v13v12, BD L5 and BD L1). However, we have also observed some outliers. For example, the comparison v17v15 in Meisterplan shows 0% for H L10-1. This is due to Java Heap Space Error. The same is true for v15v14 when analyzing the BD-levels. This emphasizes the need for a memory-saving approach.

There are cases, in which only a small subset of the test suite is selected for re-execution. This is especially true for the Meisterplan versions v4v3, v6v5 and v7v5 which have been committed by the developers during a single day. Hupa also has versions which do not need to be retested with all of the web tests in the test suite (see e.g. v6v5 and v11v2). Nevertheless, there are also many cases, in which almost all tests are selected for re-execution. Here, it becomes evident that web tests are more complex than unit tests due to side-effects on other code. In many cases, we have observed that only a few modifications are responsible for selecting almost all web tests for re-execution. The most crucial factor is whether the effect of a code change can be stemmed

on a small subset of tests. However, if the change affects the initialization phase of the application, this is very hard.

Conversely, it may always be true that specific parts of the code are executed by each UI/web test for some reason (e.g. application initialization). As soon as a piece of code in these specific parts has been modified, all UI/web tests have to be re-executed. Therefore, a solution might be to select only one of these test cases as representative to get a quick feedback whether this special test execution already results in an error. In fact, this kind of deliberations belong to already existing test prioritization or minimization techniques. But of course, the validity of such a simplification is weak. The very first question is which test should be selected as representative. Beyond that, another question is how to proceed if the execution of a representative does not fail. It might happen that this is due to a special state of the application and that one of the omitted tests would reveal a fault due to a different state of the application. Most importantly, a solution that reduces the test results artificially is not safe any longer. We come back to these questions later on.

In the end, our technique decreases the testing effort by up to 100% (see for example v2v1 in Hupa or v6v5 in Meisterplan) compared to a retest-all approach. But in many cases there is even no improvement. On average, our technique has been able to reduce the test suite by 25% in case of Meisterplan (H L10-1). In case of Hupa, the test suite reduction has even been 46% on average (H L10-1). So our evaluation shows some parallels to the evaluation of Graves et al. [119] and Rothermel and Harrold [234] (see Section 5.2). However, we have made some progress in reducing the effort and the memory consumption with the aid of our heuristics and the lookahead. Besides, we have shown that the technique has potential to be applicable even in the context of UI/web tests. The crucial factors for reducing the test selection are the structure of the application and the code modifications. Consequently, we have to adopt additional measures to avoid that our RTS technique is executed in vain.

RQ6: Cost Efficiency of Test Selection Compared to Retest-All

To find a response to RQ6, we have to look at the overall costs which depend on the application itself. We want to recall the equation of Leung and White [178, page 205] (see Section 5.4.3):

$$C_s(T_s) < [C_e(T_o) - C_e(T_s)] + [C_c(T_o) - C_c(T_s)]$$
$$C_s(T_s) + C_e(T_s) + C_c(T_s) < C_e(T_o) + C_c(T_o)$$

In practice, is it difficult to ascertain values for $C_c(T_s)$ and $C_c(T_o)$. According to Leung and White, $C_c(T_o)$ represents the costs for checking whether the old tests used for program version P fit the expected results. Accordingly, $C_c(T_s)$ represents the same with regard to T_s, where T_s contains a subset of the tests in T_o plus new tests that check new functionality in P' [178]. However, neither the test suite of Meisterplan nor the test suite of Hupa has changed during our evaluation: There have been no additional tests, and no tests have been

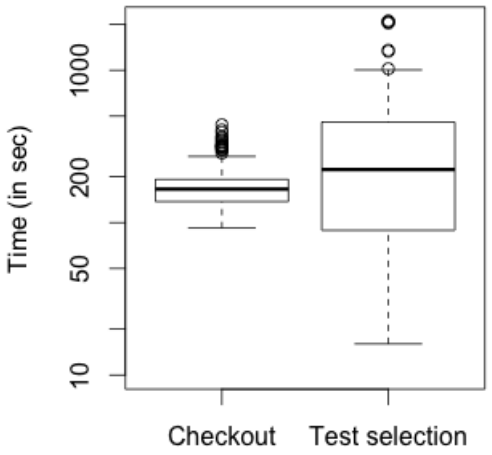


Figure 5.8: Checkout and test selection time MEISTERPLAN.

removed from the test suites. So it is always $T_s \subseteq T_o$. That is, when comparing our RTS approach with the retest-all approach in terms of costs for checking the test results, our RTS approach is always cheaper or at most equally expensive. But more important, today's test tools (such as SELENIUM or TESTCOMPLETE) automatically check whether a test fits the expected results. So we do not expect the costs $C_c(T_s)$ and $C_c(T_o)$ to be decisive for assessing the cost efficiency of our RTS approach. Consequently, we do not consider them further and simplify the equation as follows (see also Section 5.4.3):

$$C_s(T_s) + C_e(T_s) < C_e(T_o)$$
$$C_{X_{analysis}} + C_{instr} + C_{transcomp_\Delta} + C_{log} < C_e(T_o)$$

where $X_{analysis}$ represents the granularity of the analysis.

MEISTERPLAN takes 4:30 minutes to instrument the code on E*-level. The delta for transcompiling the code is less than 1 minute. Executing the web tests and logging the CIDs has an overhead of 90 minutes. That is, on average, each test takes 51 seconds longer due to logging overhead. Comparing the current version with a previous one requires a checkout. Figure 5.8 shows the resulting boxplot for MEISTERPLAN. According to this, the median for a checkout is 166 seconds. The analysis of P and P' with our heuristics E*-BD L5-5 takes additional 149 seconds (which is 2 seconds slower as E* L10). Finally, the test selection requires 223 seconds (median value, see Figure 5.8). In total, when applying our approach to MEISTERPLAN, the extra effort is 14:28 minutes (4:30 minutes + 1 minute + 166 seconds + 149 seconds + 223 seconds) for doing the analysis plus 90 extra minutes for executing the tests and logging the CIDs. So, the total extra time is 104:28 minutes. As the retest-all approach takes 9 hours, a single test takes on average 5:09 minutes. In contrast, when looking at our RTS technique, the total time for running tests and doing all the additional tasks is nearly 10:45 hours. Naturally, in order to be efficient, our

approach must not exceed the 9 hours (540 minutes) and consequently, it has to decrease the amount of tests selected for re-execution. The total time for the extra effort (instrumentation, delta for transcompiling, checkout, analysis, and test selection) is fixed at 14:28 minutes. So we have only 525:32 minutes left. Each MEISTERPLAN test takes in our approach on average 6 minutes (5:09 minutes + 51 seconds runtime overhead due to logging instrumentation code). So we can execute 87,59 tests at the maximum without exceeding the time required for the retest-all approach. Thus, our approach should decrease the amount of tests selected for re-execution by 18 tests ($\approx 17\%$). Consequently, our approach is efficient for the versions v4v3, v6v5, v7v5, and v9v8.

Please note that instrumenting code and logging CIDs are initial tasks that are done once when creating and checking test cases (see Section 4.4.3). Ideally, only a few tests have to be re-executed and thus, it is sufficient to log CIDs for these tests only. The CIDs for non-affected tests remain untouched. So these initial tasks do not have to be repeated as a whole. This reduces the effort for applying our RTS approach in subsequent analyses.

Considering HUPA, we have to deal with the following values. The instrumentation takes 40 seconds. To finish testing and logging, 6 additional minutes are necessary. Transcompiling requires less than 10 extra seconds. A checkout takes 40 seconds. The median for analyzing the code with our heuristics E*-BD L5-5 is 14 seconds. Finally, the time for selecting tests that have to be re-executed is less than 2 seconds. So, the extra effort when applying our approach to HUPA is 1:46 minute plus 6 extra minutes for executing all the tests and logging the CIDs. So each test needs additional 11,25 seconds. The retest-all approach takes in total 9:13 minutes. So, a single test requires 17,28 seconds. As the total time for the extra effort is always 1:46 minutes, we have only 7:27 minutes left. Each HUPA test takes in our approach on average 29 seconds (17,28 seconds + 11,25 seconds runtime overhead due to logging instrumentation code). So in the end, we can run 15 tests without exceeding the runtime of the retest-all approach. In total, our approach should decrease the test suite by 17 tests ($\approx 53\%$). Consequently, our approach is efficient for the versions v2v1, v3v2, v6v5, v7v6, v9v6, and v11v2.

Summing up the time required for checkout, comparison of P and P' and test selection, we are done in less than one minute. This enables us to overcome the common nightly build and test cycle towards a fast executable and repeatable cycle of code changing, test determining and bug localization.

As we can see, HUPA's test suite reduction must be greater than MEISTERPLAN's test suite reduction. This has nothing to do with the size of the application. The reason is the test setup of HUPA: When using our test suite, HUPA does not require loading any settings or databases. For this reason, the usual test execution can proceed immediately whereas the instrumented execution has to make sure that the CIDs have been memorized. This extra time prevents that the next test execution can start immediately after the previous test has been finished. Due to this delay, we have a rather high extra time for running the tests and for logging. This is the main reason why our approach

should decrease the test suite by 53%. In case of MEISTERPLAN, memorizing CIDs does not deteriorate the result as there is always a preparation phase for setting up the database and for establishing a specific application state before starting the actual test. Here, it becomes apparent that especially large applications with a big test suite gain from our approach.

In general, regarding RQ6, our technique can be efficient both in medium sized applications with small test suites and in big systems with large test suites as long as changes do not affect all tests.

5.6.5 Discussion

Code Modifications and Test Selection: Our evaluation shows that our technique is able to reduce the testing effort. However, the approach has to be refined in order to deal even with those situations when the test selection is not able to reduce the test suite. In a first step, we have optimized the test selection. At the beginning of our experiment, some of the results have even been worse. It has turned out that this was due to modifications in fields. As soon as a web test traversed the constructor, the CID representing the modified field has been executed. This has resulted directly in a test selection regardless of whether the field has been unused. Now, we consider field modifications only if their value is really used in methods executed by a test.

Memorizing CIDs in the Database: In RQ6, it has turned out that the efficiency suffers from memorizing CIDs in the database. In order to be more efficient, an expedient could be to start a new instance of our new logging server on another port. This way, the next test can start immediately and we do not have to wait until the CIDs of the previous test have been memorized.

Costs for Logging Process, Test Selection and Nature of Web Tests: According to our evaluation, another cost factor is the logging process. (Please remind that in MEISTEPLAN, a test execution takes approximately 51 seconds extra to run the instrumented tests). Unfortunately, our heuristics is not able to reduce these costs as it is $C_{H_{traces}} = C_{E^*{}_{traces}}$. Consequently, using solely the BD-level might improve the efficiency. However, an analysis on BD-level tends to select more tests as shown in our evaluation. So, a trade-off could be to do the fine-grained analysis on statement level rather than on E*-level. This would speed up the analysis a little. Nevertheless, even this trade-off might result in a higher test selection than an analysis on E*-level. For example, an analysis on statement level cannot distinguish changes in conditional expressions. So it is neither a satisfying solution for these cases in which most or even all of the tests are selected for re-execution.

As already mentioned before (see Section 5.6.4), we have observed many situations where only a few changes have been responsible for selecting most of the tests. Of course, it is possible to some degree to optimize tests in such a way that they cover as much distinct code as possible. But web tests are different

from unit tests. They do not test a small unit, but run many functions to setup the application and to perform standard actions. So it is hard to completely avoid that tests cover the same functionality. For this reason, we have to search for other solutions. The test selection remains the most relevant cost factor.

5.7 Conclusion and Future Work

In our evaluation, we have seen that our compiler-independent approach is able to identify precisely all the code changes in the source programming language. Besides, it selects exactly those UI/web tests that are affected by the code changes.

However, regression testing transcompiled applications using UI/web tests takes much time. When looking solely at the time required for detecting code changes, a more coarse-grained analysis is beneficial. It reduces the time-effort for instrumenting code and logging CIDs. Besides, less internal memory is required for holding the CFGs for P and P'. Nevertheless, there is a risk that more tests are selected for re-execution which in turn deteriorates the efficiency of our RTS technqiue. Our evaluation shows that we have found an efficient trade-off between a low memory consumption and the ability to localize code changes exactly with the aid of a heuristics. It reduces the memory consumption significantly with a low risk to select more tests than a purely fine-grained analysis level would do. This is essentially for outperforming the retest-all approach and helps to execute tests in an efficient way (see Section 1.2.1). Furthermore, it supports the developer in the bug fixing process. Lookaheads improve the efficiency additionally as we are able to find more code changes in a single analysis. So, we have managed to refine the fault localization process. At best, no subsequent errors show up when re-executing the selected tests. In addition, followup analyses like change impact analysis are unnecessary. Information about the kind of code changes contribute to solve the fault localization problem (see Section 1.2.2).

Nevertheless, according to our evaluation, our heuristics-based RTS technique (H L5-5) is still not able to outperform the retest-all approach in 70,6% of the comparisons. The results of our evaluation have revealed a big dependency on the kind of code change. Our approach is efficient as long as modifications do not affect the whole web application and if UI/web tests focus on a specific functionality rather than testing many different and unrelated features. This finding is of course also valid in other contexts (e.g. desktop applications or mobile applications) as well. Consequently, we have not solved the test effort reduction problem (see Section 1.2.1) in a satisfactory way. For this reason, we have to take further actions so that applying the RTS technique is not in vain even when many tests are selected for re-execution.

Chapter 6

Prioritizing Regression Tests based on the Execution Frequency of Modified Code

6.1 Introduction

In Chapter 4 and Chapter 5, we have seen that there are sometimes cases where regression test selection techniques select many or even all tests for re-execution. Possible reasons include a large number of changes in the code base or a single code change that has a big impact on the application. We have observed in the previous chapters that the latter often occurs in the context of end-to-end testing with web tests. Due to their special nature, they often do not test mostly disjoint functions as unit tests usually do, but might execute parts of the client-side code repeatedly. Of course, it is very unsatisfactory when there is no or only a small test suite reduction. In these cases, using regression test selection may be even more expensive than simply re-executing all tests because additional time for code instrumentation, logging, and analyzing program versions is necessary.

In order to still have an advantage over the retest-all approach, test prioritization is an expedient. As explained in Section 2.2, test prioritization does not reduce the number of tests to be re-executed, but tries to re-order them in such a way that fault-revealing tests are executed first (e.g. [71, 290]). This is a challenging task: The information which tests reveal a fault is not available before re-executing the tests [290]. So no one knows in advance which is the optimal test case ordering. But combining regression test selection with test case prioritization promises additional benefits: As already described by Do et al. [58], starting the test execution with those tests that have the highest chance to reveal faults gives fast feedback. Besides, it is possible to stop the test execution due to time constraints while being aware that the most important tests have been executed [58]. Thus, as noticed by others before (e.g. [4, 77, 142, 153, 191, 192, 216, 217, 264, 284, 289]), it is possible to enjoy the

best of both worlds by combining the two techniques. Initially, there is the safe regression test selection. Adding test prioritization results in an optimized test execution order. If necessary, re-executing the selected tests can be stopped. Of course, this might sacrifice safety as we might miss to execute tests that reveal faults not detected by other tests yet.

In Section 6.2, we point out related work and already existing approaches. There, we also discuss weaknesses of these approaches. Afterwards in Section 6.3, we explain challenges of test case prioritization in more detail. In Section 6.4, we present our novel test case prioritization technique that is based on the execution frequency of modified code entities (functions, statements, or even expressions) covered by test cases, and remark main differences to other approaches. In brief, we prioritize those tests that execute the (most) code changes as often as possible (execution count of modified code). To the best of our knowledge, we are the first to base the execution order of tests on this decision criterion. Additionally, we introduce a dynamic feedback mechanism to adjust the order of tests at runtime. In the same section, we propose three static and three dynamic ways to implement our technique. Each implementation uses a differing algorithm. We provide a formalization and explain the algorithm in examples. In order to judge the efficacy and performance, we investigate the different realizations in an evaluation in Section 6.5 and compare them to already existing approaches in the literature. Finally, we discuss the results in Section 6.6 and conclude in Section 6.7.

This Chapter is based on our publication "Prioritizing Regression Tests for Desktop and Web-Applications Based on the Execution Frequency of Modified Code" [136][1].

6.2 Related Work and Weaknesses of Existing Approaches

There are many different properties that might be considered when trying to find a suitable test prioritization order that runs fault-revealing tests as soon as possible. For this reason, researchers have come up with a wide range of definitions of prioritization criteria seeking to sort tests in an optimal way. Yoo and Harman [290] have presented a survey that covers the full diversity of existing approaches. We focus on the most similar approaches compared to ours.

One possible aspect that has been considered in the past is to incorporate data from a test history (e.g. [71, 161, 163]). Elbaum et al. [71] for example consider as criterion the test performance in previous test runs by investigating the rate of executed fault-prone functions. For this purpose, they introduce a fault index representing the fault-proneness of every function in the program. The index is updated after every test execution. In order to find a test-ordering,

[1] My own contributions to the publication [136] are as follows: Scientific ideas: 85%; Data generation: 100%; Analysis and interpretation: 85%; Paper writing: 70%.

the fault indexes of all the functions covered by a test are put together. The test with the highest sum of fault indexes has the highest priority.

Kim and Porter [163] have also considered history-based test prioritization techniques. Their implementations take different kinds of data into account that result from previous test runs. In the first implementation, test cases that have been executed rarely in the past will be prioritized in the next test run. Another idea (realized by Kim and Porter in a second implementation) is to give these tests a higher chance to get selected early for re-execution that have revealed faults in the past. Finally, the third implementation of prioritizes tests that cover functions which have not been executed in the past test runs.

Apart from historical data, structural code coverage (e.g. [126, 290]) has been used as criterion in many approaches (e.g. [56, 71, 74, 235, 237]). Here, tests are usually ordered according to the number of covered functions, blocks, or statements. For simplicity, we subsume functions, blocks, and statements as code elements. Tests covering the most code elements have the highest priority (Greedy algorithm, see Paragraph "Test Case Prioritization" in Section 2.2). Rothermel et al. [235] as well as Elbaum et al. [71, 74] propose several techniques considering the coverage of statements or functions. The authors refer to them as "total statement coverage prioritization" (e.g. [71, page 104], [235, page 181]) and "total function coverage prioritization" (e.g. [71, page 105]). In addition, Do et al. [56] consider coverage of blocks. They call it "total block coverage prioritization" [56, page 42]. We follow Yoo and Harman and refer to these kinds of coverage prioritization as *total approaches* [290, pages 87, 88]. Moreover, Rothermel et al. [235] and Elbaum et al. [71, 74] present techniques estimating the probability that a test exposes faults. Finally, Elbaum et al. [71, 74] additionally introduce a technique that tries to estimate the probability that any fault exists.

In order to improve prioritization results, some strategies assume that similar tests are negligible. There are many properties imaginable according to which tests could be considered as similar. Some approaches iteratively reduce the priority of tests that cover the same code elements as other tests which have already been executed. Rothermel et al. [235] as well as Elbaum et al. [71, 74] have presented refined versions of their prioritization techniques. Therein, the authors additionally take into account whether statements or functions have already been covered by a previous test. They start with the test that covers the most statements or functions. Then, the authors incorporate these coverage data in the decision which of the remaining tests should run next. That is, they try to find a test that traverses the most statements or functions not covered by a previous test yet. Do et al. [56] have pursued the same idea with blocks. The authors call these variants "additional statement coverage prioritization" ([71, page 104], [235, page 181]), "additional function coverage prioritization" [71, page 105], and "additional block coverage prioritization" [56, page 42], respectively. In the past, this strategy has also been used in related approaches (e.g. [192, 217]). In some papers (e.g. [192]), it is also called "feedback mechanism" [192, page 278]. We follow Yoo and Harman and subsume them as

additional approaches [290, pages 87, 88] or – accordingly – we denote them as *feedback* approaches. Other strategies investigate the sequences of covered code elements and their (dis)similarity in order to find a single test as representative for a group of related tests (e.g. [80, 176]). More details about this kind of strategy follow at the end of this section.

Elbaum et al. [75] go one step further and propose two strategies to select the most effective prioritization technique. To this end, they consider five prioritization techniques. The first two techniques (available as *total-* and *additional* approach, see above) are based on the coverage of functions. The next two techniques (available as *total-* and *additional* approach) rely on the coverage of functions that differ. The last technique prioritizes tests randomly. The basic strategy of Elbaum et al. just relies on APFD values (see Section 2.2, Paragraph "Test Case Prioritization"). The second strategy tries to incorporate specifics of the current program version under test, the code changes, and the corresponding test cases.

We also use coverage in our novel test case prioritization, but we incorporate this information only when our main decider, the execution frequency of code entities, fails to provide an unambiguous prioritization order. We will explain more details in Section 6.4. The *additional* approach [290, pages 87, 88] or *feedback* mechanism [192, page 278] has inspired us to develop a solution that greedily refines the prioritization order. As opposed to other approaches, our mechanism is completely dynamic as it modifies the prioritization order on the fly at runtime.

Test prioritization and test selection are well-suited to each other, even though test prioritization does not aim at reducing the number of executed tests. As already mentioned, there are some situations (large number of code changes, change with big impact on the application, test structure and -design) when it is hard to reduce a test suite at all. In consequence, different techniques combining (elements of) RTS and TCP have been proposed in the literature (e.g. [4, 77, 142, 153, 191, 192, 216, 217, 264, 284, 289]). Similar to RTS techniques, these approaches first try to detect changed code elements. Afterwards, tests are ordered according to a prioritization strategy, e.g. the number of changes covered by a test. We summarize some approaches in more detail:

Sampath et al. [246] have formalized three different ways to combine different prioritization strategies if a single strategy is not able to determine a clear test prioritization order due to ambiguities. They call these ways (a) "Rank", (b) "Merge", and (c) "Choice" [246, page 1327]. The first one applies different prioritization strategies according to a predefined order. "Merge" combines several strategies at the same time. Finally, "Choice" decides via a selection function which strategy provides the best result. Based on these definitions, our own prioritization technique relies on "Rank" to combine different prioritization strategies.

There are several papers that use code change information for test prioritization as we do. One of the first approaches have been proposed by Wong et al. [283]. There, the authors have investigated the effect of a minimization

technique on the ability to detect faults. They have detected significant advantages with respect to the size of the test set and the effectiveness. Later, Wong et al. [284] have combined modification-based test selection with a minimization technique and a prioritization technique. The prioritization arranges tests according to coverage in descending order.

Huang et al. [142] have presented a technique for selecting and prioritizing tests in binary Java applications. They use an extension of the tool `Javap` that belongs to the JDK to disassemble the code. Afterwards, they investigate methods for changes. Based on these data, they prioritize test that are affected by changes. For the prioritization order, they take several criteria into account: These include how many changed methods a test covers, how often a changed method is executed by a test, how many methods a test covers, and the time that a test is expected to take for its execution. Their approach has some similarities to our approach, but it relies on changes in methods, not on changes in single code entities. Besides, they give higher priority to tests that cover more changes. In contrast, we select tests first that cover changes more often.

Aggrawal et al. [4] apply a version specific test selection to obtain information about changed lines of code. The test prioritization prefers tests that cover the most changed lines of code. Mirarab and Tahvildari [191] also use code change information and consider additionally coverage metrics and software quality metrics to create a bayesian network. This model serves as a basis to decide upon a prioritization order. Beyond that, the same authors describe in another paper [192] an *additional* approach (see above). Zhao et al. [296] present a hybrid form that additionally takes a code-coverage-based clustering approach into account to detect similarities in test cases in order to reduce their priority.

Panigrahi and Mall [216] also combine a regression testing technique with prioritization. They use an extended system dependency graph and apply forward slicing in order to determine code changes on statement-level in a subsequent program version. For each change, they write the modified statement itself and the line numbers of both the original and new program version in a file. In addition, they instrument blocks in the code of the original program with `print`-statements. Based on these data, they select all the tests that cover code changes. When re-executing tests, they start with the test covering the most code changes. Later, Panigrahi and Mall have refined their approach further [217]. They still use the extended system dependency graph-based RTS technique to determine code changes and they still prioritize the tests according to the number of covered changes in the first place. But for determining the final execution order, they assign weights to changes. For each change that has already been executed by a tests, the weight is reduced by a fixed value. Tests with the highest sum of weights are preferred. This prioritization approach is contrary to ours. We assume that testing a piece of code thoroughly with many different tests is done by intention (see Section 6.4). Apart from the approach of Panigrahi and Mall [217], a very similar approach has also been presented by Srivastava and Thiagarajan [264] before. They use a binary matching tool

called BMAT to identify changed blocks. Then, they assign weights to each test according to the overall number of modified blocks covered by the test. After finishing the test, the weights are recalculated. As opposed to Panigrahi and Mall, already executed blocks are no longer considered as modified. Jeffrey and Gupta [153] rely on statement/branch coverage and additionally apply relevant slicing to obtain a prioritization order.

In contrast to our work, none of the approaches considers the execution frequency of a code change to cover as much application states as possible, which will – according to our assumption – increase the possibility to detect a fault.

Test prioritization has also been applied to web services (e.g. [17, 37]) as well as to web applications (e.g. [143, 245]). Sampath et al. [245] prioritize test cases – among others – according to the test length, according to the frequency a sequence of web pages is accessed in a test, and randomly. Moreover, Chen et al. and Huang et al. have seized on the idea to combine test selection and test prioritization. Huang et al. [143] check method changes in Java bytecode (see Section 4.2) and prioritize tests according to the risk to fail. Again, none of the authors do consider changes and the frequency with which these changes are executed by a web test.

To the best of our knowledge, execution frequency has almost never been considered as coverage measure. Only the approach of Fang et al. [80] is close to ours as they also apply the execution frequency of code entities. However, they start with the test that covers the most code elements. They use the execution count to order sequences of program entities. Based on these sequences, they apply an edit distance function to calculate the similarity of remaining tests. In the end, most dissimilar tests are executed first. Similarly, Leon and Podgurski [176] also use dissimilarity metrics to run those tests first that are the most different. As opposed to our approach, they do not use a RTS technique to identify code modifications and do not consider the execution frequency of changed code as decider on which tests should run with high priority. Moreover, we regard similar tests as an indicator of fault-prone code and do not lower their priority.

Epitropakis et al. [79] present a solution to incorporate three different prioritizing criteria to achieve multi-objective regression test case prioritization. The criteria incorporate the capability to detect faults, the coverage of modified code, and the coverage of faults in the past. This is somewhat similar to our approach as we also consider many different criteria for prioritizing test cases. However, our criteria are different as we use the accumulated execution frequencies of traversed code modifications. Besides, we do not consider criteria like the ability of a test to detect faults by investigating its previous test runs.

Walcott et al. [281] also rely on multiple prioritization criteria. In a first analysis, they prioritize tests according to a specific time constraint. The results serve as input for a subsequent calculation that determines a fitness-value by means of code coverage.

Yoo and Harman [289] propose a “multi-objective formulation of the re-

gression test case selection problem" [289, page 140]. They investigate two approaches to find an optimal subset of the original test suite that a) covers the most code without exceeding a certain time limit and b) requires the least time to achieve a specific code coverage. In the first approach, they incorporate code coverage and execution cost. The second approach additionally uses information about the ability to detect faults in previous test runs.

The prioritization techniques described above exhibit different strengths and weaknesses. A high structural coverage (of blocks/methods) does not necessarily correlate with the ability of a test to detect a fault. Furthermore, approaches prioritizing tests according to the number of traversed code modifications have the risk that a unique prioritization order is impossible because all tests are affected by the same fundamental code changes. This may especially be true in the context of web applications. Finally, approaches based on the similarity of tests assume that tests reveal the same faults if a given similarity criterion matches. These approaches just pick a representative of a class of similar tests. However, tests that appear similar still might explore different parts of the application's state space and thus can result in different test outcome.

In our opinion, existing approaches insufficiently consider recent code changes and possibly fault-revealing states. However, the application's state space is crucial for the ability to detect faults. It is therefore decisive to put the software in a fault-revealing state. We hypothesize that similar tests have been developed intentionally as they check different states of the application. We also hypothesize that code modifications often lead to a change of the application state. Based on this assumptions, we conclude that one should prioritize tests which run modified code the most often as these tests have the highest chance to detect a fault.

6.3 Motivation

Test prioritization seeks to find faults as soon as possible [290]. To achieve this, the goal is to find an optimal ordering of tests revealing these faults. The example in Figure 6.1 shows tests and the faults they are able to reveal. Tests $t_A - t_E$ have been taken from Elbaum et al. [71, page 106]. To explain our motivation, we have added two more tests t_F and t_G. Elbaum et al. argue that an ordering starting with the execution of t_C followed by t_E (order $t_C t_E$) is superior because it detects all the faults the most rapidly.

In practice, the problem is that no one knows in advance which test will detect a fault [290]. Even when using a test selection technique, we only have knowledge about code modifications. We do not know yet which modification leads to a test failure. Thus, which decider recognizes that $t_C t_E$ is the best choice?

Furthermore, the order $t_C t_E$ might not be optimal when there are several possible solutions. Taking test t_F into account, order $t_B t_F t_E$ covers the same faults. When searching for a decider which of the two possible solutions should be applied, some approaches use the number of faults previously detected and/or the number of statements covered by the tests. However in

		Fault revealed by test case									
		1	2	3	4	5	6	7	8	9	10
Test	A	×				×					
	B	×				×	×	×			
	C	×	×	×	×	×	×	×			
	D					×					
	E								×	×	×
	F		×	×	×						
	G					×					

Figure 6.1: Fault revealing tests, adapted from Elbaum et al. [71, page 106].

```
1 public boolean validate(String s) {
2   s = s.trim();
3   return s.matches("[a-zA-Z]*");
4 }
```

Figure 6.2: Validator for user inputs with marked additions in version P' compared to P.

practice, examples can be found for which this strategy is not optimal. As a simple example, Figure 6.2 shows a code snippet of a program in two different versions P and P'. The code snippet validates user inputs, for example in a mail search dialog. In the subsequent version P', valid user inputs have been restricted by adding a call to `trim()` (see the statement in line 2), which might cause tests to fail.

Imagine that test t_G validates several possible search items that have been inserted into the search box of a mail client. Another test t_D takes a search item in order to check whether the mail body will be displayed correctly when double clicking on a search result. Both test cases t_D and t_G cover the same code change that has been added erroneously (see line 2 in Figure 6.2). Test t_D covers more statements as it also checks the double click in the search results. Nevertheless, it is conceivable that t_G has a higher chance to fail and to reveal this fault because it checks more possible inputs. So it is more reasonable to prioritize test case t_G rather than t_D.

A similar problem occurs when a fundamental code modification affects (almost) all test cases, as we have seen in the previous chapters. In this situation, (almost) all tests are affected by the same single code change. So it does not suffice to consider the number of covered faults per test case. The number of executed statements might be misleading as described previously. Besides, we found that using historical data did not correlate with current code changes in one application. For this reason, we propose an approach considering the number of traversed code entities per test case.

6.4 Approach

Our approach builds on the assumption that the chance to detect a fault is greater when a code modification is executed in many different application states. In general, there are many reasons why a code entity, such as a statement, is being executed multiple times. For instance it might (a) be part of a loop which itself is executed many times or (b) be enclosed in a function that is called multiple times by either the application itself or by tests. Hence, we propose to take the *execution frequency* of code modifications into account when prioritizing tests cases.

While a test executes a code entity several times, it might change parts of the application state or use different parameters. This can reveal faults that only show up in certain application states.

In the same sense, we also assume that a different test which partially executes the same code modification as other tests does this intentionally and therefore also has potential to reveal a fault. Thus, in our approach we favor tests that execute more code modifications than others.

In the remainder of this section, we explain our approach in detail. We present various variants that incorporate the number of code modifications and their execution frequency in different ways. For each of the variants we propose an *additional* approach, incorporating a dynamic *feedback* mechanism, that uses knowledge about test failures during test execution. This allows to dynamically adjust the test prioritization and reevaluate made decisions.

6.4.1 Considering Execution Frequency of Modified Code

To perform prioritization, our technique always considers two versions of a program P and P'. For our purposes, we use the CIDs (see Section 4.4.3) and the trace of a test. Please remind that CIDs represent unique identifier assigned to every source code entity. In Java this can be for instance methods, statements, or even expressions leading to a different granularity of the analysis.

Further, we specify the term *test trace* (see Section 4.3.1 and Section 4.4.3) and define it to be a multi set of CIDs that are traversed during the execution of the test. The order of the entities is not important for our approach, however, we count for each entity in the trace how often it has been traversed by a test.

Our approach is divided into two steps. The first step performs regression test selection and thereby tries to reduce the set of tests we have to consider for prioritization. At the same time, it provides the necessary information required for step two: For every test, it provides its corresponding test trace. And it expounds which CIDs in the trace correspond to source code modifications. The second step then performs the actual prioritization.

Step 1 – Regression Test Selection

This first step is independent of the choice of RTS technique. Many different techniques have been studied in the past. Graph walk-based approaches,

Test	Execution Frequency						
	c_1^Δ	c_2^Δ	c_3^Δ	c_4^Δ	c_5^Δ	c_6	c_7
t_1	4	0	4	2	0	0	1
t_2	0	4	3	0	1	7	0
t_3	2	1	4	0	2	2	1
t_4	2	5	0	1	0	0	0

(a) Matrix f of execution frequencies per test and CID.

Test	Metrics			
	$\texttt{sum}^\Delta$	$\texttt{count}^\Delta$	$\texttt{max}^\Delta$	`count`
t_1	10	3	4	4
t_2	8	3	0	4
t_3	9	4	4	6
t_4	8	3	5	3

(b) Result of the test metrics.

Figure 6.3: Example matrix f and resulting metrics for several tests covering the same CIDs with $n = 4, m = 5, p = 7$.

slicing-, and firewall-based approaches are only a few examples. More details can be found for instance in the survey of Yoo and Harman [290]. Based on our discussion in Section 2.2 (see Paragraph "Discussing the Approaches"), we apply our graph walk-based RTS technique that we have explained in Section 4.3.2. At every code entity, the application source code is instrumented to report the corresponding CID to a central logging server that inserts the CIDs into a database. This enables us to create traces for a test, identifying all the code entities which are executed by the test. As our RTS technique is completely generic, our prioritization technique will be applicable for instance to both desktop and web applications.

Figure 6.3a shows an output of the first phase as a matrix f which we will use as a running example. Every entry f_{ij} in the matrix represents how often code-entity c_j has been traversed while executing test t_i. We will refer to an entry f_{ij} as *execution frequency*. CIDs representing code changes are marked with Δ. Thus, c_6 and c_7 denote CIDs that do not represent a code change. (Explanations on Figure 6.3b follow in Section 6.4.2.)

Step 2 – Prioritization

The second step of our approach computes a prioritization of the tests in form of a strict total ordering of the tests, using the matrix of execution frequencies as its sole input. After establishing necessary preliminaries, we propose multiple prioritization techniques and give their formal definitions.

$$
\begin{aligned}
\texttt{sum}_i^{\Delta} &= \sum_{1 \le j \le m} f_{ij} \\
\texttt{count}_i &= \sum_{1 \le j \le p} \min(f_{ij}, 1) \\
\texttt{count}_i^{\Delta} &= \sum_{1 \le j \le m} \min(f_{ij}, 1) \\
\texttt{max}_i^{\Delta} &= \max_{1 \le j \le m} \left(f_{ij} * \left\lfloor \frac{f_{ij}}{\max(\max_{1 \le k \le n} (f_{kj}), 1)} \right\rfloor \right) \\
\texttt{id}_i &= i
\end{aligned}
$$

Figure 6.4: Metrics assigning each test t_i an integer value as an estimate for its importance.

Let T be a test suite or – more formally – a set of test cases, let n be the number of test cases in T, p be the total number of CIDs and m to be the number of CIDs that correspond to changes. Without loss of generality, we assume CIDs c_j with $1 \le j \le m$ to represent code changes and CIDs with $j > m$ and $j \le p$ to represent unchanged code entities. For each test t_i and each CID c_j we record the frequency of t_i executing c_j in the matrix f and reference the execution frequency f_{ij} by indexing.

Each of our priorization techniques is based on a series of *test metrics*. A test metric M assigns each test t_i a corresponding integer value as an estimate of the chance of t_i to reveal a fault. Figure 6.4 gives five such metrics ($\texttt{sum}_i^{\Delta}$, $\texttt{count}_i$, $\texttt{count}_i^{\Delta}$, $\texttt{max}_i^{\Delta}$, and $\texttt{id}_i$) that will be explained in more detail when used to define our prioritization techniques.

Definition 1 (Lifting of Metrics)**.** Given a metric M we define a corresponding order $t_i \lesssim_M t_l$ on tests t_i and t_l by

$$t_i \lesssim_M t_l \quad \text{iff} \quad M_i \le M_l$$

■

That is, to compare two tests we pointwise compare the corresponding integer values yielded by the metric. The resulting relation $\lesssim_M$ is a total preorder[2] and induces an equivalence relation for tests according to metric M:

$$t_i \sim_M t_l \quad \text{iff} \quad t_i \lesssim_M t_l \ \wedge \ t_i \gtrsim_M t_l$$

We also refer to the total preorder on tests $\lesssim_M$ induced by a metric M as *prioritization criterion* (PC) and use the name of the metric also for the prioritization criterion (that is, the preorder) when it is clear from the context.

[2]Since a metric might assign the same value to two distinct tests, $\lesssim_M$ is not antisymmetric.

For instance, we use $\texttt{sum}^{\Delta}$ to refer to the preorder $(T, \lesssim_{\texttt{sum}^{\Delta}})$ and to refer to the metric for a test t_i as $\texttt{sum}^{\Delta}_i$.

In general, a metric M might assign the same integer values to two or more distinct tests resulting in no strict order for the tests in the corresponding equivalence class $\sim_M$. To allow a more fine-grained ordering of tests we introduce the hierarchical composition for two total preorders PC_1 and PC_2.

Definition 2 (Hierarchical Composition)**.** For every two tests t_i and t_l the hierarchically composed order $PC_{1\triangleright 2} = PC_1 \triangleright PC_2$ is defined by:

$$t_i \lesssim_{PC_{1\triangleright 2}} t_l \quad \text{iff} \quad \begin{array}{l} (t_i \lesssim_{PC_1} t_l) \vee \\ (t_i \sim_{PC_1} t_l) \wedge (t_i \lesssim_{PC_2} t_l) \end{array}$$

■

That is, two tests can either strictly be ordered by PC_1 or if they are equivalent with regard to PC_1 they are (not strictly) ordered by PC_2. The hierarchical composition of two total preorders is again a total preorder and hence hierarchical composition can be applied recursively. In addition, the hierarchical composition is associative.

Finally, given the matrix of execution frequencies, a *prioritization technique* uses a series of PCs to create a suggested ordering of execution of the tests in the test suite T. A strict total order is achieved by terminating the sequence of hierarchical compositions with the metrics `id`. Thus the scheme for defining some X-frequency-based prioritization technique using k prioritization criteria is:

$$X\text{FP} = PC_1 \triangleright PC_2 \triangleright \cdots \triangleright PC_k \triangleright \texttt{id}$$

6.4.2 Global Frequency-based Prioritization Technique (GFP)

The first prioritization technique of *global frequency-based prioritization* (*GFP*) builds on the assumption that tests which execute the most code changes are the most likely to reveal a fault. To this end, GFP applies $\texttt{sum}^{\Delta}$ which simply accumulates the execution frequencies of all code changes covered by a test t_i (PC_1). The definition of the corresponding metric can be found along the other metrics in Figure 6.4. Tests with a higher accumulation value (global frequency) have a higher priority than tests with a lower global frequency.

To find an ordering of tests with the same global frequency, $\texttt{count}^{\Delta}$ is applied. Here, the frequency is ignored and it is only counted how many code changes have been executed by a test t_i, hence modeling a coverage of changes (PC_2). A test covering more changes has higher priority. Please note again that PC_2 only affects tests whose order is not unambiguous, yet. The same applies to all subsequent prioritization criteria.

		PC_1		PC_2		PC_3		PC_4
CFP	=	$\texttt{count}^{\Delta}$	$\triangleright$	$\texttt{sum}^{\Delta}$	$\triangleright$	count	$\triangleright$	id
LFP	=	$\texttt{max}^{\Delta}$	$\triangleright$	$\texttt{count}^{\Delta}$	$\triangleright$	count	$\triangleright$	id
GFP	=	$\texttt{sum}^{\Delta}$	$\triangleright$	$\texttt{count}^{\Delta}$	$\triangleright$	count	$\triangleright$	id

Table 6.1: Definitions of our different prioritization techniques.

If there are still ambiguities, `count` is applied to also include CIDs that do not correspond to code changes, hence modeling a general, classic code coverage metric (PC_3). Tests achieving a higher code coverage are executed earlier.

Finally, as for all of our prioritization techniques, if no ordering can be decided after applying the first three prioritization criteria, the last criterion `id` arranges the tests in the order they appear in the matrix of frequencies (PC_4).

The prioritization criteria that GFP uses are defined in Table 6.1.

Example: In the example of Figure 6.3a test t_1 covers three changes c_1^{Δ}, c_3^{Δ}, and c_4^{Δ}. Summing up the corresponding execution frequencies gives $\texttt{sum}_1^{\Delta} = 10$. All other tests in this example yield lower values as can be seen in Figure 6.3b and thus t_1 has the highest priority. Applying $\texttt{sum}^{\Delta}$ (PC_1) gives the following equivalence classes from high to low priority: $\{t_1\}, \{t_3\}, \{t_2, t_4\}$. We can notice that t_2 and t_4 cannot be distinguished according to PC_1, so following our definition of hierarchical composition, $\texttt{count}^{\Delta} \triangleright \texttt{count} \triangleright \texttt{id}$ will be applied to find an ordering for this equivalence class. The criterion $\texttt{count}^{\Delta}$ (PC_2) gives 3 for both t_2 and t_4, so again $\texttt{count} \triangleright \texttt{id}$ needs to be applied. Finally, PC_3 gives $\texttt{count}_2 = 4$ and $\texttt{count}_4 = 3$ leading to the strict ordering $\{t_1\}, \{t_3\}, \{t_2\}, \{t_4\}$.

If in the previous example t_2 and t_4 would have been equivalent according to PC_3 then `id` (PC_4) would give $\texttt{id}_2 = 2$ and $\texttt{id}_4 = 4$. In general, `id` always results in a stable and strict total ordering since every test has a unique and stable row index.

6.4.3 Local Frequency-based Prioritization Technique (LFP)

The *local frequency-based prioritization technique* (*LFP*) builds on the assumption that for every code change, there is an optimal test: The test that executes this code change the most often. To this end, for each code change c_j, this technique first selects the test t_i with the greatest execution frequency f_{ij} (local maximum, $\texttt{max}^{\Delta}$, PC_1). This selection mechanism is encoded numerically in the equation for $\texttt{max}_i^{\Delta}$ in Figure 6.4 by first dividing by the maximum of a

column j followed by rounding[3]. Tests are then prioritized in descending order of their corresponding maximum.

Similar to GFP, tests that are equivalent according to $\texttt{max}^{\Delta}$ are further ordered by $\texttt{count}^{\Delta}$ (PC_2), `count` (PC_3), and `id` (PC_4).

Example: For each code changes in Figure 6.3a we first determine the test with the highest frequency for that code change as: t_1 for c_1^{Δ} with $f_{11} = 4$, t_4 for c_2^{Δ} with $f_{42} = 5$, t_1 and t_3 for c_3^{Δ} with $f_{13} = f_{33} = 4$, t_1 for c_4^{Δ} with $f_{14} = 2$ and t_3 for c_5^{Δ} with $f_{35} = 2$. Selecting the maximum of these highest frequencies for each test results in the metric $\texttt{max}^{\Delta}$ in Figure 6.3b. In particular test t_2 never has a highest frequency for any of the code changes and is assigned $\texttt{max}_2^{\Delta} = 0$ in turn. Applying $\texttt{max}^{\Delta}$ gives the following equivalence classes $\{t_4\}, \{t_1, t_3\}, \{t_2\}$. To further compare t_1 and t_3 PC_2 gives $\texttt{count}_1^{\Delta} = 3$ and $\texttt{count}_3^{\Delta} = 4$ which leads to the strict ordering $\{t_4\}, \{t_3\}, \{t_1\}, \{t_2\}$. It is not necessary to apply PC_3 or PC_4 in this example.

6.4.4 Change Frequency-based Prioritization Technique (CFP)

The *change frequency-based prioritization technique* (*CFP*) prioritizes tests that execute the most changes. If two tests execute the same amount of changes it prefers the test that has a higher global frequency. It thus works in exactly the same way as GFP but swaps PC_1 and PC_2 in order to investigate whether global frequency or number of code changes tend to have a higher relevance in practice.

Example: To prioritize the tests in Figure 6.3a with CFP we first apply $\texttt{count}^{\Delta}$ to obtain $\{t_3\}, \{t_1, t_2, t_4\}$. Applying $\texttt{sum}^{\Delta}$ (PC_2) yields $\{t_3\}, \{t_1\}, \{t_2, t_4\}$ and `count` (PC_3) finally gives $\{t_3\}, \{t_1\}, \{t_2\}, \{t_4\}$.

6.4.5 Discussing Frequency-based Prioritization Techniques

As opposed to CFP, the techniques GFP and LFP imply that we might execute tests first that do not cover the maximum number of modifications. In Figure 6.3 for example, t_3 covers 4 code changes (c_1^{Δ}, c_2^{Δ}, c_3^{Δ}, c_5^{Δ}). All other tests cover less changes. Despite this fact, t_3 will not be executed first in GFP and LFP. However, this does not contradict the goal of test prioritization. As t_3 executes specific changes less often, it might fail to reveal a fault. In comparison, t_1 (in case of GFP) or t_4 (in case of LFP) respectively might set the application under test in more states and thus might be able to expose a fault. Besides, a test covering less code modifications might finish faster. So, there might be more time to run other tests.

Another implication of our GFP- and CFP-technique is that several highly prioritized tests might cover the same code changes. If a test has already failed,

[3]To avoid division by zero the denominator needs to be at least 1, whence $\max(..., 1)$.

another test covering the same faulty modification runs in vain. For this reason, we propose the following dynamic feedback mechanism.

6.4.6 Dynamic Feedback for Frequency-based Prioritization

In previous evaluations (e.g. [56, 74, 192, 237]), it became apparent that proposed prioritization techniques often performed better in conjunction with the *additional* approach [56, 74, 237], also known as *feedback* mechanism [192, page 278]. These kinds of techniques use knowledge gained during the current prioritization of tests and recursively include this information to re-prioritize the remaining tests. Thus, the final test execution order is not known before all tests have been executed.

Employing a similar strategy, we propose for the (static) X-frequency-based prioritization technique (XPF; see Section 6.4.1, step 2) a *dynamic* counterpart. It aims at lowering the priority of not yet executed test cases that will traverse one or several CIDs due to which previously executed tests have already failed. Thus, the dynamic X-frequency-based prioritization technique (DXPF) continuously adapts the test execution ordering at runtime by re-calculating prioritization criteria using information about test failures as input.

Initially, the dynamic technique employs static XPF to specify an ordering of the tests in the test suite T. After each test case execution, it checks the result of the test t that has just finished. If test t has succeeded, the next test in the prioritization order will be executed as usual. If test t has failed, the prioritization is re-evaluated. For this purpose, it checks which code changes have been shown to be fault-revealing. Their execution frequencies will be excluded from further re-prioritization. If DXPF cannot determine the exact code changes that has lead to a test failure, it ignores all CIDs that have been traversed by a failed test. Afterwards, XPF is applied to the remaining test cases resulting in a test suite T' with a new test prioritization order. The test system repeats this process until all tests have been executed or until another predefined stop criterion is fulfilled.

The dynamic feedback variant can be applied to any static (X)-frequency-based prioritization technique. Considering the above introduced strategies, we refer to the resulting variants as *Dynamic Global Frequency-based Prioritization Technique* (*DGFP*), *Dynamic Local Frequency-based Prioritization Technique* (*DLFP*), and *Dynamic Change Frequency-based Prioritization Technique* (*DCFP*), correspondingly.

Example: Let us illustrate the dynamic variant for GFP, DGFP by revisiting our example in Figure 6.3. This technique accumulates solely the execution frequencies of those code changes that have not been shown to be fault-revealing in any previously executed tests. Whenever a test fails during test execution, the accumulation has to be re-calculated for all tests not yet executed.

DGFP starts by prioritizing tests using GFP. As already showed in Section 6.4.2, this results in the ordering $\{t_1\}, \{t_3\}, \{t_2\}, \{t_4\}$. Let us assume

Test	Execution Frequency						
	c_1^Δ	c_2^Δ	c_3^Δ	c_4^Δ	c_5^Δ	c_6	c_7
t_2	0	4	3	0	1	7	0
t_3	2	1	4	0	2	2	1
t_4	2	5	0	1	0	0	0

(a) Revised matrix f of execution frequencies per test and CID. c_3^Δ will be ignored when calculating metrics, see Figure 6.5b.

Test	Metrics			
	`sum`$^\Delta$	`count`$^\Delta$	`max`$^\Delta$	`count`
t_2	5	2	0	3
t_3	5	3	2	5
t_4	8	3	5	3

(b) Revised result of the test metrics incorporating the failure of t_1 due to c_1^Δ.

Figure 6.5: Revised example matrix f and resulting metrics for DGFP after t_1 has been executed.

that t_1 fails due to c_3^Δ. This triggers a new re-prioritization of the remaining test cases t_2, t_3, t_4. During the recalculation, c_3^Δs execution frequencies f_{i3} $(1 \leq i \leq 4)$ will be ignored. Thus, we obtain new results of the test metrics, which can be found in Figure 6.5.

Figure 6.5a shows the revised matrix. As c_3 is excluded from further calculation of metrics, t_4 gives gives $\texttt{sum}_4^\Delta = 8$ when summing up the corresponding execution frequencies. All other remaining tests yield lower values as can be seen in Figure 6.5b. Applying `sum`$^\Delta$ (PC_1) gives the following equivalence classes: $\{t_4\}, \{t_2, t_3\}$. The criterion `count`$^\Delta$ (PC_2) gives 3 for t_3 and 2 for t_2. This already leads to the strict re-ordering $\{t_4\}, \{t_3\}, \{t_2\}$.

At some point, it might happen that there are only tests left that do not cover any code changes or whose execution frequencies are excluded. So in this case, there are no execution frequency data left and thus, the dynamic approach cannot further improve the execution order with the aid of PC1 or PC2. We would fall back to structural code coverage (PC3). For this reason, we cancel any further re-ordering and the last known prioritization order will be used to run the remaining tests. This way, a strict ordering is guaranteed.

6.5 Evaluation

In order to assess our solutions to provide an efficient selective regression testing technique for desktop and web applications, we have implemented our prioritization techniques as library. This enables us to discuss our approach in terms of the following three research questions:

RQ1 How do our techniques perform compared to standard coverage-based approaches [56, 71], bayesian network-approaches [191, 192], and similarity-based approaches [80]?

RQ2 Is a dynamically adapted execution order able to outperform a static prioritization order?

RQ3 Is our technique suitable to enrich a RTS technique in a cost effective way? How effective is the prioritization in terms of reducing test execution time?

6.5.1 Software under Evaluation

Previous test prioritization approaches often use software provided in the "Software-artifact Infrastructure Repository (SIR)" [55, 162]. It contains a wide range of applications in different versions that can be used to investigate new techniques. In order to be able to compare the results of our prioritization techniques with already existing test prioritization approaches, we utilize the SIR and choose three Java desktop applications as benchmark that have been frequently used in the literature: JMeter, JTopas, and XML-Security. Additionally, we enhance our study by two web applications created with Googe Web Toolkit (GWT), namely Hupa [144] and Meisterplan [150].

JMeter [275] is a load and performance tool to test web services and web applications. According to the SIR [55, 162], the most recent version contains 43.400 LOC in 389 classes. JTopas [30] is able to tokenize and parse text files or streams. It is the smallest tool in our evaluation and contains 5400 LOC in 50 classes as stated in the SIR [55, 162]. XML-Security [11] encompasses libraries supporting XML-signature syntax as well as XML-encryption syntax. According to the SIR [55, 162], it contains 16800 LOC in 143 classes.

As representatives of transcompiled cross-platform web applications, we reuse once more Hupa [144] and Meisterplan [150]. As already described in Section 5.6.1, Hupa comprises approximately 40.000 non-empty lines of code (NLOC) in 979 classes and interfaces. As Meisterplan is an industrial application with approximately 170.000 NLOC (without imports) in roughly 2300 classes and interfaces, it is very-well suited to serve as test object. It is the largest application in our evaluation.

6.5.2 Variables and Measures

During our evaluation, we have used different variable settings. We provide an overview below.

Independent Variables

Our evaluation depends on two independent variables: regression test selection technique that offers many different settings to analyze the software, and prioritization technique.

Regression test selection technique: As described in Section 6.4.1, our prioritization technique uses the results of a previous regression test selection step as input. Our regression test selection technique offers several parameters (see Section 5.4). The user can decide to analyze the software statically on a predefined analysis level or to use a heuristics which decides dynamically which analysis granularity of the code is the best one. Additionally, the user can define a lookahead value that enables a more in depth-analysis finding more code modifications.

For the evaluation, we have used a parameter settings which is the result of our findings in Section 5.6.4. It takes advantage of a heuristics that analyses modified code on the static analysis level *Expression Star* (= statement level that also considers e.g. conditional expressions) with a lookahead value 5. Unmodified code will be analyzed on body declaration level with the same lookahead value 5.

Prioritization technique: We investigate three different versions of our prioritization technique and additionally analyze for each of them the impact of a feedback mechanism. So in total, we consider six test prioritization techniques.

Dependent Variables

We evaluate the results of our prioritization technique on the basis of a metric, called Average of the Percentage of Faults Detected (APFD) [71] (see also Section 2.2, Paragraph "Test Case Prioritization") which is the standard metric in the literature. It measures the ability of an approach to detect faults as soon as possible [237].

Using the APFD metric, an objective comparison with previously published approaches is possible. This is even true when considering our dynamic feedback techniques. The APFD metric is applied only when the prioritization order does not change any more after all tests have been finished.

6.5.3 Experimental Setup

For each of the applications described in Section 6.5.1, several versions are available. The desktop applications contain seeded errors which are independent. Thus, we know for each code modification whether it is a fault. The web applications represent regular revisions taken from a repository with real errors (if there are any) plus some versions with seeded errors (as explained in Section 6.5.1). From our regression test selection, we know which code change(s) might be the cause of a test failure. However, in case of real, not seeded errors in Hupa and Meisterplan, we do not know exactly which code change is/ which combination of code changes are faulty in the end. So, in case of a test failure, we consider all changes covered by the failed test as faults. This way, we are able to compare our results.

Figure 6.6 provides an overview on the overall number of versions available per software, the aggregate amount of faults in all versions of a software, and the number of tests in the latest version. The evaluation has been performed on a Intel Core i5 2.4 GHz with 8 GB RAM.

	JMeter	JTopas	XML-Security
versions	6	4	4
faults	9	6	5
tests	78	128	83

	Hupa	Meisterplan
versions	4	6
changes considered as faults[4]	8[4]	136[4]
tests	32	106

Figure 6.6: Errors in software.

6.5.4 Threats to Validity

The results of our evaluation and our conclusions might be threatened by different factors. We will discuss these issues and how we have tried to minimize their effects.

External Threats to Validity

In general, the results might again depend on the size of the applications used for the evaluation. Besides, the kind of application (desktop application versus web application) might influence the ability to generalize the results. To limit these factors, we have studied both desktop applications and web applications of different size. As we also investigate an industrial application, we can benefit from real faults and real tests cases. However, in order to be able to compare our results with previous results in the literature, the range of possible applications have been predetermined to some degree by the selection of former state-of-the-art publications. Concerning the applications in the "Software-artifact Infrastructure Repository (SIR)" [55, 162], errors have been seeded manually, so these faults might differ from those emerging in industrial environments. Besides, we rely on the original software obtained from the SIR to preserve comparability without additionally seeding errors. However, some other authors additionally seeded errors.

[4]More details on considering code changes as faults follow in Section 6.5.4.

Internal Threats to Validity

Our instrumentation and our logging tool as well as our tool to determine the test prioritization could contain a fault. To exclude this threat as far as possible, we have executed our tools on a set of small programs and manually inspected the results.

Another threat to validity of our conclusions might arise out of the number of faults. Some applications contain seeded faults, some contain faults emerging during further development. Nevertheless, the number of faults has been rather small except of three versions in Meisterplan. We could imagine that a higher number of faults lead to additional or other insights. In case of the Meisterplan versions, the company could not tell us anymore which code modifications are actually faults. For this reason, we have considered all code changes covered by a failed test as faults, which might be wrong.

Finally, we have noticed that some of the test suites belonging to SIR-projects seem to have evolved slightly. Comparing the statistics of previous papers, the number of test-methods sometimes slightly differed (e.g. [56, 295]). For this reason, our results might be not completely comparable with results obtained from other techniques. Nevertheless, the differences are small, so we still get a good indication on how our technique performs.

Threats to construct validity

We use the APFD metric that measures the weighted average percentage of detected faults. Elbaum et al. [73] introduce a "cost-cognizant" [73, page 331] version of the APFD metric that considers the severity of faults as well as costs for test cases. Especially the test costs (i.e. time) play an important role in our case, too. However, as all necessary data (mainly code instrumentation and creation of test execution logs) can be obtained from the previous regression test selection, our ordering algorithm takes only a few seconds. Besides, as we do not have data on the severity (e.g. time to correct faults or damages caused by a fault) and – again – in order to be able to compare our results with approaches in the literature, we use the APFD metric.

6.5.5 Results

In the following subsections, we discuss the results of our evaluation in terms of our research questions.

RQ1: Comparing Performance of our Technique with existing Approaches

Figure 6.7 and Figure 6.8 show the APFD values obtained during our evaluation. We use box plots to report on the results of each software under test. The horizontal line within the box plots represents the median. The boxes below/above the median contain the lower/upper 25% of the APFD values. The

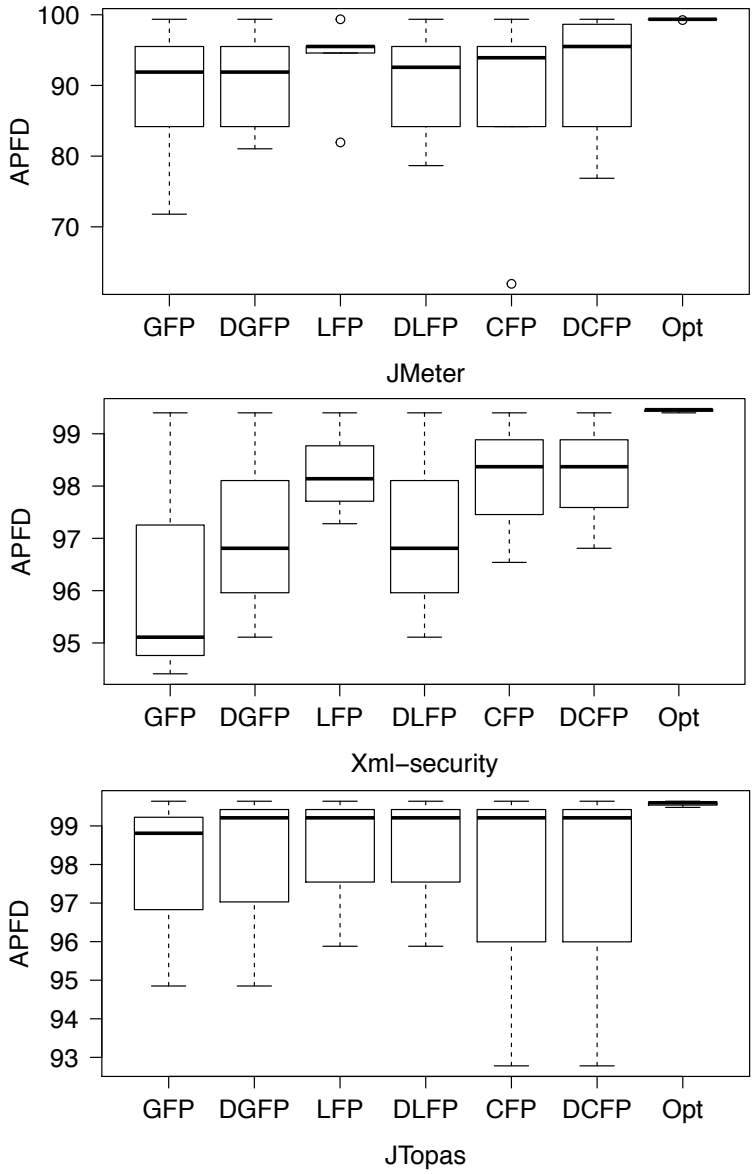


Figure 6.7: APFD values for our techniques applied to standard Java applications.

vertical lines (whiskers) at both ends of the box show the other values ignoring possible outliers. The horizontal axis of the figures represents the proposed techniques as well as the optimal technique that orders the tests according to their ability to detect faults as fast as possible. Of course, this is only possible because we have knowledge which tests fail.

Our techniques achieve very high APFD values which are sometimes close or even equals to the optimum. Furthermore, we note that the value distribution of our technique is often very small. When comparing the median values of all

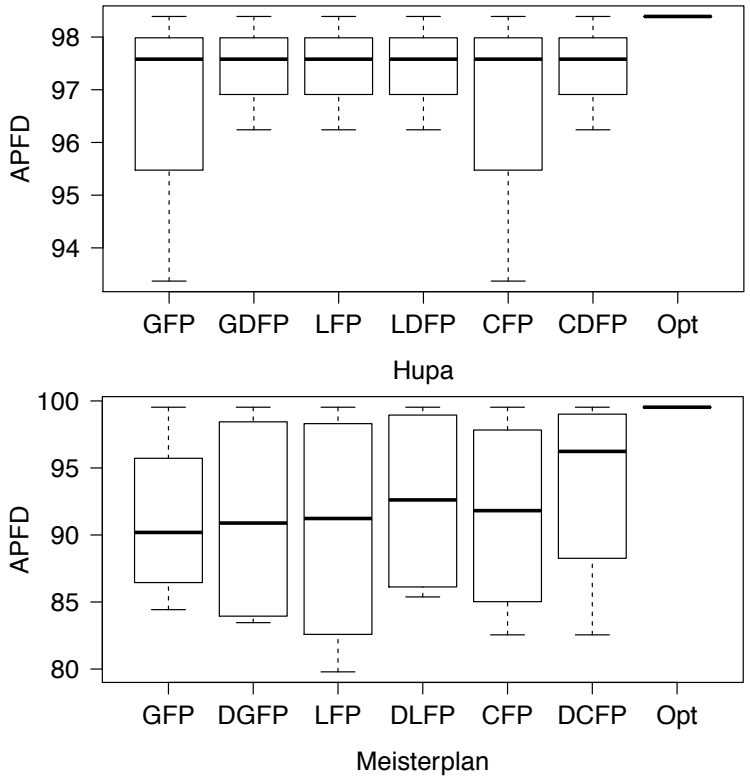


Figure 6.8: APFD values for our techniques applied to transcompiled GWT applications.

six prioritization variants, the difference always lies in the interval $I = [0; 6, 1]$. So none of our techniques falls extremely behind the others. Having a closer look at the results, LFP and (D)CFP outperforms GFP in most of the cases. Considering XML-Security, (D)CFP shows slightly better results than all other techniques. However, when considering JMeter, LFP is better. Besides, LFP has – with one exception – always a smaller deviation than CFP (see whiskers).The same is also true for DLFP that falls behind DCFP in one case only.

In order to judge the performance of our technique better, we compare our results with those obtained by other state-of-the-art techniques published in former times. Our comparison includes structural coverage techniques investigated by Do et al. [56]. Table 6.2a shows the results of their study [56, page 50]. It includes the following techniques: original ordering (T1), random ordering (T2), and orderings according to some variants of block/method coverage with/without considering change information (T3+T4/T5-T8). In detail, T3 prioritizes tests according to the total number of blocks. T4 additionally incorporates feedback. T5 prioritizes tests according to the total number of methods. Similar to T4, T6 adds the feedback-mechanism to T5. T7 runs tests first that show the highest coverage of methods that have been modified.

Objects	M	Do et al. Test-method level							
	SD	T1	T2	T3	T4	T5	T6	T7	T8
JMeter	M	48	60	34	74	34	77	42	55
	SD	37	19	38	24	38	18	35	35
XML-S.	M	48	71	96	96	97	87	96	96
	SD	34	17	3	3	3	12	3	3
JTopas	M	35	61	68	97	68	97	77	75
	SD	21	12	50	2	51	2	19	19

(a) Mean value (M) and standard deviation (SD) of APFD values in Do et al.'s former assessment [56, page 50].

	Mirarab et al.			
	S1	S2	S3	S4
JMeter	[65; 75]	[62; 72]	[70; 75]	[62; 72]
XML-Security	[90; 95]	[90; 95]	[90; 95]	[90; 95]

(b) Median APFD values of a technique replicated from Mirarab and Tahvildari [192, page 284].

	Fang et al.			
	T1	T2	T3	T4
JTopas	[60; 65]	[55: 60]	[50; 55]	[50; 55]
XML-Security	[80; 85]	[80: 85]	[65; 70]	[65; 70]
	T5	T6	T7	T8
JTopas	[55; 60]	[45; 50]	[30; 35]	[55; 60]
XML-Security	[80; 85]	[80; 85]	[80; 85]	[75; 80]

(c) Median APFD values of a technique replicated from Fang et al. [80, page 349].

Table 6.2: APFD values of alternative techniques.

Do et al. obtain change information by using the *diff function* provided by Unix. Finally, T8 extends T7 by a feedback mechanism.

Table 6.2b and Table 6.2c summarizes the median values we have gathered from boxplots presented by Mirarab and Tahvildari [192, page 284] and Fang et al. [80, page 349]. As it is difficult to determine the exact values from their boxplots, we use intervals $I = [a; b]$. Fang et al. have considered similarity to prioritize test cases. They present four own approaches (T5-T8) and compare them to other similarity-based approaches. T5 and T6 are based on an algorithm called "Farthest-first Ordered Sequence" [80, page 337]. T7 and T8 use an algorithm called "Greed-aided-clustering Ordered Sequence" [80, page 337]. Mirarab and Tahvildari have used bayesian networks to prioritize test cases. They have evaluated their approach with four different settings S1-S4. For each

of these settings, they obtain several APFD values. We unify the APFD values of every setting in the corresponding interval in Table 6.2b.

Compared with the approaches in Table 6.2, our techniques achieve higher APFD-values, which especially outperform the results of Do et al. [56] (compare e.g. JMeter in Table 6.2a with the corresponding values in Figure 6.7). Furthermore, we observe that our standard deviation is usually smaller. Regarding XML-Security, the overall results of Fang et al. [80] and Do et al. are more similar to ours, but especially Fang et al. show lower values as we do.

In summary, the evaluation of our test-prioritization technique shows that considering the execution frequency as main factor is a good criterion to calculate a test prioritization order which is close to the optimum.

RQ2: Dynamically Adapted Execution Order vs. Static Execution Order

As a first result, we can see that the dynamically adapted execution orders created with DGFP and DCFP are never worse than its statically calculated counterpart. Solely DLFP repeatedly falls behind the static variant. In some cases (e.g. XML-Security's DGFP, JMeter's DCFP, and Meisterplan's DCFP), the dynamic variant performs better due to the knowledge about failed tests. Of course, this benefit involves an increasing runtime as the prioritization order has to be recalculated after each failed test. Basically, this is no problem as long as the following conditions are met: a) the runtime of the prioritization technique is low, and b) the dynamically adapted order prioritizes fault revealing tests.

	Software whose tests will be prioritized				
	JMeter	XML-Security	JTopas	Hupa	Meisterplan
GFP	1,77	1,81	1,94	2,55	38
DGFP	1,93	1,76	2,05	2,98	91
LFP	1,46	1,94	1,97	2,71	34
DLFP	2,31	1,77	1,85	3,43	75
CFP	1,74	1,82	1,67	2,65	34
DCFP	1,89	1,97	2,24	3,24	91

Table 6.3: Runtime of prioritization techniques in seconds.

Table 6.3 shows median values for the runtime of all technqiues. The runtime for static variants of the desktop applications has been shown to be always less than two seconds. Regarding the larger applications Hupa and Meisterplan, the static variants still required less than one minute to calculate the test execution order.

Concerning the dynamic variants, the runtime increases. The main reason for requiring more runtime is the criterion PC_3 (highest structural coverage) as all the CIDs affected by a test have to be summed up. In large applications

with several millions of CIDs, this becomes easily a time factor especially when the prioritization order has to be updated often due to many tests failures. This is in particular true for MEISTERPLAN.

In total, dynamic techniques only provide a rather small benefit compared to the static ones. This is somewhat different to the findings of some researchers who observed significant improvements when using a feedback mechanism (e.g. [56]), but confirms the findings of other authors (e.g. [192]) who also observed small improvements, but an even worse runtime overhead than we have noticed. Thus, we conclude that the results are twofold as a dynamic approach improves the APFD value, but impairs the runtime. If we expect many test failures, the static technique is preferable as we do not have to re-calculate the prioritization order.

RQ3: Prioritization as Supplement to RTS Techniques

RTS techniques always have an overhead for analyzing the code and selecting the tests affected by code changes. When comparing RTS techniques with a retest-all approach, they have to reduce the number of tests in order to be cost effective. Sometimes however, a change affects all tests so the RTS technique is unable to determine a subset of the original test suite that is still safe according to the definition of Rothermel and Harrold [232]. Besides, the test execution order is unclear. To avoid running all tests requires knowledge about the probability that a test will reveal a fault. The extra overhead should be minimal. Our prioritization technique is very well suited as it is able to define such an order very quickly. All required data (i.e. code modifications, execution frequency, test traces) are already available from the test selection and therefore, the effort is completely negligible. Regarding the desktop applications, the techniques have always finished in less than two seconds. Even prioritizing tests for the industrial application has been very fast. So basically, our prioritization technique could be applied to the result of the test selection even if only a few tests have been selected for re-execution. According to Do et al. [57], this is considerably better than executing these tests unordered or randomly.

When applying dynamic techniques, we have to take into account how often the order will be recalculated. This implies that we have to estimate, how error-prone the code is. If the source base is almost stable, a dynamic technique would be a good choice. Otherwise, a static technique should be preferred. In any case, our techniques are able to define a test prioritization order that usually accomplishes in running fault revealing tests soon. The extra time-effort is minimal. This is very important as developers can decide to run only a subset of the original test suite. The risk to miss fault due to a test that has not been executed is rather small.

Due to the low extra time-effort, the overall time for determining changes in P' (see in particular Section 4.3.2, Section 5.4.2 and Section 5.4.4), and for prioritizing affected test cases remain low. For example, when checking out an old version P of MEISTERPLAN and determining the changes that have been in-

troduced in the new version P', our Eclipse plug-in requires for the analysis 315 seconds (median value, see RQ6, Section 5.6.4). When combining this analysis with the LFP or CFP technique, the analysis requires in total 349 seconds. If we additionally take the test selection into account, the overall time is 572 seconds. But actually, we do not have to wait for the test selection result. Instead, we can do the prioritization contemporaneously with the test selection. As the prioritization is faster than the test selection, the total time is 538 seconds at the maximum. But by the way, we believe that additional code optimizations could reduce this runtime further. That is, in any case, the combination of RTS and TCP techniques can be used during the usual development several times a day in order to determine a subset of test cases that needs to be re-executed. If the set of tests that should be re-executed is too large and if we just want to get a quick feedback whether the most important tests reveal any faults, we can start a *smoke test* (see Appendix A.2, Paragraph "Smoke Test") that just executes some of the tests with the highest priority. Depending on how much time the smoke test may take, we can dynamically determine the number of tests that will be executed. This way, we overcome the common nightly build and test cycle and get early feedback. This highly resembles continuous integration [84] as developers are able to commit their work several times a day and because they are able to test the changes even with UI/web tests. We explicitly say that it resembles continuous integration due to the extra analyses that are required for example to instrument code whenever test cases have been adapted or to do the test selection and prioritization.

6.6 Discussion

Test Suite Granularity: Do et al. [56] have considered in their paper the effects of different test suite granularities. They have investigated tests at class and at method level. Tests at class level may consist of an arbitrary-sized set of test methods. Executing a test class always implies executing all of the test methods within that class. As opposed to that, tests at method level execute only a single test method.

Actually, we could do easily the same. But in our opinion, considering tests at class level is too imprecise. Imagine a test class C consisting of many test methods. Let us assume that the test selection (see step 1) selects many test classes for re-execution, including C, but only a single test method in C covers a code change. As a consequence, all test methods in C would be executed, even though it is obvious from the test selection step that only one test method has the chance to reveal a fault. This is contradictory to our target to run only a subset of the original test suite in order to be cost effective. Furthermore, we have analyzed the applications on statement level and we have considered even expressions (e.g. conditional expression) to keep the set of selected tests as small as possible. For this reason, we consider in our approach every test method to be a test.

Dynamic Prioritization Variant: As already discussed in Section 6.4.5, our main motivation for the dynamic feedback mechanism has been to avoid running tests in vain. This could easily happen when a test has already failed and another one covering similar code is about to start. It has turned out that a dynamic approach in fact often improves the APFD value, but only to a small extend. This is due to the relatively high APFD values we already obtain in our static variants. In case of parameterized tests, it might even happen that the dynamic variant provides no benefit at all. This is for example true for JTopas. The test suite contains several parameterized tests. All of them already have a high priority in the static variant.

Moreover, it could happen that the dynamic variant ends up in a worse APFD value than its static counterpart. This might occur in the following situation. Let us imagine that there are three test cases r, s, and t. t runs with two different parameter settings p_1 and p_2. So in the end, there are four tests r, s, t_{p_1} and t_{p_2}. Let us assume that a dynamic prioritization strategy results in the following ordering: $\{t_{p_1}\}, \{t_{p_2}\}, \{r\}, \{s\}$. Now, test t_{p_1} fails. Consequently, the dynamic prioritization reduces the priority of t_{p_2}. The new ordering is: $\{r\}, \{s\}, \{t_{p_2}\}$. However, the different parameter settings p_2 could have revealed another bug. If this is the case and if the tests r and s cannot reveal any faults, the APFD value becomes worse due to the re-ordering. So as already stated before, the developer has to decide as the case arises whether a dynamic prioritization strategy could be beneficial.

Running Tests in Parallel: When searching for opportunities to reduce the runtime overhead, companies try to run tests in parallel. Our approach of combining a regression test selection technique with a test prioritization approach meets this demand. For our static variants, the only requirement is that tests run independently. Regarding our dynamic variants, further work has to be done. These approaches gain from knowledge of failed tests. Even if all processes that run in parallel have this knowledge, it has to be ensured that similar tests with differing parameters do not run in parallel on separate machines. Otherwise, this could impact the runtime benefit gained by parallel execution. To solve this problem, test groups are adequate. This concept is already known from unit testing tools like TestNG [274]. But even without using a test framework that supports groups, it is straightforward to provide meta data on which tests belong together. These tests should not run in parallel.

6.7 Conclusion and Future Work

Test case prioritization is a possibility to improve the execution order of a test suite when other techniques are not able to find an unambiguous one. To the best of our knowledge, we are the first who use the execution frequency of code modifications as major measure to prioritize tests. We have presented three different static variants of our technique plus three dynamic counterparts with

a feedback mechanism. To assess the performance of our technique, we have evaluated the different variants on three Java desktop and two web applications of different size. One of the web application has been an industrial application of large scale. The other applications have been of mid- or low scale size. In order to compare our results with findings in former papers, we have used the standard APFD metric.

The results of our evaluation show that our prioritization techniques perform very good. The APFD-values are sometimes close or even equals to the optimum. Most important, both our prioritization techniques LFP/DLFP and CFP/CDFP have outperformed existing state-of-the-art techniques. Thus, we are able to detect faults earlier than others.

When comparing our static techniques with the dynamic counterparts that rely on a feedback mechanism, we have observed that the dynamic techniques provide rather small improvements at a higher runtime. As the APFD values of our static variants are already high and the deviation is rather small, it has to be decided individually whether one of our dynamic variants is able to outperform the static counterpart. If we do not expect many test failures, the dynamic technique is a clear option. In this case, we do not have to re-calculate the prioritization order often and consequently, we do not loose much time. But especially when looking at the industrial application, the additional runtime overhead of the dynamic techniques has been too large to be cost efficient.

Finally, combining our prioritization technique with our RTS technique solves the test effort reduction problem (see Section 1.2.1). Our regression test selection reveals which tests should run at all. Even if the RTS technique already achieves a high reduction, the prioritization gives a clear advice about the execution order of the selected tests. But most notably, if the RTS technique has selected many or even all tests for re-execution, our prioritization runs the most important tests first in order to find faults as soon as possible. This way, if there are time constraints that prevent us from executing all the tests, we know that the tests with the biggest potential to reveal faults have been executed. Of course, this kind of test reduction is not safe any more.

With our approach, it is possible to establish a workflow similar to continuous integration that even runs UI/web tests. Developers commit their work several times per day. In smoke tests, our approach checks with the most important tests whether the application still works as expected. This is highly beneficial as we get a fast feedback about faults that arise during the user interaction. We are not restricted to small and isolated unit tests any more. Consequently, we overcome the common nightly build and test cycle. This supports developers in faster creating software of high quality.

Concerning the three problems listed in Section 1.2, the coverage identification problem is the only one that is left. We attend to this topic in the next chapter.

Chapter 7

Code Coverage for Any Kind of Test in Transcompiled Cross-Platform Applications

7.1 Introduction

Up to now, we have concentrated on ways to reduce the effort of regression testing by means of regression test selection and test case prioritization. While doing so, we have always worked with already existing test suites. An additional problem associated with selective retest techniques [233] is to decide whether a test suite is complete or which parts of the code require additional tests. In this chapter, we focus this problem which is known as the coverage identification problem [233]. We provide a general solution that is usable for all kinds of tests including UI/web tests, and that supports standard desktop applications as well as transcompiled cross-platform applications.

We start in Section 7.2 with an overview of state-of-the-art approaches. Then, we explain in Section 7.3 problems of these approaches in more detail and which challenges emerge when applying them on transcompiled cross-platform applications. Afterwards, we show in Section 7.4 how our approach solves the coverage identification problem (see Section 1.2.3) in our special context. Thereby, we are able to decide whether a test suite is complete or which parts of the code require additional tests. Section 7.5 presents an extended version of our Eclipse plug-in TC3 that calculates the following metrics: statement coverage, branch coverage, loop coverage, method coverage, and class coverage. In Section 7.6 and in Section 7.7, we investigate, evaluate, and discuss the results of our approach when applied in various kinds of real software projects. Our main target is to reveal deficiencies in test suites. In this context, we also compare the results of our coverage tool with the results of other code coverage testing tools in terms of correctness and efficiency. Finally, we conclude in Section 7.8.

This chapter is based on our publication "Code Coverage For Any Kind Of Test In Any Kind Of Transcompiled Cross-Platform Applications" [134].

7.2 Overview and Related Work

One possibility to solve the coverage identification problem consists in measuring to which extent the code of an application is covered by test cases. This is an old way to dynamically assess the completeness of a test suite. For example, Miller and Maloney [190] have already described in 1963 a way to check whether the whole program works as expected with differing input data. In general, having many specific tests that collectively achieve a high coverage increases the likelihood to reveal most of the errors in the code. Of course, as already shown for example by Sneed [259], there might still be lots of faults in the software left. But it is generally accepted that exploiting code coverage data is at least a good starting point to recognize which functionality needs additional testing.

Today, for almost all programming languages, diverse code coverage tools are available differing in the way how the statistics are obtained. Common to many coverage tools is the approach to insert *instrumentation code* into the application. Differences affect the kind of files selected for adding instrumentation code and the point in time when instrumentation code is added. Some techniques insert instrumentation code into binaries (e.g. [261]) or into Java bytecode in `.class` files (e.g. [43, 195, 210]), others add instrumentation code directly to plain source code (e.g. [19, 44, 254]). Several techniques require this instrumentation step before any test or program is executed (e.g. [211]). This is called off-line or static instrumentation [140, 211]. In contrast, other techniques perform this operation dynamically (on-the-fly) at runtime [140, 211].

In the environment of Java, existing code coverage tools often focus on a specific kind of test and/ or application. For example, tools like JaCoCo/ EclEmma [139, 140, 195], JCov [210, 211], or Cobertura [42, 43] instrument Java bytecode. In today's web applications however, bytecode is usually never executed on the client side. So these tools are unable to report on the code coverage of web tests. They focus on unit testing. Sometimes, there are adaptations available for unit testing frameworks like JUnit [160]. For example, EclEmma is able to run unit tests for the Remote Application Platform via JUnit RAP [196] in order to calculate code coverage. But of course this is no end-to-end testing.

Moreover, to the best of our knowledge, there is almost no tool available that supports transcompiled cross-platform applications. For GWT, solely the framework itself offers a partial solution for client-side code coverage via a barely known possibility to instrument the JavaScript code that has just been created by the GWT compiler [96]. After executing the web application, the browser's local storage provides key/value pairs that contain for each Java source file the line numbers (keys) and a flag indicating whether the line has been traversed (value) [96]. However, this information meets line coverage only, the most simple coverage metric. GWT is unable to compute more advanced coverage statistics like branch coverage. For server-side code coverage and for

measuring the coverage of GWTTestCases (see Section 3.1), the GWT tutorial still recommends EclEmma to calculate the code coverage [111].

Approaches determining the code coverage by instrumenting source code (e.g. [19, 44, 254]) are close to our approach. Hanussek et al. [124] describe the instrumentation process for CodeCover [44]. To provide for example statement coverage in Java, they add instrumentation code in front of statements as we do in our compiler-independent variant. To this end, they generate a parser by using the Java Tree Builder [278] in order to build up an AST which is used to insert instrumentation code and to generate instrumented source files. Besides, they require a modified abstract syntax tree which contains only information about the source code that is of interest for calculating the code coverage. This additional tree is the link between their coverage log and the corresponding source code. One of the main differences compared to our approach is the kind of instrumentation code. They add simple counters (see Section 4.4.3) whereas we assign unique identifiers to code elements of the software under test. These identifiers remain unchanged during a transcompilation and/ or a potential code obfuscation. Another main difference is that we do not need an additional syntax tree as this would require additional memory.

The proprietary tool Atlassian Clover [19] is also similar as it calculates the code coverage by instrumenting source code, too [22]. Furthermore, it uses a database to calculate the coverage as we do [20]. However, if we would like to use it for determining the code coverage of a GWT-based web application, it is not able to instrument client-side code properly as described in the official documentation [18, pages 224-227], [21]. According to this, the problem is that client-side code with Clover instrumentation code will fail to run. Thus, only code coverage of server-side code can be determined. The documentation recommends the user to use mock-frameworks like Mockito or EasyMock, or to use GWTTestCases instead of web tests.

Finally, the proprietary Java Test Coverage Tool offered by Semantic Designs [254] does not require class files for code instrumentation either. But it offers less detailed coverage measures as it only supports coverage values for methods or even more coarse-grained elements like classes [254].

When comparing tools based on source code instrumentation with tools based on bytecode instrumentation, we want to remind the interesting findings of Li et al. [179] (see Section 4.2). They have noticed that there exist only three tools under active development that support branch coverage for methods. The first one is EclEmma (bytecode instrumentation), the other two are Clover and CodeCover (source code instrumentation). But most interestingly, they have found that – at least for the tool EclEmma used to investigate branch coverage – "Bytecode instrumentation is not a valid technique to measure branch coverage" [179, page 387]. Apart from that, they have discovered that both CodeCover and Clover have weaknesses in their implementation of branch coverage. Li et al. outline that branch coverage should theoretically subsume statement coverage. However, according to their results, neither EclEmma, nor CodeCover, nor Clover implement branch coverage

correctly. As a result, branch coverage does not subsume statement coverage in these tools.

With regard to efficiency, there is research on how the amount of instrumentation code added to a source file can be reduced. Tikir and Hollingsworth [276] propose a dynamic instrumentation approach to reduce the number of instrumentation points by using dominator trees. The idea of reducing the amount of instrumentation code at runtime has also been pursued by Chilakamarri and Elbaum [40]. They use an approach called "disposable coverage instrumentation" [40, page 267] to remove instrumentation code after it has been executed. Häubl et al. [127] present a runtime system that takes advantage of a virtual machine designed to collect profiling data for an application. Their system does not need to instrument code to provide coverage information. Approaches to reduce the instrumentation overhead could basically be orthogonal to our code coverage method. However, all these approaches are dynamic and fail to provide code coverage data for transcompiled applications.

Kim [165] allege many reasons why code coverage calculation is problematic in large software applications. Among others, they reason this with a bad performance due to necessary code instrumentation and the uncertainty that the extra effort will pay off. For these reasons, they present an approach that incorporates error-proneness of modules, their size, and whether code has just been added. They rely on observations made by other authors who state that many defects are located in a small number of modules (e.g. [31]; this is also called Pareto-like distribution, see Appendix A.1, Paragraph "Pareto-like Distribution"). Accordingly, Kim perform in general a coarse coverage analysis and argue that a precise coverage analysis would waste time for instrumenting code. They perform a precise code coverage only on added and error-prone code. The distinction between precise and coarse-grained analysis is reminiscent of our method when analyzing two program versions for code changes (see Section 5.4.2). Basically, it would be no problem to implement the approach of Kim in our code coverage technique. In our evaluation in Section 5.6 however, we could not confirm a Pareto-like distribution. Right now, we believe that it is desirable to obtain precise code coverage measures for the whole software as it displays exactly which parts of a function is uncovered by tests. Sometimes, it is even a firm requirement to prove a certain (100%) statement coverage. In addition, we claim – other than Kim – that calculating code coverage precisely is not extremely time and resource expensive in a coherent testing environment like the one proposed by us.

To overcome the restrictions of focusing on specific tests and not supporting transcompiled cross-platform applications, we explain in this chapter how we adapt our method based on logging code identifiers (CIDs) of traversed pieces of code. This way, we are able to manage and display code coverage for any kind of test (unit, integration, or UI/web test) regardless of whether the software under test (desktop, mobile, or web applications) is transcompiled or directly developed in the target language.

7.3 Motivation and Challenges

Similarly to the problem of localizing faults in the source code, the main problem when investigating the code coverage of a software under test is that tests run the application in the target programming language and that the application might be platform independent. Figure 7.1 shows this principle in the ellipse on the right. Let us imagine the following situation: A test engineer wants to know which parts of the code of a web or mobile application are not covered by existing test cases yet. This is decisive for creating additional test cases that address the uncovered parts of code. For testing the user interface and for checking the overall behavior of a web application/mobile application that has been transcompiled, the test engineer runs acceptance tests (i.e. web/user interface tests) that simulate actions in a browser/in the smartphone. As a result, data c_{tl} describe the coverage in the target language. Basically, we could use c_{tl} to calculate a percentage describing the code coverage. Besides, we could even highlight the source code of the target language with red and green colors (which is common in code coverage tools) to indicate which parts of the code are (un)covered.

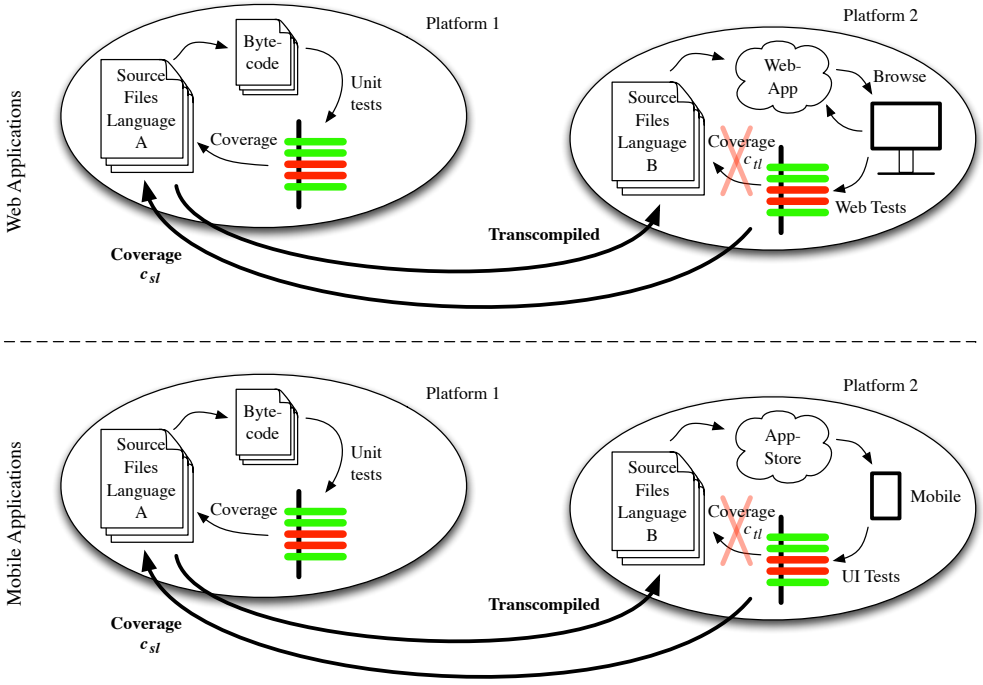


Figure 7.1: Problem of transferring coverage data back to source programming language in a transcompiled cross-platform web application.

However, we want to remind that when analyzing where additional test cases are required, application developers would always have to have domain knowledge about possibly many different target languages depending on the transcompiler. In case of Haxe, there are 12 different target languages

in total [131]. While this drawback could be handled, things get very difficult when source code of the target language has been obfuscated additionally. Now, recognizing which parts of the code require additional testing is hard. Deducing coverage in the source language (c_{sl}) from c_{tl} is very time consuming and still very difficult for developers. For these reasons, analyzing the source code of the target language is not helpful. Instead, we are interested in getting data c_{sl} about (un)covered code in the source language used by developers. Consequently, a code coverage tool should be able to transfer c_{tl} back to c_{sl}.

Unfortunately, existing instrumentation approaches just provide coverage for applications executed on the same platform (see the ellipse on the left in Figure 7.1), but do not support a transfer of coverage data from the target language back to the source language, or are unable to calculate the code coverage for all parts of the software. As an example, even the proprietary tool Atlassian Clover [19] cannot determine the code coverage of client-side code in GWT applications, although it instruments source code rather than bytecode [21, 22]. In order to test client-side code, Clover needs additional workarounds trying for example to simulate the client-side of a GWT web application inside the JVM by means of mocks or specialized tests based on JUnit (GWTTestCase) [21].

7.4 Approach

Our method to calculate the code coverage of a software under test aims at collecting coverage information that can be assigned unambiguously to the code of the source language to identify (un)covered source code. To achieve this, we can reuse most of our general approach explained in Section 4.5 and Section 4.6, respectively. We explain the differences in the next section in detail.

7.4.1 Code Coverage of Transcompiled Applications

For selecting test cases, it was sufficient to know which code entities have been traversed by a test. Depending on the coverage measures that should be computed, it is additionally necessary to know how much classes, methods, loops, branches, and statements the tests should have been traversed in theory to achieve 100% code coverage. To this end, we adapt the last step in our basic approach (see Section 4.5). The first two steps (see Section 4.5, "Step 1 – Code Instrumentation" and "Step 2 – Test Execution and CID Logging") remain the same. The instrumentation level has to be set to expression star (see Section 5.4.1). This is necessary to include data on conditional expressions in the branch coverage measure.

Calculating Code Coverage and Creating Reports (Step 3)

We want to calculate percentages that describe to which extent code fulfills certain code coverage criteria (i.e. statement coverage, branch coverage, loop-,

method-, and class coverage). As a result of step 2, we know which CIDs have been executed by a specific test and how often a CID has been executed by a test. Now, we have to determine which CIDs fulfill a specific coverage criterion. In addition, we have to calculate which code entities should be covered by the tests to achieve total coverage for every coverage criterion. That is, we have to count the total amount of functions, statements, branches, conditions, and optionally other syntactical elements to provide the usual code coverage measures.

To achieve this, we traverse the AST of the software written by the developer and calculate for each syntax element of interest (e.g. statement, function etc.) its CID. As a result, we obtain a table that maps each CID to its kind of syntactical element. The data will be stored in a database. For a better understanding, we illustrate this in an example based on the code presented in Section 4.5. We re-illustrate the examples in Figure 4.19 and in Figure 4.20 for convenience. There, we can see several pseudo-CIDs:

```
1  class C {
2    private int i;
3    private static int j;
4
5    C() {
6      instrument(<unique class-id>);
7      instrument(<unique id representing j>);
8      instrument(<unique id representing i>);
9    }
10
11   static {
12     instrument(<unique id representing j>);
13   }
14 }
```

Figure 4.19, taken from Section 4.5: Field instrumentation, class variables instrumentation, and class instrumentation.

```
1 public int m() {
2   instrument(<unique function-id representing m()>);
3   instrument(<unique statement-id>);
4   return 1;
5 }
```

Figure 4.20, taken from Section 4.5: Standard and function instrumentation.

In Table 7.1, the first column shows all the pseudo-CIDs in Figure 4.19 and Figure 4.20. The remaining columns represent syntactical elements like type declarations (TD), method declarations (MD), field declarations (FD), or statements (S). Hence, Table 7.1 assigns each pseudo-CID in Figure 4.19 and Figure 4.20 one or several syntactical element(s). We refer to these data as *syntax analysis*. Please note that the mapping is neither injective nor surjective

CID	Syntactical Elements			
	TD	FD	MD	S
`<unique class-id>`	×			
`<unique id representing i>`		×		
`<unique id representing j>`		×		
`<unique function-id representing m()>`			×	
`<unique statement-id>`				×

Table 7.1: Mapping of CIDs to syntactical elements.

(see also Appendix A.1, Paragraph "Properties of Functions"). That is, a CID can correspond to more than one syntactical element. For example, a then-statement corresponds to both statements and branches. Conversely, some syntactical elements might not be used in a project at all. But many different CIDs point to a specific syntactical element.

Finally, in order to calculate the code coverage, we join the tables containing information on test traces (see Section 4.5) and syntax analysis. This way, we can extract which CIDs executed by tests fulfill which coverage criteria. Besides, we can extract from the syntax analysis the total amount of statements, branches, loops, methods, and classes. Using these values, we are able to determine the final code coverage measures. We are even able to display exactly, how often a code entity has been traversed by tests by querying its execution frequency. In addition, our data affords us to determine which/how much code entities a test case traverses. But most importantly, we can use the data in order to check whether a specific syntactical element represented by a CID is covered by a test. Based on this knowledge, we highlight the code in the source language with green and red colors to indicate the coverage status. Now, developers can see where additional tests are necessary.

7.4.2 Discussing the Instrumentation Approach in Terms of Code Coverage

Apart from the discussion in Section 4.9, there are additional aspects of our code instrumentation approach that we want to discuss with respect to code coverage.

Instrumenting Code Explicitly

Our approach is based on inserting CIDs for each code entity of interest. We do not use implicit CIDs in our approach. That is, when traversing for example the instrumentation code that belongs to the field `i` (see Figure 4.19), we do not conclude that class `C` is covered. Instead, we add an additional CID in the first line of the constructor representing the class declaration. When calling

the constructor, the CID will be passed to the logging server which in turn will persist the CID in the database.

Our explicit way of instrumenting source code has the advantage that we do not have to care about semantics. There is no need to deduce the pass of a code entity (in the example: class `C`) from the pass of another code entity (in the example: field `i`). Querying the database is therefore straight forward. For example, in order to calculate the class coverage, we traverse the AST of the source code and identify all nodes representing classes and their corresponding CIDs. Afterwards, we query the database in order to check which of the CIDs representing a class declaration have been executed by a test.

General Applicability

As already discussed in Section 4.9, our approach is completely generic as it can be applied to both transcompiled and non-transcompiled desktop, mobile, or web applications and to most programming languages in general. Basically, all programming languages can be inspected using an abstract syntax tree. The only restriction is the availability of frameworks like the Eclipse JDT to parse the syntax of the code quickly and the effort for creating and comparing CFGs. Apart from that, we handle each code entity in the same way which makes it easy to calculate the code coverage for other syntactical elements. In particular, we consider conditional expressions which are unsupported by some tools (e.g. the "Java Test Coverage Tool" [254]). Moreover, our approach makes it easy to determine the code coverage of additional syntactical elements in the source language. For example, we might include try-/catch-blocks (unconsidered by e.g. EclEmma [197]).

7.5 Tool Implementation

We have implemented a prototype of our code coverage approach as Eclipse plug-in called TransCompiledCodeCoverage (TC3), which is an extension to our previously described Eclipse plug-in GWTTestCaseSelection. It supports an easy and quick usage in the development process of web applications created with GWT and in Java desktop applications. Additionally, the tool could be extended easily to support mobile applications that are written in Java. The basic principle is identical to the one applied for GWT-based web applications. We explain this in Section 7.7 when discussing adaptations that have to be done to use the approach in other frameworks or in other languages.

TC3 offers the following coverage metrics: Statement coverage (S), branch coverage (B), loop- (L), method-/ function- (M), and class coverage (C). Despite our approach enables us to calculate even code metrics for expressions, we only support conditional expressions in order to provide data for branch coverage. However, because of our generic instrumentation approach, we are able to calculate the coverage of any kind of code entity.

TC3 consists of several modules which have mostly been described in Section 4.7 and particularly in Section 5.5. TC3 introduces some new interfaces. Thereby, it is possible to pass the name of the test that is currently running. This is helpful if code coverage should be calculated for single tests rather than for the whole application. If the software under test is a web application, client and logging server can establish a connection via the WebSocket protocol. This way, it is possible to pass and persist the CIDs that have been traversed by web tests. In case of desktop applications, this detour is of course not necessary. Instead, CIDs are written directly to the database. An additional module has the task to collect absolute data about the total number of statements, branches, loops, methods, and classes that could be traversed theoretically. Again, this is accomplished by traversing the AST of the software with the aid of the Eclipse JDT. Finally, another module calculates the code coverage and creates HTML-reports. Herein, source code is highlighted with green or red color, depending on whether it is covered by a UI/web test or not.

Our tool is mostly independent from any test tool. Examples are JUnit [160] or TestNG [273] for testing desktop applications/server-side code and Selenium [252] or TestComplete [258] for testing client-side code. For mobile applications that run on Android devices, Selendroid [250] is an option to test the user interface.

7.6 Evaluation

In order to assess our code coverage technique, we discuss this approach by asking the following three research questions:

RQ1 Is our approach able to reveal deficiencies in test suites, i.e. where additional test cases are required?

RQ2 Does our tool completely calculate the code coverage of UI/web tests and are the results correct?

RQ3 How long does our coverage technique take to get a result and how efficient is TC3 compared to others?

7.6.1 Software under Evaluation

In order to show the applicability of our approach on cross compiled applications, we again use Hupa [144] as web application created with Google Web Toolkit (GWT). As we additionally want to demonstrate the usability in other applications, we enhance our study by desktop applications. The Java desktop applications JTopas [30] and XML-Security [11] from the "Software-artifact Infrastructure Repository (SIR)" [55, 162] meet our demand for real world applications, so we reuse them. Furthermore, we investigate some small Swing-based Java applications provided by the contributors of Abbot [1].

Abbot [1] is a framework that enables developers to test Java user interfaces automatically. We have used the example code shipped with version 1.3. It

consists of 1.300 non-empty lines of code in 17 classes. For analyzing the code coverage of JTopas and XML-Security, we have always used the most recent version and the corresponding test suite that has been available in the SIR repository. (According to SIR [55, 162], JTopas encompasses 5.400 lines of code (LOC) in 50 classes, XML-Security contains 16.800 LOC in 143 classes.)

When investigating Hupa, we have also used the latest revision in the public repository (revision number 1684702).

7.6.2 Experimental Setup

Both JTopas and XML-Security are shipped with a JUnit test suite. Unfortunately, no UI tests are available. For this reason, these applications are suitable solely for unit testing and do not provide insights in transcompiled applications. Nevertheless, existing code coverage tools work fine with these applications. So we use them as benchmarks to check our code coverage results for correctness and completeness.

The test suite provided by Abbot runs tests that examine user interfaces created with the Swing library. The tests are also JUnit-based, so we can compare our results with other tools once again. In case of Hupa, our Hupa web test suite comprises 35 web tests created with Selenium. Because we do not have (unit) tests for the server side, we investigate Hupa only on client side which is our main concern. That is, results displayed for Hupa represent the coverage of client-side code.

For every test suite, we have calculated the code coverage using our tool TC3. To check our results, we have calculated the coverage with other tools, namely Atlassian Clover [19] and EclEmma [195]. This way, we are able to compare the results of our tool with established ones. The evaluation has again been performed on an Intel Core i5 2.4 GHz with 8 GB RAM.

7.6.3 Threats to Validity

The results of our evaluation and our conclusions might be threatened by different factors. We will discuss these issues and how we have tried to minimize their effects.

External Threats to Validity

It is difficult to find a transcompiled open source application shipped with web tests. With regard to industrial transcompiled applications, it is usually not allowed to publish any results. For this reason, we have used only one transcompiled web application in our study. Of course, this might influence the ability to generalize our results. To gain confidence in our approach and the validity of the results, we have additionally studied desktop applications. As our approach is generic, the procedure remains the same leaving aside the

fact that we do not need a logging server. CIDs traversed by tests can be inserted directly into the database. Besides, using desktop applications in our evaluation enables us to compare our results with other code coverage tools.

Another threat to validity might be the size of the applications. The performance and the ability to cope with large applications depends heavily on this measure. In this context again, we try to limit these factor by comparing the efficiency of our tool with other tools.

Internal Threats to Validity

In addition to the threat of possible faults contained in the modules that are responsible for inserting instrumentation code for logging CIDs, there might be also a fault in the module that creates the HTML-report. To provide confidence in our tool, we first have made initial checks on a set of small programs and we have manually inspected the results. In particular, we run our tool on real world applications and compare our code coverage results with the results of established code coverage tools.

7.6.4 Results

In the following subsections, we will discuss the results of our evaluation in terms of our research questions.

RQ1: Ability of our Approach to Reveal Deficiencies in a Test Suite

Table 7.2 and Table 7.3 show code coverage metrics for the software projects with a user interface. The columns show the coverage metrics we have investigated. The last column depicts the total number of Java classes in the project (more details follow later in this section). The rows contain the names of the coverage tools we have used. These are Clover and EclEmma, which serve as reference for our own tool TC3.

We have obtained the code coverage by running the corresponding test suites of our software projects. The values in the table cells represent the code coverage of the entire project. For example, column 2 in Table 7.2 presents the branch coverage when running the test suites. Not all tools provide the same code coverage metrics. Unsupported metrics are labeled with $\times$. For example, Clover does not provide results for class coverage.

As far as RQ1 is concerned, our tool TC3 has calculated coverage values for all the metrics listed in the tables and for both the transcompiled web application and the desktop applications. Besides, TC3 creates a HTML-report that shows for each class in detail which parts of the class's code (in the source language) are (un)covered by UI/web tests. An example of a HTML-report created with TC3 for Hupa can be found in Figure 7.2. Thus, the percentages indicate how thoroughly the web/desktop application is tested. By using the report, the developer can easily check which functionality is not covered by tests and therefore might fail when used by clients.

Tools	Code Coverage Metrics					
	S	B	L	M	C	#Classes
Clover	63%	55%	×	48%	×	
EclEmma	57%	56%	×	51%	72%	17
TC3	56%	54%	88%	48%	76%	

Table 7.2: Code coverage metrics for Abbot.

Tools	Code Coverage Metrics					
	S	B	L	M	C	#Classes
Clover	×	×	×	×	×	
EclEmma	×	×	×	×	×	484
TC3	24%	13%	18%	28%	25%	

Table 7.3: Code coverage metrics for Hupa. There are only web tests available. Only our tool supports web tests.

Tools	Code Coverage Metrics					
	S	B	L	M	C	#Classes
Clover	39%	29%	×	40%	×	
EclEmma	33%	28%	×	40%	50%	50
TC3	30%	29%	35%	38%	44%	

Table 7.4: Code coverage metrics for JTopas.

Tools	Code Coverage Metrics					
	S	B	L	M	C	#Classes
Clover	35%	38%	×	27%	×	
EclEmma	34%	37%	×	31%	52%	143
TC3	31%	37%	49%	27%	52%	

Table 7.5: Code coverage metrics for XML-Security.

```
110         @Override
111         public void onStop() {
112             super.onStop();
113         }
114
115         private boolean noContent() {
116             return "".equals(display.getMessage().getText()) && "".equals(display.getSubject().getText());
117         }
118
119         @Override
120         public void onCancel() {
121
122         }
```

Figure 7.2: Excerpt of a HTML-report created with TC3 to display (un)covered code in Hupa.

RQ2: Completeness and Correctness of the Code Coverage of (Transcompiled) Applications

In order to check the correctness and the completeness of our code coverage for UI/web tests, we investigate the coverage results presented in Table 7.2, Table 7.4, and Table 7.5 in detail. Table 7.3 is not usable as Clover and EclEmma are not able to measure the code coverage of a transcompiled application.

As we can see, Clover and EclEmma often differ slightly in their coverage results. This has also been detected by Alemerien and Magel [7]. We have found that this is due to a divergent computation model. When considering for example statement coverage, we have observed that the tools show divergent total numbers. Consequently, the results will differ. Obviously, the tools do not follow the same definition of statements. This seems also to be true for other syntactical elements like branches. Regarding methods, it is known [197] that EclEmma sometimes takes implicit default constructors or initializers into account when calculating the cover coverage. This is because the tool relies on bytecode [197]. We assume this as the reason for divergent values (compare for example the method coverage in Table 7.2 or Table 7.5).

When calculating statement coverage, we totally adhere to the Eclipse abstract syntax tree [62] and the Eclipse JDT [66]. That is, we rely completely on Eclipse's AST. It tells us about the kind of code entity and it helps us to insert our instrumentation code. In the same sense, we use from the Eclipse JDT `AbstractTypeDeclaration` as representatives of classes. EclEmma on the contrary also considers elements that correspond to `ClassInstanceCreation` in the JDT-API (i.e. in Java code: `new <Type>`, where <Type> might represent an interface declaration). As these elements correspond to expressions in the Java-AST, we do not take them into account. For this reason, our class coverage results differ in some cases from the results obtained by EclEmma. However, when ignoring `ClassInstanceCreation`-elements in the HTML-report provided by EclEmma, we found that in case of Abbot, the class coverage percentages are identical.

Taken as a whole, our results (percentages and HTML-reports) are very similar to the results of the other tools. Having regard to the given explanations, our approach seems to work correctly with respect to RQ2.

RQ3: Runtime and Efficiency of our Coverage Technique

Coverage approaches that instrument the source code are known to be slower than approaches based on instrumenting bytecode or binaries. The main reason is that techniques instrumenting source code always require a separate build [22]. As Clover also instruments source code rather than bytecode, we use this tool for our comparison.

We have observed that our approach is generally more time consuming. In the case of XML-Security and JTopas, our approach requires two times longer than Clover for instrumentation. The values often varied even though

we tried to stop all background activities that could distort our comparison. TC3 has always spent most of the time on instrumenting the code. Calculating coverage values with TC3 takes additional time. CLOVER presents results almost instantly. We will have to do further optimization to improve the efficiency. In particular, our tool currently takes almost no benefit of concurrency. This could be a first starting point for future work.

Notwithstanding, the time spent on instrumentation and on calculating the coverage results has never exceeded 3 minutes, even in the case of our biggest transcompiled cross-platform application HUPA. On average, HUPA has taken 122 seconds to complete. XML-SECURITY has required 1 minute on average. JTOPAS and ABBOT have been faster as there are less classes to instrument. So, although TC3 is slower than other tools, it does not take overly much time.

7.7 Discussion

Easy Recalculation of Coverage Data: Our approach requires a database to persist test traces. This is similar to other existing tools like CLOVER that also uses a database [20]. Based on the data in our database, it is possible to recalculate the code coverage directly as long as the source code did not change.

Need for Executing Tests Manually: Right now, calculating the code coverage can only be done semi-automatically. This is because our tool is completely independent from the tool that is responsible for running the test suite. In case of unit tests, this could be for example JUNIT or TESTNG; in case of web tests, this could be for example SELENIUM or TESTCOMPLETE. Of course, it is technically possible to provide additional settings that permit the user to define the required test tool and to start the tests automatically. Then, the entire process (code instrumentation, test execution and CID logging, and calculating the code coverage) is fully automated.

Mobile Applications and Effort to Support Corresponding Transcompilers: As already mentioned in Section 4.9, adding instrumentation code and creating test traces remain the same in the area of transcompiled mobile applications. It is just important that the corresponding transcompiler supports WebSockets. But this does not impose any restrictions as transcompilers either support WebSockets directly (e.g. Codename One [265]) or it is just a technical question to provide an implementation. Having a closer look at TC3, it currently injects a module M into the code that uses native JavaScript to implement WebSockets. This module has to be replaced to address the framework capabilities or the corresponding WebSocket extension provided by other developers. So in the end, we obtain for each UI test a test trace that describes exactly which code entities have been traversed. We want to emphasize that when utilizing the syntax analysis (see Section 7.4.1), we can calculate coverage measures for mobile applications in the same way as described for web- and desktop applications.

7.8 Conclusion and Future Work

In this chapter, we have presented a novel approach to calculate the code coverage. The approach is generic in the sense that it is applicable to cross-platform web-, desktop-, and mobile applications. In particular, it allows to compute the code coverage of transcompiled applications, whose source code has been compiled from a source programming language into code of another target programming language. Thus, we succeed to map the code coverage ascertained in a completely different target system back to the original source code. In the end, this solves the coverage identification problem (see Section 1.2.3) even for transcompiled cross-platform applications.

Our approach is implemented as an Eclipse plug-in and has been evaluated on both desktop and transcompiled web applications. We support the following widespread metrics: statement coverage, branch coverage, loop coverage, method coverage, and class coverage. The results of the evaluation show that our technique helps to decide where additional tests are required. By considering desktop applications, we have been able to compare our tool with other code coverage tools in terms of correctness and performance. On the one hand, it has turned out that our technique requires more runtime. The most time expensive task is instrumenting the code. As this is rather technical, we firmly believe that we can speedup the runtime via code optimization and concurrency. Furthermore, using a database server instead of a local MySQL distribution could additionally enhance calculations. This is left for future work. On the other hand, to the best of our knowledge, our tool is the only one that supports calculating the code coverage of transcompiled (web) applications. Besides, as we rely on source code instrumentation and as we also consider conditional expressions, we are able to calculate branch coverage correctly. Some tools ignore conditional expressions, which in turn results in wrong coverage results. Other tools rely on bytecode instrumentation, which is known to be potentially unsuitable for calculating branch coverage [179].

Part III

Overview: Solutions and Contributions for Challenging Problems

Chapter 8

Conclusions

8.1 Summary And Results

Our main intention has been to provide solutions for the test effort reduction problem, the fault localization problem, and the coverage identification problem in the context of transcompiled cross-platform applications that use UI/web tests to ensure software quality. As a first starting point, we have analyzed lots of existing approaches. However, they have all shown deficiencies and none has directly been applicable for our purposes. Due to transcompilation, the applications that we consider exist in two different languages: a source programming language and a target programming language whose code is usually highly optimized and obfuscated. The application is implemented in the source programming language, but the final, transcompiled application that runs on the device of the end user is written in the target programming language. UI/web tests execute the application in the same way as the end user does. So code changes made by a developer in the source programming language affect UI/web test cases that execute code in the target language. As we have outlined, the distinction between source and target programming language as well as the dependencies in between makes it difficult to determine a reduced set of UI/web test cases that should be re-executed due to code changes. Another difficulty is to find the reason for a test failure in the source programming language that has emerged when running the application in the target language. Finally, the gap between source and target programming language makes it difficult to determine the coverage of UI/web tests in the code of the source programming language.

In our research, we have found a way to close this gap and to solve all three problems. The first step has been to develop a tool that efficiently determines changes made in the source programming language of a modified program version. This is a precondition to identify all the UI/web tests that run modified transcompiled code in the target language. To this end, we have created a special control flow graph – the Extended Java Interclass Graph (EJIG) – as an improvement of an already existing Regression Test Selection (RTS) technique [126, 233]. We represent both the new and the old program version as EJIG

and compare these graphs node by node in order to find code changes. From this, we have demonstrated to be able to select a (preferably small) subset of UI/web tests that needs to be re-executed. This way, we avoid to re-execute the entire test suite, which reduces the test execution effort. Just as the original RTS technique, our improved RTS technique is safe and ensures that we detect exactly the same faults that a retest-all approach would detect.

Compared with other techniques, the main improvements of our RTS technique are:

- Our technique is applicable to UI/web tests that run on transcompiled cross-platform applications.
- The calculation of EJIGs is simpler and more straightforward compared with other approaches in the literature. We rely on the Eclipse Java Development Tools to create the EJIGs. Additional analyses of e.g. class hierarchies are unnecessary. In particular, this improves efficiency.
- We provide a more precise analysis of the source language that enables us to precisely identify and localize code changes, even in e.g. conditional expressions.

However, apart from the information about code changes, selecting tests requires some additional information about code executed by UI/web tests. In a second step, we have introduced an instrumentation approach that enables us to find out which UI/web tests traverse which parts of the target language. It is tailored to and fits exactly the demands of transcompiled applications that might run on completely different platforms. Here, we have solved several challenges:

- Which information must the instrumented source code provide?
- How should the structure of the instrumentation code look like to provide the required information?
- How can we access the data produced by the instrumentation code?
- Where should the instrumentation code be added in order to support many different transcompiler frameworks?

We have introduced a generic approach that is universally applicable in different transcompilers. It is based on code identifiers (CIDs) that represent code entities in the source programming languages via a special structure. On top of this, we have created two different approaches to instrument the code and to transfer the CIDs from the source to the target language without any modification during transcompilation and obfuscation. The first one – the compiler-dependent approach – exploits the internals of the transcompiler whereas the second one – the compiler-independent approach – works completely independently. In both approaches, we register each CID that is traversed when executing the application in the target language. When looking at the pros and cons of the two approaches, we have ascertained the following facts:

- The compiler-dependent approach is very promising. On the one hand, there is no risk that the relation between code entities and their corresponding CIDs gets broken during code optimization or during the transcompilation process. Besides, it is not necessary to insert instrumentation code in the source code. On the other hand, compiler adaptations seem to be inevitable which are tedious for external developers. So this approach unifies big advantages, but also a strong disadvantage. We found that for regular end-users, it is far too expensive to maintain a tool that implements this approach. A transcompiler might evolve rapidly as we have seen in case of Google Web Toolkit. Instead, this approach should be provided by the developers of the corresponding transcompiler themselves. Alternatively, it could also be interesting for third parties by offering such a tool as a (fee-based) service.

- The compiler-independent approach is more suitable in supporting transcompilers with many different target languages. Basically, it works with every transcompiler. The only prerequisite is that there is a well-defined set of rules that clearly determines where instrumentation code has to be injected in the source programming language. Especially due to the high maintenance effort that is necessary in the compiler-dependent approach, we have used the compiler-independent approach for further investigations. However, we want to emphasize that all of our investigations could have been performed with the compiler-dependent approach as well.

In general, a main prerequisite for applying an arbitrary technique is that it is efficient. In our special case, when focusing on the test effort reduction problem and the fault localization problem, the analysis needs to be fast, the memory consumption has to be low, the test selection should pick exactly these tests that are really affected by code changes, and the localization of faults should be easy and time-saving. Precise test selection and exact fault localization are closely related, but impact memory consumption and time effort. For this reason, we have looked in a third step at possibilities of our technique to fulfill all these conditions in our special context and to improve the state of the art known from other techniques.

We have investigated and compared several analysis precision levels. To this end, we have implemented several analysis precision levels that can be adjusted by the user.

- Naturally, the fine-grained analysis provides better results for test selection (i.e. it selects less unnecessary tests) than a more coarse-grained analysis levels that models the applications less detailed. Besides, the fine-grained analysis localizes code changes/possible faults more precisely which reduces the time for bug fixing.

- However, the fine-grained analysis consumes much memory than the less detailed analyses. Especially in large real world applications, this might end up in memory problems. Surprisingly, the coarse-grained analyses

often have execution times similar to fine-grained analyses. It has turned out that this is due to the need to analyze method invocations (i.e. plain expressions) in order to represent calls in the CFG.

- In order to combine the bigger test suite reduction from the fine-grained analysis levels and the lower memory consumption from the less detailed analysis levels, we have developed a heuristic that individually decides about which parts of the source code have to be analyzed precisely and which parts can be analyzed coarse-grained. It has turned out that with the heuristics, the test selection remains similar to the fine-grained analysis. In addition, the memory consumption can be reduced significantly.

In order to reduce the overall time-effort to reveal all the faults in a new program version, we have made additional investigations:

- To detect more possible faults in a single analysis, we have applied a lookahead strategy that is based on a formerly published lookahead strategy [12]. Other than previous techniques, we do not stop the analysis of e.g. a function as soon as a change has been detected. Instead, we try to resume the analysis at an appropriate position in order to analyze the rest of the function. This way, we can point out more potential faults in a single analysis. If there are consequential errors, other approaches cannot detect these faults immediately. Instead, they need another analysis with test selection, re-execution of the selected tests, and fault localization. To achieve our goal, we have used two different search strategies and a lookahead to find a point in the code where we can continue our analysis. In an evaluation, we have pointed out that our strategy accomplished our demands and expectations to reduce the overhead for subsequential analyses.

- We have shown that the test selection can (sometimes) be reduced significantly no matter what kind of instrumentation approach we use. With our heuristics and lookaheads, the technique often shows potential to be more efficient than a retest-all approach. However, this heavily depends on the kind of code change and on the structure of the UI/web test. As this kind of tests does not execute a small isolated unit in the source code, a code change might affect many different test cases. For this reason, there might be cases where the test reduction is not enough to outperform the retest-all approach. Consequently, additional measures are necessary to ensure that the analysis provides a benefit compared with a retest-all approach.

To solve problems with low test suite reductions that arise from the special nature of UI/web tests, we have developed in a fourth step several novel prioritization techniques that build on the results obtained from our test selection. With these techniques, we can find out which tests have the highest potential to reveal faults and therefore should run first. If we do not have enough time

to execute all the tests, we know that the most important ones have been executed. For this purpose, we have introduced, investigated, and compared three static approaches that incorporate several criteria. The basic assumption has been that the chance to detect a fault is greater when a code modification is executed in many different application states. So we have proposed to take the execution frequency of code modifications into account when prioritizing tests cases. Apart from this, we have also included other criteria like coverage of changes by a test, common code coverage, or the determination of an optimal test for a given code change. For each of the static approaches, we have additionally proposed a dynamic counterpart that adjusts the execution order at runtime based on the test execution result. The main insights of our evaluations have been:

- We have found out that our test case prioritization techniques are almost for free, that is, they require only little additional time for execution. This is because they benefit heavily from the results of our preceding test selection, which provides all necessary data about code changes and execution frequency. This information is already available from the analysis, the code instrumentation, and the logging.

- Our prioritization techniques are highly performant as they achieve very high APFD values. Sometimes, the APFD vales are close or even equals to the optimum.

- When comparing the static prioritization techniques with their dynamic counterparts, the dynamic ones only provide a rather small benefit compared to the static ones. If there is a high chance for many failures, the static techniques should be preferred.

- In total, our prioritization technique is very well suited as supplement to RTS techniques because it is able to quickly define a clear order in which tests should be executed. So in total, with our heuristics, the lookaheads, and our prioritization, our RTS technique is applicable to UI/web tests in transcompiled cross-platform applications in an efficient way.

Naturally, however, the best test selection is useless if essential parts of the code remain untested. In order to understand which functionality in the code of the source programming language needs additional UI/web tests, we have presented an extension of our instrumentation and tracing approach to provide detailed code coverage metrics. The main results of this final step are:

- Similar to the test prioritization, all necessary data are available from the test selection process. So, the code coverage analysis is cheap, although it takes more time than an existing state-of-the-art tool.

- Unlike some other code coverage techniques, we are able to calculate branch coverage correctly, as we also consider conditional expressions.

- To the best of our knowledge, there is almost no tool available that fully supports code coverage for UI/web tests in transcompiled cross-platform applications.

8.2 Future Work

While solving the test effort reduction problem, the fault localization problem, and the coverage identification problem in the context of transcompiled cross-platform applications, many questions and challenges arised. We have answered most of them, but some topics and tasks are still open and could be addressed in future work. Here is a list of the main topics and tasks:

First of all, there are some technical topics left that could be included in our RTS technique:

- Currently, our comparison algorithm does neither support the Java reflection mechanism, nor concurrency (see Section 4.3.2, Paragraph "Preconditions" for the reasons). For these purposes, our approach would have to be extended. Especially when considering multi-threading, we could imagine that some of the ideas proposed by Apiwattanapong et al. [12] could be integrated.

- Some transcompilers offer to embed code written in other programming languages within the source programming language. These secondary programming languages might require special methods when determining code changes in a new program version. A prominent example is Google Web Toolkit that offers to include JavaScript. Currently, we do not model these parts of the code in our approach. However, there are approaches (e.g. [154, 155]) that could be used to model these parts as well. An interesting question is how these approaches could be integrated in order to model the complete control flow in a cross-language way.

When trying to facilitate fault localization, we could imagine that our technique could be enhanced further:

- As mentioned in Section 4.2, spectrum-based fault localization techniques order statements according to their risk to be faulty. Currently, our analysis indicates in the best case the exact code location that causes a test failure. Often however, a set of code changes might be responsible. An order that reflects the probability of a code change to be the reason for a test failure could be helpful. Of course, it has to be investigated whether the extra analysis effort pays off in practice.

In the area of (re-) creating test traces for test selection and code coverage analysis, there is in particular one interesting question open:

- It concerns the effort to recreate traces for tests that have been selected for re-execution. Currently, we usually recreate the whole traces auto-

matically while running the selected tests. Although this is done automatically, it requires instrumentation code in the target language that naturally slows down the test execution. This problem is not unique to transcompiled applications. It is a general problem. To overcome resulting drawbacks, Chittimalli and Harrold [41] have proposed an approach for pure desktop applications to re-compute coverage information automatically. Although the approach does not fit our demands in transcompiled cross-platform applications directly, it might be a first starting point for future work to reduce the overhead for recreating test traces.

In order to reduce the effect of code changes on the number of tests selected for re-execution, we have been concerned with the optimal design of UI/web tests. One possible approach is to modularize tests in such a way that they are reusable. This way, we are able to create test traces for small test cases that attend to a small specific part in the user interface. In fact, we have used this approach several times (e.g. for the login in MEISTERPLAN). However, this introduces dependencies among test cases that have to be taken into account during prioritization. And it still cannot resolve another major problem that we have recognized. It affects the traversal of code just for initializing or navigation reasons. It might happen that a UI/web tests traverses code changes although it actually focuses on completely different functionality and thus has nothing to do with these code changes. Nonetheless, it could be selected for re-execution. Modularizing the tests does not suffice as the preceding tests have to be executed anyway. Although we could turn off logging in order to create a test trace for the test we are interested in, we would lose time due to the execution of the preceding test. So, as another optimization, our idea has been to create a kind of snapshot of the individual application state and to resume from this state for several follow-up tests in the user interface. This way, it would be possible to test individual parts of the user interface of an application without the need to return to a specific view. We would expect that the impact of some kinds of code changes could be minimized. This could reduce the effect of code changes on UI/web tests which in turn leads to smaller subsets of tests that have to be re-executed. Traversing code just for initializing or navigation reasons would be obsolete. Nevertheless, this idea is very challenging. Major questions in this context are:

- How is it possible to capture the state of a web/mobile/desktop application?

- How can an application state (taken as kind of snapshot) be recovered? This of course includes recovering the data from the internal memory as well as setting up the user interface as it has been left before.

- Does our expectation prove to be true that tests starting from a previously captured application state can further reduce the set of tests that have to be executed after code modifications?

- Which option requires less time: running the preceding tests and the main test or capturing the application state and resetting the application in this state?

Finally, we have explained that the rule set for code instrumentation might have to be adapted in other languages. Even if the overhead is small: it would be nice to get rid of the need to do adaptations.

- It would be interesting to explore whether a completely dynamic and generic instrumentation approach could be established that can be used for any kind of source programming language without the need of a single adaptation. Geimer et al. [92] have already pursued a way to create a dynamic, generic instrumentation technique for multiple programming languages. But as explained by the authors, there are also several open questions left. For us, another open question is in particular whether an enhancement of their approach could be applied in our special settings as well.

Appendix

A Terminology

A.1 General Terms

Pareto-like Distribution:

> "Software defect distribution following Pareto distribution (i.e small number of modules causing majority of defects)." Kim [165, page 148]

Properties of Functions:

> "**(*a*) Injective (One-to-One) Functions.** Let $f : X \to Y$. The function f is called one-to-one or injective if different elements in X have different images in Y i.e., if $f(a) = f(a') \Rightarrow a = a', \forall\ a, a' \in X$.
>
> Another way of defining injective function is that every element of domain X has a unique image in the co-domain Y and there is no element of Y which is image of more than one element of domain X." Gupta [123, page 79]
>
> "**(*b*) Surjective (Onto) Functions.** Let $f : X \to Y$. The function f is called surjective function if each element in Y, is the image of at least one element in X. In other words, in surjective functions, the range of f is equal or co-domain Y i.e., $\forall\ b \in Y, b = f(a)$ for some $a \in X$." Gupta [123, page 79]

A.2 Testing

Test Case:

> "**1.** A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. *IEEE Std 1012-2004 IEEE Standard for Software Verification and Validation.3.1.31.* **2.** Documentation specifying inputs, predicted results, and a set of execution conditions for a test item. *IEEE Std 1012-2004 IEEE Standard for Software Verification and Validation.3.1.31*" ISO/IEC/IEEE [149, page 368]

Acceptance Test:

> "The test of a system or functional unit usually performed by the purchaser on his premises after installation with the participation of the vendor to ensure that the contractual requirements are met. *ISO/IEC 2382-20:1990, Information technology – Vocabulary – Part 20: System development.20.05.07.*" ISO/IEC/IEEE [149, page 5]

Integration Test:

> "The progressive linking and testing of programs or modules in order to ensure their proper functioning in the complete system. *ISO/IEC 2382-20:1990, Information technology – Vocabulary – Part 20: System development.20.05.06. Syn:* integration testing" ISO/IEC/IEEE [149, page 181]

Integration Testing:

> "Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them. *IEEE Std 1012-2004 IEEE Standard for Software Verification and Validation.3.1.14; IEEE Std 829-2008 IEEE Standard for Software and System Test Documentation.3.1.14*" ISO/IEC/IEEE [149, page 181]

Regression Testing:

> "**1.** Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements **2.** Testing required to determine that a change to a system component has not adversely affected functionality, reliability or performance and has not introduced additional defects. *ISO/IEC 90003:2004, Software engineering – Guidelines for the application of ISO 9001:2000 to computer software.3.11.* **3.** Functional testing that follows modification and maintenance" ISO/IEC/IEEE [149, page 295]

Smoke Test:

> "The smoke test (build verification test) focuses on test automation of the system components that make up the most important functionality. Instead of repeatedly retesting everything manually whenever a new software build is received, a test engineer plays back the smoke test, verifying that the major functionality of the system still exists." Dustin et al. [61, pages 43-44]

System Testing:

> "Testing conducted on a complete, integrated system to evaluate the systems compliance with its specified requirements. *IEEE Std 829-2008 IEEE Standard for Software and System Test Documentation. 3.1.37*" ISO/IEC/IEEE [149, page 361]

Test Coverage:

> "**1.** The degree to which a given test or set of tests addresses all specified requirements for a given system or component. **2.** Extent to which the test cases test the requirements for the system or software product. *ISO/IEC 12207:2008 (IEEE Std 12207-2008), Systems and software engineering – Software life cycle processes.4.51*" ISO/IEC/IEEE [149, page 369]

Instrumentation:

> "A common prerequisite for a number of debugging and performance-analysis techniques is the injection of auxiliary program code into the application under investigation, a process called *instrumentation*." Geimer et al. [92, page 696]

Unit Test:

> "**1.** Testing of individual routines and modules by the developer or an independent tester. **2.** A test of individual programs or modules in order to ensure that there are no analysis or programming errors. *ISO/IEC 2382-20:1990, Information technology – Vocabulary – Part 20: System development.20.05.05.* **3.** Test of individual hardware or software units or groups of related units" ISO/IEC/IEEE [149, page 386]

Program Slicing:

> "A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C constitute the *program slice with respect to criterion C*. The task of computing program slices is called *program slicing*.
>
> The original concept of a program slice was introduced by Weiser [...]. Weiser claims that a slice corresponds to the mental abstractions that people make when they are debugging a program,

> and advocates the integration of program slicers in debugging environments. [...] Weiser defined a program slice S as a *reduced, executable program* obtained from a program P by removing statements, such that S replicates part of the behavior of P. Another common definition of a slice is a *subset* of the statements and control predicates of the program that directly or indirectly affect the values computed at the criterion, but that do not necessarily constitute an executable program. An important distinction is that between a *static* and a *dynamic* slice. The former is computed without making assumptions regarding a programs input, whereas the latter relies on some specific test case." Tip [277, pages 1-2]

Backward Static Slices and Forward Static Slices:

> "The slices mentioned so far are computed by gathering statements and control predicates by way of a *backward* traversal of the programs control flow graph (CFG) or PDG, starting at the slicing criterion. Therefore, these slices are referred to as *backward* (static) slices. Bergeretti and Carré [...] were the first to define the notion of a *forward* static slice, although Reps and Bricker [...] were the first to use this terminology. Informally, a forward slice consists of all statements and control predicates dependent on the slicing criterion, a statement being "dependent" on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not. Backward and forward slices are computed in a similar way; the latter requires tracing dependences in the forward direction." Tip [277, page 3]

Dynamic Slicing:

> "In the case of dynamic program slicing, only the dependences that occur in a *specific* execution of the program are taken into account. A *dynamic slicing criterion* specifies the input, and distinguishes between different occurrences of a statement in the execution history; typically, it consists of triple (input, occurrence of a statement, variable). In other words, the difference between static and dynamic slicing is that dynamic slicing assumes *fixed* input for a program, whereas static slicing does not make assumptions regarding the input." Tip [277, page 3]

A.3 Graphs

Call Graph: According to Ryder [241, page 216], a call graph is a "directed graph".

> "The nodes of the graph are the procedures of the program; each edge represents one or more invocations of a procedure P_j by a procedure P_i." Ryder [241, page 216]

More formally:

> "[...] to form the call graph of a program we must examine all procedure definitions and their references, and determine all possible formal procedure parameter associations with external procedures. The reference relations between procedures in a program can be represented by a directed graph $G = \{N, E\}$ called a *call graph* where:
>
> 1. each node N_i corresponds in a one-to-one manner to a procedure P_i and its procedure vector set;
> 2. if P_i contains a reference $B_0(B_1, \ldots, B_k)$ then for each expansion $P_{j_0}(P_{j_1}, \ldots, P_{j_k})$ of that reference, there is a directed edge (N_i, N_{j_0}) in the graph and $(P_{j_1}, \ldots, P_{j_k})$ is in the procedure vector set of N_{j_0}."
>
> Ryder [241, page 219]

Please note: According to this definition, call graphs only model calls from one procedure to another. So it is a special CFG (see Section 2.4) (on procedure/method level). In our approach, we use CFGs on different levels, more precisely, on method, statement, and even on expression level, respectively.

Program Dependence Graph (PDG):

> "The PDG represents a program as a graph in which the nodes are statements and predicate expressions (or operators and operands) and the edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends. [...] The set of all dependences for a program may be viewed as inducing a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program.
>
> Dependences arise as the result of two separate effects. First, a dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. [...] Dependences of this type are *data* dependences. Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of the statement. [...] Dependences of this type are *control* dependences." Ferrante et al. [81, page 322]

System Dependence Graph (SDG):

> "A system dependence graph includes a *program dependence graph*, which represents the system's main program, *procedure dependence graphs*, which represent the system's auxiliary procedures, and some additional edges. These additional edges are of two sorts: (1) edges that represent direct dependences between a call site and the called procedure, and (2) edges that represent transitive dependences due to calls." Horwitz et al. [141, pages 35-36]

B Excerpt of a Source Map Created by GWT

```
1  # jsName, jsniIdent, className, memberName, sourceUri, sourceLine, fragmentNumber
2  $addStock,com.google.gwt.sample.stockwatcher.client.StockWatcher::$addStock(Lcom/google/gwt/sample/
       stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,
       $addStock,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,101,0
3  $onModuleLoad,com.google.gwt.sample.stockwatcher.client.StockWatcher::$onModuleLoad(Lcom/google/gwt/
       sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,
       $onModuleLoad,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,39,0
4  $refreshWatchList,com.google.gwt.sample.stockwatcher.client.StockWatcher::$refreshWatchList(Lcom/google
       /gwt/sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.
       StockWatcher,$refreshWatchList,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,147,0
5  $updateTable,com.google.gwt.sample.stockwatcher.client.StockWatcher::$updateTable(Lcom/google/gwt/
       sample/stockwatcher/client/StockWatcher;Lcom/google/gwt/sample/stockwatcher/client/StockPrice;)V,
       com.google.gwt.sample.stockwatcher.client.StockWatcher,$updateTable,com/google/gwt/sample/
       stockwatcher/client/StockWatcher.java,183,0
6  $updateTable_0,com.google.gwt.sample.stockwatcher.client.StockWatcher::$updateTable(Lcom/google/gwt/
       sample/stockwatcher/client/StockWatcher;[Lcom/google/gwt/sample/stockwatcher/client/StockPrice;)V,
       com.google.gwt.sample.stockwatcher.client.StockWatcher,$updateTable,com/google/gwt/sample/
       stockwatcher/client/StockWatcher.java,167,0
7  StockWatcher_0,com.google.gwt.sample.stockwatcher.client.StockWatcher::StockWatcher()V,com.google.gwt.
       sample.stockwatcher.client.StockWatcher,StockWatcher,com/google/gwt/sample/stockwatcher/client/
       StockWatcher.java,25,0
8  addPanel,com.google.gwt.sample.stockwatcher.client.StockWatcher::addPanel,com.google.gwt.sample.
       stockwatcher.client.StockWatcher,addPanel,com/google/gwt/sample/stockwatcher/client/StockWatcher.
       java,30,-1
9  addStockButton,com.google.gwt.sample.stockwatcher.client.StockWatcher::addStockButton,com.google.gwt.
       sample.stockwatcher.client.StockWatcher,addStockButton,com/google/gwt/sample/stockwatcher/client/
       StockWatcher.java,32,-1
10 lastUpdatedLabel,com.google.gwt.sample.stockwatcher.client.StockWatcher::lastUpdatedLabel,com.google.
       gwt.sample.stockwatcher.client.StockWatcher,lastUpdatedLabel,com/google/gwt/sample/stockwatcher/
       client/StockWatcher.java,33,-1
11 mainPanel,com.google.gwt.sample.stockwatcher.client.StockWatcher::mainPanel,com.google.gwt.sample.
       stockwatcher.client.StockWatcher,mainPanel,com/google/gwt/sample/stockwatcher/client/StockWatcher.
       java,28,-1
12 newSymbolTextBox,com.google.gwt.sample.stockwatcher.client.StockWatcher::newSymbolTextBox,com.google.
       gwt.sample.stockwatcher.client.StockWatcher,newSymbolTextBox,com/google/gwt/sample/stockwatcher/
       client/StockWatcher.java,31,-1
13 stocks,com.google.gwt.sample.stockwatcher.client.StockWatcher::stocks,com.google.gwt.sample.
       stockwatcher.client.StockWatcher,stocks,com/google/gwt/sample/stockwatcher/client/StockWatcher.
       java,34,-1
14 stocksFlexTable,com.google.gwt.sample.stockwatcher.client.StockWatcher::stocksFlexTable,com.google.gwt.
       sample.stockwatcher.client.StockWatcher,stocksFlexTable,com/google/gwt/sample/stockwatcher/client/
       StockWatcher.java,29,-1
15 StockWatcher$1,,com.google.gwt.sample.stockwatcher.client.StockWatcher$1,,com/google/gwt/sample/
       stockwatcher/client/StockWatcher.java,71,-1
16 StockWatcher$1_0,com.google.gwt.sample.stockwatcher.client.StockWatcher$1::StockWatcher$1(Lcom/google/
       gwt/sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.
       StockWatcher$1,StockWatcher$1,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,71,0
17 this$0,com.google.gwt.sample.stockwatcher.client.StockWatcher$1::this$0,com.google.gwt.sample.
       stockwatcher.client.StockWatcher$1,this$0,com/google/gwt/sample/stockwatcher/client/StockWatcher.
       java,71,-1
18 StockWatcher$2,,com.google.gwt.sample.stockwatcher.client.StockWatcher$2,,com/google/gwt/sample/
       stockwatcher/client/StockWatcher.java,80,-1
19 StockWatcher$2_0,com.google.gwt.sample.stockwatcher.client.StockWatcher$2::StockWatcher$2(Lcom/google/
       gwt/sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.
       StockWatcher$2,StockWatcher$2,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,80,0
20 onClick,com.google.gwt.sample.stockwatcher.client.StockWatcher$2::onClick(Lcom/google/gwt/event/dom/
       client/ClickEvent;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher$2,onClick,com/google/
       gwt/sample/stockwatcher/client/StockWatcher.java,81,0
```

Figure B.1: Excerpt of a Source Map Created by GWT, `PRETTY` variant.

```
# { 'user.agent' : 'gecko1_8' }
# jsName, jsniIdent, className, memberName, sourceUri, sourceLine, fragmentNumber
F,,com.google.gwt.core.client.JavaScriptException,,com/google/gwt/core/client/JavaScriptException.java,46,-1
Q,com.google.gwt.core.client.JavaScriptException::$clinit()V,com.google.gwt.core.client.JavaScriptException,$clinit,com/google/gwt/core/client/JavaScriptException.java,46,0
[...]
Y,com.google.gwt.core.client.JavaScriptObject::hashCode__devirtual$(Ljava/lang/Object;)I,com.google.gwt.core.client.JavaScriptObject,hashCode__devirtual$,com/google/gwt/emul/java/lang/Object.java,78,0
Z,com.google.gwt.core.client.JsDate::create(D)Lcom/google/gwt/core/client/JsDate;,com.google.gwt.core.client.JsDate,create,com/google/gwt/core/client/JsDate.java,35,0
$,,com.google.gwt.core.client.Scheduler,,com/google/gwt/core/client/Scheduler.java,33,-1
eb,com.google.gwt.core.client.impl.Impl::apply(Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;,com.google.gwt.core.client.impl.Impl,apply,com/google/gwt/core/client/impl/Impl.java,280,0
[...]
vb,com.google.gwt.core.client.impl.SchedulerImpl::push(Lcom/google/gwt/core/client/JsArray;Lcom/google/gwt/core/client/impl/SchedulerImpl$Task;)Lcom/google/gwt/core/client/JsArray;,com.google.gwt.core.client.impl.SchedulerImpl,push,com/google/gwt/core/client/impl/SchedulerImpl.java,144,0
xb,com.google.gwt.core.client.impl.StackTraceCreator::fillInStackTrace(Ljava/lang/Throwable;)V,com.google.gwt.core.client.impl.StackTraceCreator,fillInStackTrace,com/google/gwt/core/client/impl/StackTraceCreator.java,418,0
[...]
Fb,com.google.gwt.core.client.impl.StringBufferImplAppend::$append(Lcom/google/gwt/core/client/impl/StringBufferImplAppend;Ljava/lang/Object;I)V,com.google.gwt.core.client.impl.StringBufferImplAppend,$append,com/google/gwt/core/client/impl/StringBufferImplAppend.java,41,0
Ib,com.google.gwt.core.client.impl.StringBufferImplAppend::$replace(Lcom/google/gwt/core/client/impl/StringBufferImplAppend;Ljava/lang/Object;IILjava/lang/String;)V,com.google.gwt.core.client.impl.StringBufferImplAppend,$replace,com/google/gwt/core/client/impl/StringBufferImplAppend.java,71,0
[...]
Kb,com.google.gwt.core.client.impl.UnloadSupport::clearInterval0(I)V,com.google.gwt.core.client.impl.UnloadSupport,clearInterval0,com/google/gwt/core/client/impl/UnloadSupport.java,25,0
Lb,com.google.gwt.core.client.impl.UnloadSupport::clearTimeout0(I)V,com.google.gwt.core.client.impl.UnloadSupport,clearTimeout0,com/google/gwt/core/client/impl/UnloadSupport.java,29,0
Mb,com.google.gwt.core.client.impl.UnloadSupport::setInterval0(Lcom/google/gwt/core/client/JavaScriptObject;I)I,com.google.gwt.core.client.impl.UnloadSupport,setInterval0,com/google/gwt/core/client/impl/UnloadSupport.java,33,0
Nb,com.google.gwt.core.client.impl.UnloadSupport::setTimeout0(Lcom/google/gwt/core/client/JavaScriptObject;ILcom/google/gwt/core/client/impl/Disposable;)I,com.google.gwt.core.client.impl.UnloadSupport,setTimeout0,com/google/gwt/core/client/impl/UnloadSupport.java,40,0
[...]
Zc,,com.google.gwt.event.dom.client.KeyPressEvent,,com/google/gwt/event/dom/client/KeyPressEvent.java,23,-1
_c,com.google.gwt.event.dom.client.KeyPressEvent::$clinit()V,com.google.gwt.event.dom.client.KeyPressEvent,$clinit,com/google/gwt/event/dom/client/KeyPressEvent.java,23,0
ad,com.google.gwt.event.dom.client.KeyPressEvent::$dispatch(Lcom/google/gwt/event/dom/client/KeyPressEvent;Lcom/google/gwt/event/dom/client/KeyPressHandler;)V,com.google.gwt.event.dom.client.KeyPressEvent,$dispatch,com/google/gwt/event/dom/client/KeyPressEvent.java,78,0
[...]
cl,com.google.gwt.sample.stockwatcher.client.StockPrice::StockPrice(Ljava/lang/String;DD)V,com.google.gwt.sample.stockwatcher.client.StockPrice,StockPrice,com/google/gwt/sample/stockwatcher/client/StockPrice.java,12,0
a,com.google.gwt.sample.stockwatcher.client.StockPrice::change,com.google.gwt.sample.stockwatcher.client.StockPrice,change,com/google/gwt/sample/stockwatcher/client/StockPrice.java,7,-1
b,com.google.gwt.sample.stockwatcher.client.StockPrice::price,com.google.gwt.sample.stockwatcher.client.StockPrice,price,com/google/gwt/sample/stockwatcher/client/StockPrice.java,6,-1
c,com.google.gwt.sample.stockwatcher.client.StockPrice::symbol,com.google.gwt.sample.stockwatcher.client.StockPrice,symbol,com/google/gwt/sample/stockwatcher/client/StockPrice.java,5,-1
dl,,com.google.gwt.sample.stockwatcher.client.StockWatcher,,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,26,-1
el,com.google.gwt.sample.stockwatcher.client.StockWatcher::$addStock(Lcom/google/gwt/sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,$addStock,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,102,0
fl,com.google.gwt.sample.stockwatcher.client.StockWatcher::$onModuleLoad(Lcom/google/gwt/sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,$onModuleLoad,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,40,0
gl,com.google.gwt.sample.stockwatcher.client.StockWatcher::$refreshWatchList(Lcom/google/gwt/sample/stockwatcher/client/StockWatcher;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,$refreshWatchList,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,148,0
hl,com.google.gwt.sample.stockwatcher.client.StockWatcher::$updateTable(Lcom/google/gwt/sample/stockwatcher/client/StockWatcher;Lcom/google/gwt/sample/stockwatcher/client/StockPrice;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,$updateTable,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,184,0
il,com.google.gwt.sample.stockwatcher.client.StockWatcher::$updateTable(Lcom/google/gwt/sample/stockwatcher/client/StockWatcher;[Lcom/google/gwt/sample/stockwatcher/client/StockPrice;)V,com.google.gwt.sample.stockwatcher.client.StockWatcher,$updateTable,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,168,0
jl,com.google.gwt.sample.stockwatcher.client.StockWatcher::StockWatcher()V,com.google.gwt.sample.stockwatcher.client.StockWatcher,StockWatcher,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,26,0
a,com.google.gwt.sample.stockwatcher.client.StockWatcher::addPanel,com.google.gwt.sample.stockwatcher.client.StockWatcher,addPanel,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,31,-1
b,com.google.gwt.sample.stockwatcher.client.StockWatcher::addStockButton,com.google.gwt.sample.stockwatcher.client.StockWatcher,addStockButton,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,33,-1
c,com.google.gwt.sample.stockwatcher.client.StockWatcher::lastUpdatedLabel,com.google.gwt.sample.stockwatcher.client.StockWatcher,lastUpdatedLabel,com/google/gwt/sample/stockwatcher/client/StockWatcher.java,34,-1
[...]
```

Figure B.2: Transcompiled code: `OBFUSCATED` variant.

C Tries

> "A trie – pronounced "try" – is essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level l represents the set of all keys that begin with a certain sequence of l characters called its *prefix*; the node specifies an M-way branch, depending on the $(l + 1)$st character." Knuth [168, pages 492 - 495]

> "Trie memory for computer searching was first recommended by René de la Briandais [*Proc. Western Joint Computer Conf.* **15** (1959), 295-298]. He pointed out that we can save memory space at the expense of running time if we use a linked list for each node vector, since most of the entries in the vectors tend to be empty. [...] Searching [...] proceeds by finding the root that matches the first character, then finding the child node of that root that matches the second character, etc." Knuth [168, pages 494 - 495]

The search process becomes clear when looking at the example depicted in Figure C.1, which we have taken from Knuth [168, page 495].

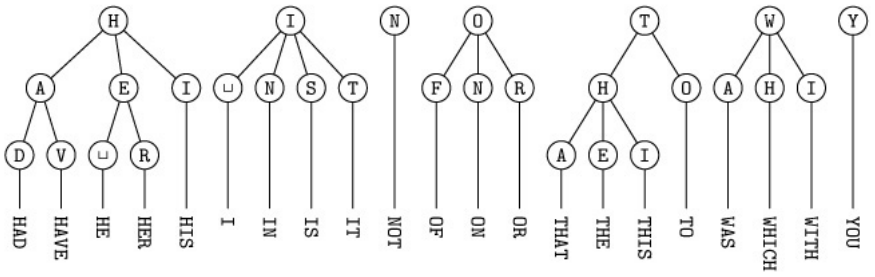


Figure C.1: Example of a trie, taken from Knuth [168, page 495].

Bibliography

[1] Abbot. Abbot framework for automated testing of Java GUI components and programs. `http://abbot.sourceforge.net/`, 2011. [Last access: 05th May, 2016].

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[3] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Comput. Surv.*, 14(2):159–192, June 1982.

[4] K. K. Aggrawal, Y. Singh, and A. Kaur. Code Coverage Based Technique for Prioritizing Test Cases for Regression Testing. *SIGSOFT Softw. Eng. Notes*, 29 (5):1–4, Sep. 2004.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, 1986.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[7] K. Alemerien and K. Magel. Examining the Effectiveness of Testing Coverage Tools: An Empirical Study. *International Journal of Software Engineering and Its Applications*, 8(5):139–162, 2014.

[8] F. E. Allen. Control Flow Analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[9] N. Alshahwan and M. Harman. Automated Session Data Repair for Web Application Regression Testing. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 298–307, Apr. 2008.

[10] J. W. Anderson, P. Lawson, M. Renschler, and M. Lange. csUnit. `http://www.csunit.org/`, 2009. [Last access: 30th Apr., 2017].

[11] Apache Santuario. XML Security. `http://santuario.apache.org/`, 2016. [Last access: 27th Jan., 2016].

[12] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, Mar. 2007.

[13] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`, Nov. 2005. [Last access: 04th Dec., 2013].

[14] R. Arnold and S. Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[15] A. Arora and M. Sinha. Web Application Testing: A Review on Techniques, Tools and State of Art. *International Journal of Scientific & Engineering Research*, 3(2):1–6, Feb. 2012.

[16] A. Asadullah, R. Mishra, M. Basavaraju, and N. Jain. A Call Trace Based Technique for Regression Test Selection of Enterprise Web Applications (SoRTEA). In *Proceedings of the 7th India Software Engineering Conference*, ISEC '14, pages 22:1–22:6, New York, NY, USA, 2014. ACM.

[17] B. Athira and P. Samuel. Web services regression test case prioritization. In *Computer Information Systems and Industrial Management Applications (CISIM), 2010 International Conference on*, pages 438–443, Oct 2010.

[18] Atlassian. Documentation for Clover 4.0. `https://confluence.atlassian.com/alldoc/files/71598770/650641563/1/1408328787139/CLOVER-4-0-20140818-PDF.pdf`, 2014. [Last access: 03rd Mar., 2016].

[19] Atlassian. Java Code Coverage. `https://www.atlassian.com/software/clover/overview`, 2016. [Last access: 03rd Mar., 2016].

[20] Atlassian. Managing the Coverage Database. `https://confluence.atlassian.com/display/CLOVER/Managing+the+Coverage+Database`, 2016. [Last access: 24th June, 2016].

[21] Atlassian. Using Clover with the GWT-maven plugin. `https://confluence.atlassian.com/display/CLOVER/Using+Clover+with+the+GWT-maven+plugin`, 2016. [Last access: 03rd Mar., 2016].

[22] Atlassian. Why does Clover use source code instrumentation? `https://confluence.atlassian.com/pages/viewpage.action?pageId=79986998`, Mar. 2016. [Last access: 03rd Mar., 2016].

[23] A. Ayers, A. Agarwal, and R. Schooler. Method for Back Tracing Program Execution, Mar. 2002. URL `https://www.google.com/patents/US6353924`. US Patent 6,353,924.

[24] G. K. Baah, A. Podgurski, and M. J. Harrold. The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 189–200, New York, NY, USA, 2008. ACM.

[25] T. Ball. On the Limit of Control Flow Analysis for Regression Test Selection. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 134–142, New York, NY, USA, 1998. ACM.

[26] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[27] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1999.

[28] P. Benedusi, A. Cimitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Proceedings of the Conference on Software Maintenance*, pages 352–361, 1988.

[29] J. Bible, G. Rothermel, and D. S. Rosenblum. A Comparative Study of Coarse-

and Fine-Grained Safe Regression Test-Selection Techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, Apr. 2001.

[30] H. Blau. JTopas. `http://jtopas.sourceforge.net/jtopas/`, Nov. 2004. [Last access: 27th Jan., 2016].

[31] B. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, Jan. 2001.

[32] C. Britton. Choosing a Programming Language. `https://msdn.microsoft.com/en-us/library/cc168615.aspx`, Jan. 2008. [Last access: 31th Mar., 2016].

[33] S. Chandel. Please Don't Repeat GWT's Mistake! GoogleGroups. `https://groups.google.com/d/msg/google-appengine/QsCMpKbyOJE/HbpgorMhgYgJ`, Oct. 2008. [Last access: 24th Mar., 2014].

[34] S. Chandel. Testing methodologies using GWT. `http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html`, Mar. 2009. [Last access: 25th Mar., 2014].

[35] S. Chandel. Is GWT's compiler java->javascript or java bytecode -> javascript? GoogleGroups. `https://groups.google.com/d/msg/google-web-toolkit/SIUZRZyvEPg/OaCGAfNAzzEJ`, July 2009. [Last access: 24th Mar., 2014].

[36] A. Chawla and A. Orso. A Generic Instrumentation Framework for Collecting Dynamic Information. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, Sep. 2004.

[37] L. Chen, Z. Wang, L. Xu, H. Lu, and B. Xu. Test Case Prioritization for Web Service Regression Testing. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 173–178, June 2010.

[38] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[39] O. C. Chesley, X. Ren, and B. G. Ryder. Crisp: A Debugging Tool for Java Programs. In *In Proc. International Conference on Software Maintenance (ICSM2005)*, pages 401–410, Sep. 2005.

[40] K.-R. Chilakamarri and S. Elbaum. Leveraging Disposable Instrumentation to Reduce Coverage Collection Overhead: Research Articles. *Softw. Test. Verif. Reliab.*, 16(4):267–288, Dec. 2006.

[41] P. K. Chittimalli and M. J. Harrold. Re-computing Coverage Information to Assist Regression Testing. In *2007 IEEE International Conference on Software Maintenance*, pages 164–173, Oct 2007.

[42] Cobertura. FAQ Cobertura Wiki. `https://github.com/cobertura/cobertura/wiki/FAQ`, Sep. 2014. [Last access: 23th Sep., 2017].

[43] Cobertura. Cobertura - A code coverage utility for Java. `http://cobertura.github.io/cobertura/`, Mar. 2016. [Last access: 03rd Mar., 2016].

[44] CodeCover. An open-source glass-box testing tool. `http://codecover.org/`, 2016. [Last access: 07th Mar., 2016].

[45] Codename One. Codename One - Cross-Platform Mobile Native Development Using Java. `https://www.codenameone.com/`, 2016. [Last access: 05th May, 2016].

[46] Codename One. Codename One - Advanced Topics - Under The Hood. `https://www.codenameone.com/manual/advanced-topics.html`, 2016. [Last access: 23th May, 2016].

[47] Codename One. Codename One - Manual: Introduction. `https://www.codenameone.com/manual/`, 2016. [Last access: 23th May, 2016].

[48] Codename One. Codename One - Getting Started. `https://www.codenameone.com/download.html`, 2016. [Last access: 23th May, 2016].

[49] Codename One. Codename One - Performance, Size & Debugging. `https://www.codenameone.com/manual/performance-debugging.html`, 2016. [Last access: 23th May, 2016].

[50] R. Cromwell. People sometimes ask me why Google itself doesn't use GWT. `https://plus.google.com/+RayCromwell/posts/ivVepvxCu3g`, Dec. 2011. [Last access: 14th July, 2014].

[51] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight Defect Localization for Java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag.

[52] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[53] DFB - Deutscher Fuball-Bund e.V. Sepp Herberger. `http://www.dfb.de/die-mannschaft/historie/bundestrainer/sepp-herberger/`, 2016. [Last access: 27th May, 2016].

[54] H. Do and G. Rothermel. An Empirical Study of Regression Testing Techniques Incorporating Context and Lifetime Factors and Improved Cost-benefit Models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 141–151, New York, NY, USA, 2006. ACM.

[55] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[56] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis. *Empirical Software Engineering*, 11(1):33–70, 2006.

[57] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An Empirical Study of the Effect of Time Constraints on the Cost-benefits of Regression Testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 71–82, New York, NY, USA, 2008. ACM.

[58] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments. *IEEE Trans. Softw. Eng.*, 36(5):593–617, Sep. 2010.

[59] Docker. Docker - Build, Ship, and Run Any App, Anywhere. `https://www.docker.com/`, 2017. [Last access: 06th July, 2017].

[60] S. Doğan, A. Betin-Can, and V. Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91:174 – 201, 2014.

[61] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Pearson Education, 1999.

[62] Eclipse. Abstract Syntax Tree. `http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html`, Nov. 2006. [Last access: 04th Dec., 2013].

[63] Eclipse. Abstract Syntax Tree - PHP Development Tools. `http://www.eclipse.org/pdt/articles/ast/PHP_AST.html`, May 2008. [Last access: 04th Apr., 2016].

[64] Eclipse. IBinding. `http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FIBinding.html`, 2013. [Last access: 04th Dec., 2013].

[65] Eclipse. Eclipse CompilationUnit. `http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FCompilationUnit.html`, 2013. [Last access: 30th June, 2016].

[66] Eclipse. JDT Core Component. `http://www.eclipse.org/jdt/core/index.php`, 2014. [Last access: 28th Apr., 2014].

[67] Eclipse. Eclipse Documentation: AST. `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAST.html`, 2014. [Last access: 14th Apr., 2017].

[68] Eclipse. PHP Development Tools (PDT). `http://www.eclipse.org/pdt/`, 2015. [Last access: 04th Apr., 2016].

[69] Eclipse. Eclipse Java Development Tools (JDT). `http://www.eclipse.org/jdt/`, 2016. [Last access: 04th Apr., 2016].

[70] Eclipse. Remote Application Platform. `http://www.eclipse.org/rap/`, Mar. 2016. [Last access: 02nd Mar., 2016].

[71] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing Test Cases for Regression Testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, New York, NY, USA, 2000. ACM.

[72] S. Elbaum, D. Gable, and G. Rothermel. The Impact of Software Evolution on Code Coverage Information. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 170–179, Washington, DC, USA, 2001. IEEE Computer Society.

[73] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[74] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, Feb. 2002.

[75] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Journal*, 12(3): 185–210, Sep. 2004.

[76] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging User-Session Data to Support Web Application Testing. *IEEE Trans. Softw. Eng.*, 31(3): 187–202, Mar. 2005.

[77] S. Elbaum, G. Rothermel, and J. Penix. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[78] E. Engström, P. Runeson, and M. Skoglund. A Systematic Review on Regression Test Selection Techniques. *Inf. Softw. Technol.*, 52(1):14–30, Jan. 2010.

[79] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke. Empirical Evaluation of Pareto Efficient Multi-objective Regression Test Case Prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 234–245, New York, NY, USA, 2015. ACM.

[80] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2): 335–361, 2013.

[81] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3): 319–349, July 1987.

[82] J.-C. Filliâtre. Deductive Software Verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[83] N. Fitzgerald and R. Nyman. Compiling to JavaScript, and Debugging with Source Maps. `https://hacks.mozilla.org/2013/05/compiling-to-javascript-and-debugging-with-source-maps/`, May 2013. [Last access: 19th May, 2016].

[84] M. Fowler. Continuous Integration. `http://martinfowler.com/articles/continuousIntegration.html`, May 2006. [Last access: 15th Sep., 2014].

[85] M. Fowler. Xunit. `https://www.martinfowler.com/bliki/Xunit.html`, Jan. 2006. [Last access: 30th Apr., 2017].

[86] M. Fowler. Eradicating Non-Determinism in Tests. `http://martinfowler.com/articles/nonDeterminism.html`, Apr. 2011. [Last access: 17th Apr., 2014].

[87] P. G. Frankl, G. Rothermel, K. Sayre, and F. I. Vokolos. An empirical comparison of two safe regression test selection techniques. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ISESE '03, pages 195–204, Washington, DC, USA, 2003. IEEE Computer Society.

[88] Freedekstop. CppUnit. `https://freedesktop.org/wiki/Software/cppunit/`, Apr. 2017. [Last access: 30th Apr., 2017].

[89] Gargoyle Software Inc. HtmlUnit. `http://htmlunit.sourceforge.net/`, Feb. 2014. [Last access: 19th Mar., 2014].

[90] J. J. Garrett. Ajax: A New Approach to Web Applications. `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/`, Feb. 2005. [Last access: 19th Mar., 2014].

[91] O. Gaudin. Measure Coverage by Integration Tests with Sonar Updated. `http://www.sonarqube.org/measure-coverage-by-integration-tests-with-sonar-updated/`, June 2012. [Last access: 17th May, 2016].

[92] M. Geimer, S. S. Shende, A. D. Malony, and F. Wolf. *A Generic and Configurable Source-Code Instrumentation Component*, pages 696–705. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[93] M. Gligoric, L. Eloussi, and D. Marinov. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM.

[94] G. Gokdogan. JsInterop v1.0: Nextgen GWT/JavaScript Interoperability. `https://docs.google.com/document/d/10fmlEYIHcyead_4R1S5wKGs1t2I7Fnp_PaNaa7XTEk0/edit?pref=2&pli=1`, Oct. 2015. [Last access: 23th May, 2016].

[95] Google. Understanding the GWT Compiler. https://developers.google.com/web-toolkit/doc/latest/DevGuideCompilingAndDebugging#DevGuideJavaToJavaScriptCompiler, Oct. 2012. [Last access: 13th Mar., 2013].

[96] Google. Add instrumentation for collecting client-side code coverage. https://github.com/gwtproject/gwt/commit/8549003236db60c0c70cdd61387293c5fe543616, July 2012. [Last access: 03rd Mar., 2016].

[97] Google. Coding Basics JSNI. http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html, Dec. 2013. [Last access: 11th Dec., 2013].

[98] Google. Overview. http://www.gwtproject.org/overview.html, Dec. 2013. [Last access: 03rd Dec., 2013].

[99] Google. UIBinder. http://www.gwtproject.org/doc/latest/DevGuideUiBinder.html, Dec. 2013. [Last access: 04th Dec., 2013].

[100] Google. Tutorial Stockwatcher. http://www.gwtproject.org/doc/latest/tutorial/gettingstarted.html, 2014. [Last access: 20th Apr., 2014].

[101] Google. Architecting Your App for Testing. http://www.gwtproject.org/doc/latest/DevGuideTesting.html, 2014. [Last access: 22th Apr., 2014].

[102] Google. Compiling Java to JavaScript. http://www.gwtproject.org/doc/latest/tutorial/compile.html, May 2014. [Last access: 21th May, 2014].

[103] Google. Deferred Binding Benefits. http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsDeferred.html, Apr. 2014. [Last access: 17th Apr., 2014].

[104] Google. Developing with GWT. http://www.gwtproject.org/overview.html#how, Mar. 2014. [Last access: 25th Mar., 2014].

[105] Google. Language support. http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsCompatibility.html, Apr. 2014. [Last access: 17th Apr., 2014].

[106] Google. Making GWT Better: Working with the Code. http://www.gwtproject.org/makinggwtbetter.html#workingoncode, Apr. 2014. [Last access: 27th Apr., 2014].

[107] Google. Styling Existing Widgets. http://www.gwtproject.org/doc/latest/DevGuideUiCss.html#widgets, May 2014. [Last access: 11th May, 2014].

[108] Google. Compiling and Debugging. http://www.gwtproject.org/doc/latest/DevGuideCompilingAndDebugging.html, June 2015. [Last access: 08th June, 2015].

[109] Google. Super Dev Mode. http://www.gwtproject.org/articles/superdevmode.html, Dec. 2015. [Last access: 22th May, 2016].

[110] Google. Closure Compiler. https://developers.google.com/closure/compiler/, 2016. [Last access: 23th May, 2016].

[111] Google. Code Coverage. http://www.gwtproject.org/doc/latest/DevGuideTestingCoverage.html#eclemma, 2016. [Last access: 03rd Mar., 2016].

[112] Google. FAQ - Debugging and Compiling. http://www.gwtproject.org/doc/latest/FAQ_DebuggingAndCompiling.html, May 2016. [Last access: 05th May, 2016].

[113] Google. Organizing Projects. http://www.gwtproject.org/doc/latest/DevGuideOrganizingProjects.html, 2016. [Last access: 22th May, 2016].

[114] Google. ScriptInjector (GWT Javadoc). `http://www.gwtproject.org/javadoc/latest/com/google/gwt/core/client/ScriptInjector.html`, 2016. [Last access: 22th May, 2016].

[115] Google. Client. `http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsClient.html`, Apr. 2017. [Last access: 15th Apr., 2017].

[116] Google. GWT Git repositories - Git at Google. `https://gwt.googlesource.com/`, Apr. 2017. [Last access: 02nd Apr., 2017].

[117] Google. Master - GWT - Git at Google. `https://gwt.googlesource.com/gwt/+/master`, Apr. 2017. [Last access: 02nd Apr., 2017].

[118] GraphStream Team. GraphStream - A Dynamic Graph Library. `http://graphstream-project.org/`, 2015. [Last access: 20th Apr., 2017].

[119] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[120] Grooscript. Grooscript - library to convert groovy code to javascript. `http://grooscript.org/`, 2017. [Last access: 07th Aug., 2017].

[121] O. Group. The Open Group Base Specifications Issue 7 - diff. `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html`, 2016. [Last access: 5th Mar., 2018].

[122] K. Grupp. Eclipse Plugin zur Ermittlung von Codeänderungen für Regressionstests. Bachelor's thesis, Wilhelm-Schickard Institut, 2013.

[123] S. B. Gupta. *Comprehensive Discrete Mathematics & Structures*. Laxmi Publications, 2005.

[124] R. Hanussek, S. Kie, T. Scheller, and M. Wittlinger. CodeCover: Glass Box Testing Tool Design. `http://codecover.org/development/Design.pdf`, May 2008. [Last access: 07th Mar., 2016].

[125] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun. Interactive Fault Localization Using Test Information. *Journal of Computer Science and Technology*, 24(5): 962–974, Sep. 2009.

[126] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression Test Selection for Java Software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

[127] C. Häubl, C. Wimmer, and H. Mössenböck. Deriving Code Coverage Information from Profiling Data Recorded for a Trace-based Just-in-time Compiler. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 1–12, New York, NY, USA, 2013. ACM.

[128] Haxe. Dead Code Elimination - Haxe. `http://haxe.org/manual/cr-dce.html`, May 2016. [Last access: 05th May, 2016].

[129] Haxe. Haxe - Global Compiler Flags. `https://haxe.org/manual/compiler-usage-flags.html`, May 2016. [Last access: 05th May, 2016].

[130] Haxe. Haxe - The Cross-platform Toolkit. `http://haxe.org/`, Mar. 2016. [Last access: 02nd Mar., 2016].

[131] Haxe. Haxe - Compiler Targets. `https://haxe.org/documentation/`

introduction/compiler-targets.html, Feb. 2017. [Last access: 27th Feb., 2017].

[132] P. Heidegger and P. Thiemann. Recency Types for Dynamically-Typed, Object-Based Languages . In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Jan. 2009.

[133] M. Hirzel. Selective Regression Testing for Web Applications Created with Google Web Toolkit. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 110–121, New York, NY, USA, 2014. ACM. http://doi.acm.org/10.1145/2647508.2647527.

[134] M. Hirzel and H. Klaeren. Code Coverage for Any Kind of Test in Any Kind of Transcompiled Cross-Platform Applications. In *Proceedings of the Second International Workshop on User Interface Test Automation*, INTUITEST 2016, pages 1–10, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2945404.2945405.

[135] M. Hirzel and H. Klaeren. Graph-Walk-based Selective Regression Testing of Web Applications Created with Google Web Toolkit. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016*, pages 55–69, 2016.

[136] M. Hirzel, J. I. Brachthäuser, and H. Klaeren. Prioritizing Regression Tests for Desktop and Web-Applications Based on the Execution Frequency of Modified Code. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2972206.2972222.

[137] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[138] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-Aware Trace Analysis. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 453–464, New York, NY, USA, 2009. ACM.

[139] M. R. Hoffmann, B. Janiczak, and E. Mandrikov. EclEmma - JaCoCo Java Code Coverage Library. http://eclemma.org/jacoco/, Feb. 2016. [Last access: 03rd Mar., 2016].

[140] M. R. Hoffmann, B. Janiczak, and E. Mandrikov. JaCoCo - Implementation Design. http://eclemma.org/jacoco/trunk/doc/implementation.html, Feb. 2016. [Last access: 03rd Mar., 2016].

[141] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[142] S. Huang, Y. Chen, J. Zhu, Z. J. Li, and H. F. Tan. An Optimized Change-driven Regression Testing Selection Strategy for Binary Java Applications. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 558–565, New York, NY, USA, 2009. ACM.

[143] S. Huang, J. Zhu, and Y. Ni. ORTS: A Tool for Optimized Regression Testing Selection. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 803–804, New York, NY, USA, 2009. ACM.

[144] Hupa. Overview. http://james.apache.org/hupa/index.html, June 2012. [Last access: 24th May, 2014].

[145] IEEE. IEEE Standard for Software Reviews and Audits. *IEEE STD 1028-2008*, pages 1–52, Aug 2008.

[146] Intel. Pin - A Dynamic Binary Instrumentation Tool. `http://www.pintool.org/`, June 2012. [Last access: 30th May, 2017].

[147] Intel. Pin 3.2 User Guide. `https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/`, Feb. 2017. [Last access: 30th May, 2017].

[148] Internet Engineering Task Force (IETF). RFC 6455. `https://tools.ietf.org/html/rfc6455`, Dec. 2011. [Last access: 17th June, 2016].

[149] ISO/IEC/IEEE. Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

[150] itdesign. Cloud PPM Software for Project Portfolios that Work. `https://meisterplan.com/`, 2017. [Last access: 10th Sep., 2017].

[151] J2ObjC. What J2ObjC is. `https://developers.google.com/j2objc/`, 2017. [Last access: 07th Aug., 2017].

[152] K. Jacoby and H. Layton. Automation of Program Debugging. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, pages 123.201–123.204, New York, NY, USA, 1961. ACM.

[153] D. Jeffrey and R. Gupta. Test Case Prioritization Using Relevant Slices. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, volume 1, pages 411–420, Sept 2006.

[154] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[155] S. H. Jensen, M. Madsen, and A. Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 59–69, New York, NY, USA, 2011. ACM.

[156] C. Jones. Software defect-removal efficiency. *Computer*, 29(4):94–95, Apr 1996.

[157] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[158] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *2007 IEEE International Conference on Software Maintenance*, pages 234–243, Oct 2007.

[159] JSFiddle. Create a new fiddle. `http://jsfiddle.net/`, 2016. [Last access: 23th May, 2016].

[160] JUnit. Overview. `http://junit.org/`, Feb. 2016. [Last access: 04th Mar., 2016].

[161] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh. An improved method for test case prioritization by incorporating historical test case data. *Science of Computer Programming*, 78(1):93 – 116, 2012.

[162] S. Khurshid, D. Marinov, G. Rothermel, T. Xie, W. Motycka, M. B. Dwyer, S. Elbaum, J. Hatcliff, H. Do, and A. Kinneer. Software-artifact Infrastructure Repository. `http://sir.unl.edu/`. [Last access: 03rd Jan., 2016].

[163] J.-M. Kim and A. Porter. A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *Proceedings of*

the 24th International Conference on Software Engineering, ICSE '02, pages 119–129, New York, NY, USA, 2002. ACM.

[164] J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Application Frequency. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 126–135, New York, NY, USA, 2000. ACM.

[165] Y. W. Kim. Efficient Use of Code Coverage in Large-scale Software Development. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pages 145–155. IBM Press, 2003.

[166] H. Klaeren. Skriptum Softwaretechnik (Entwurf), 2011. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.230.5157&rep=rep1&type=pdf`.

[167] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[168] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[169] Kuefler, Erik and Broyer, Thomas. GwtMockito: Better GWT unit testing. `https://github.com/google/gwtmockito`, May 2017. [Last access: 18th June, 2017].

[170] A. Kumar and R. Goel. Event Driven Test Case Selection for Regression Testing Web Applications. In *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on*, pages 121–127, March 2012.

[171] J. R. Larus. Efficient Program Tracing. *Computer*, 26(5):52–61, May 1993.

[172] J. R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM.

[173] J. Laski and W. Szermer. Identification of Program Modifications and its Applications in Software Maintenance. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 282–290, Nov 1992.

[174] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 583–594, New York, NY, USA, 2016. ACM.

[175] J. Lenz and N. Fitzgerald. Source Map Revision 3 Proposal. `https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-_2gc6fAH0KY0k/edit?pref=2&pli=1`, Feb. 2011. [Last access: 23th May, 2016].

[176] D. Leon and A. Podgurski. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 442–453, Washington, DC, USA, 2003. IEEE Computer Society.

[177] H. K. N. Leung and L. White. Insights into Regression Testing. In *Proceedings of the Conference on Software Maintenance 1989*, pages 60–69, October 1989.

[178] H. K. N. Leung and L. White. A Cost Model to Compare Regression Test Strategies. In *Proceedings of the Conference on Software Maintenance 1991*, pages 201–208, 1991.

[179] N. Li, X. Meng, J. Offutt, and L. Deng. Is Bytecode Instrumentation as Good as Source Code Instrumentation: An Empirical Study with Industrial Tools (Experience Report). In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 380–389, Nov 2013.

[180] 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.

[181] R. Lublinerman. Inside the compiler. `https://docs.google.com/presentation/d/1n0BSQGCBkxfHLzDVFCMyWjqTYuraUr09uc7n5n6JuLU/edit?pli=1#slide=id.p18`, Jan. 2015. [Last access: 27th Apr., 2015].

[182] J. I. Maletic and M. L. Collard. Supporting Source Code Difference Analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 210–219, Washington, DC, USA, 2004. IEEE Computer Society.

[183] P. Mayer and A. Schroeder. Cross-Language Code Analysis and Refactoring. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 94–103, Washington, DC, USA, 2012. IEEE Computer Society.

[184] T. J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[185] A. M. Memon and M. L. Soffa. Regression Testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 118–127, New York, NY, USA, 2003. ACM.

[186] A. Mesbah and A. van Deursen. Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[187] A. Mesbah, E. Bozdag, and A. v. Deursen. Crawling AJAX by Inferring User Interface State Changes. In *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, pages 122–134, Washington, DC, USA, 2008. IEEE Computer Society.

[188] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1):3:1–3:30, Mar. 2012.

[189] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[190] J. C. Miller and C. J. Maloney. Systematic Mistake Analysis of Digital Computer Programs. *Commun. ACM*, 6(2):58–63, Feb. 1963.

[191] S. Mirarab and L. Tahvildari. A Prioritization Approach for Software Test Cases Based on Bayesian Networks. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 276–290, Berlin, Heidelberg, 2007. Springer-Verlag.

[192] S. Mirarab and L. Tahvildari. An Empirical Study on Bayesian Network-based Approach for Test Case Prioritization. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 278–287, April 2008.

[193] S. Mirshokraie and A. Mesbah. JSART: Javascript Assertion-based Regression

Testing. In *Proceedings of the 12th International Conference on Web Engineering*, ICWE'12, pages 238–252, Berlin, Heidelberg, 2012. Springer-Verlag.

[194] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSEFT: Automated Javascript Unit Test Generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.

[195] Mountainminds. EclEmma - Java Code Coverage for Eclipse. `http://eclemma.org/`, Sep. 2012. [Last access: 03rd Mar., 2016].

[196] Mountainminds. EclEmma - Launching in Coverage Mode.html. `http://www.eclemma.org/userdoc/launching.html`, Sep. 2012. [Last access: 20th June, 2016].

[197] Mountainminds. JaCoCo - Coverage Counters. `http://eclemma.org/jacoco/trunk/doc/counters.html`, Apr. 2016. [Last access: 05th May, 2016].

[198] Mozilla. Firebug. `https://getfirebug.com/`, 2016. [Last access: 23th May, 2016].

[199] Mozilla. Source Map. `https://github.com/mozilla/source-map`, 2016. [Last access: 23th May, 2016].

[200] MSDN. Introduction to Instrumentation and Tracing. `https://msdn.microsoft.com/en-us/library/aa983649%28VS.71%29.aspx`, 2016. [Last access: 24th Apr., 2016].

[201] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[202] Neomades SAS. 5. Developing an application with NeoMAD - NeoMAD 3.9 documentation. `http://docs.neomades.com/en/3.9/user-guide/developing.html`, 2015. [Last access: 21th Apr., 2016].

[203] Neomades SAS. NeoMAD, cross platform mobile development. `http://neomades.com/en/`, 2016. [Last access: 21th Apr., 2016].

[204] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen, and T. N. Nguyen. iDiff: Interaction-based Program Differencing Tool. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 572–575, Washington, DC, USA, 2011. IEEE Computer Society.

[205] NUnit. NUnit - Home. `http://www.nunit.org/`, 2015. [Last access: 30th Apr., 2017].

[206] F. S. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah. Automatic Fault Localization for Client-side JavaScript. *Software Testing, Verification and Reliability*, 26(1):69–88, Jan. 2016.

[207] Opal. Opal - Ruby to JavaScript Compiler. `http://opalrb.com/`, 2017. [Last access: 07th Aug., 2017].

[208] J. Öqvist, G. Hedin, and B. Magnusson. Extraction-Based Regression Test Selection. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 5:1–5:10, New York, NY, USA, 2016. ACM.

[209] Oracle. JavaServer Faces Technology. `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`, 2014. [Last access: 22th May, 2016].

[210] Oracle. JCov. `https://wiki.openjdk.java.net/display/CodeTools/jcov`, Jan. 2016. [Last access: 03rd Mar., 2016].

[211] Oracle. JCov FAQ. `https://wiki.openjdk.java.net/display/CodeTools/JCov+FAQ`, Jan. 2016. [Last access: 03rd Mar., 2016].

[212] Oracle. JavaServer Pages Technology. `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`, 2016. [Last access: 22th Dec., 2016].

[213] A. Orso and G. Rothermel. Software Testing: A Research Travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

[214] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 128–137, New York, NY, USA, 2003. ACM.

[215] A. Orso, N. Shi, and M. J. Harrold. Scaling Regression Testing to Large Software Systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 241–251, New York, NY, USA, 2004. ACM.

[216] C. Panigrahi and R. Mall. An approach to prioritize the regression test cases of object-oriented programs. *CSI Transactions on ICT*, 1(2):159–173, 2013.

[217] C. Panigrahi and R. Mall. A heuristic-based regression test case prioritization approach for object-oriented programs. *Innovations in Systems and Software Engineering*, 10(3):155–163, 2014.

[218] Pivotal. Spring Framework. `http://projects.spring.io/spring-framework/`, 2016. [Last access: 22th Dec., 2016].

[219] T. Polychniatis, J. Hage, S. Jansen, E. Bouwers, and J. Visser. Detecting Cross-Language Dependencies Generically. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 349–352, Washington, DC, USA, 2013. IEEE Computer Society.

[220] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.

[221] Ranorex. Test Automation for GUI Testing. `https://www.ranorex.com/`, 2017. [Last access: 07th Aug., 2017].

[222] J. Reghunadh and N. Jain. Selecting the optimal programming language. `http://www.ibm.com/developerworks/library/wa-optimal/wa-optimal-pdf.pdf`, Sep. 2011. [Last access: 31th Mar., 2016].

[223] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby. Chianti: A Prototype Change Impact Analysis Tool for Java. Technical report, 2003.

[224] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[225] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[226] D. Roest, A. Mesbah, and A. v. Deursen. Regression Testing Ajax Applications: Coping with Dynamism. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 127–136, Washington, DC, USA, 2010. IEEE Computer Society.

[227] G. Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. PhD thesis, Clemson, SC, USA, 1996. AAI9703440.

[228] G. Rothermel and M. J. Harrold. A Safe, Efficient Algorithm for Regression Test Selection. In *Software Maintenance ,1993. CSM-93, Proceedings., Conference on*, pages 358–367, Sep 1993.

[229] G. Rothermel and M. J. Harrold. A Framework for Evaluating Regression Test Selection Techniques. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 201–210, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[230] G. Rothermel and M. J. Harrold. Selecting Regression Tests for Object-Oriented Software. In *Proceedings 1994 International Conference on Software Maintenance*, pages 14–25, Sep 1994.

[231] G. Rothermel and M. J. Harrold. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 169–184, New York, NY, USA, 1994. ACM.

[232] G. Rothermel and M. J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[233] G. Rothermel and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

[234] G. Rothermel and M. J. Harrold. Empirical Studies of a Safe Regression Test Selection Technique. *IEEE Trans. Softw. Eng.*, 24(6):401–419, June 1998.

[235] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test Case Prioritization: An Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–188, Washington, DC, USA, 1999. IEEE Computer Society.

[236] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression Test Selection for C++ Software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[237] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing Test Cases For Regression Testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, Oct 2001.

[238] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The Impact of Test Suite Granularity on the Cost-effectiveness of Regression Testing. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 130–140, New York, NY, USA, 2002. ACM.

[239] N. Rutar, C. B. Almazan, and J. S. Foster. A Comparison of Bug Finding Tools for Java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256, Nov 2004.

[240] M. Ruth and S. Tu. A Safe Regression Test Selection Technique for Web Services. In *Proceedings of the Second International Conference on Internet and Web Applications and Services*, ICIW '07, pages 47–52, Washington, DC, USA, 2007. IEEE Computer Society.

[241] B. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 5:216–226, 1979.

[242] B. G. Ryder and F. Tip. Change Impact Analysis for Object-oriented Programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. ACM.

[243] Sahi. Compare Sahi with Others. `http://sahi.co.in/compare-sahi-with-others/`, 2014. [Last access: 10th May, 2014].

[244] Sahi. Test Automation Tool For Browser Based Web Applications - Sahi. `http://sahi.co.in/`, 2014. [Last access: 19th Mar., 2014].

[245] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru. Prioritizing User-Session-Based Test Cases for Web Applications Testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 141–150, Washington, DC, USA, 2008. IEEE Computer Society.

[246] S. Sampath, R. Bryce, and A. M. Memon. A Uniform Representation of Hybrid Criteria for Regression Testing. *IEEE Transactions on Software Engineering*, 39 (10):1326–1344, Oct 2013.

[247] S. R. Schach. *Practical Software Engineering*. The Aksen Associates series in electrical and computer engineering. Taylor & Francis, 1992.

[248] S. Schumm. Praxistaugliche Unterstützung beim selektiven Regressionstest. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Oktober 2009. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2923&engl=0`.

[249] Sebastian Bergmann. PHPUnit The PHP Testing Framework. `https://phpunit.de/`, 2017. [Last access: 30th Apr., 2017].

[250] Selendriod. Selendroid - Selenium for Android. `http://selendroid.io/`, 2015. [Last access: 05th May, 2016].

[251] SeleniumHQ. Selenium-IDE. `http://docs.seleniumhq.org/docs/02_selenium_ide.jsp`, May 2014. [Last access: 31th May, 2014].

[252] SeleniumHQ. Selenium - Web Browser Automation. `http://docs.seleniumhq.org/`, 2014. [Last access: 19th Mar., 2014].

[253] SeleniumHQ. Selenium WebDriver. `http://www.seleniumhq.org/docs/03_webdriver.jsp#htmlunit-driver`, Oct. 2016. [Last access: 06th Nov., 2016].

[254] Semantic Designs. Java Test Coverage Tool. `http://www.semanticdesigns.com/Products/TestCoverage/JavaTestCoverage.html?Home=TestCoverage`, 2015. [Last access: 03rd Mar., 2016].

[255] Sencha. GXT - Java Framework for Building Web Apps Using Google Web Toolkit. `http://www.sencha.com/products/gxt/`, 2016. [Last access: 23th May, 2016].

[256] P. Sestoft. *Programming Language Concepts*. Undergraduate Topics in Computer Science. Springer International Publishing, 2nd edition, 2017.

[257] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[258] SmartBear Software. TestComplete. `http://smartbear.com/product/testcomplete/overview/`, 2015. [Last access: 19th Dec., 2015].

[259] H. M. Sneed. Software-Testen, Stand der Technik. *Informatik Spektrum*, 11(6): 303–311, 1988.

[260] E. Soechting, K. Dobolyi, and W. Weimer. Syntactic Regression Testing for Tree-Structured Output. In *2009 11th IEEE International Symposium on Web Systems Evolution*, pages 39–48, Sept 2009.

[261] SoftwareVerify. C++ code coverage. `http://www.softwareverify.com/cpp-coverage.php`, Mar. 2016. [Last access: 03rd Mar., 2016].

[262] M. Soltys. *An Introduction to the Analysis of Algorithms.*, volume 2nd ed. World Scientific Publishing Company, 2012.

[263] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated Replay and Failure Detection for Web Applications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 253–262, New York, NY, USA, 2005. ACM.

[264] A. Srivastava and J. Thiagarajan. Effectively Prioritizing Tests in Development Environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.

[265] Steve Hannah. Websockets library for Codename One. `https://github.com/shannah/cn1-websockets`, 2015. [Last access: 05th May, 2016].

[266] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 57–68, New York, NY, USA, 2006. ACM.

[267] D. Strein, H. Kratz, and W. Lowe. Cross-Language Program Analysis and Refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 207–216, Washington, DC, USA, 2006. IEEE Computer Society.

[268] I. Sun Microsystems. Java Look And Feel Design Guidelines. `http://www.oracle.com/technetwork/java/hig-142147.html?printOnly=1`, 2001. [Last access: 20th Dec., 2017].

[269] P. G. Taboada. GWT Reference List. `http://gwtreferencelist.appspot.com/`, June 2014. [Last access: 14th July, 2014].

[270] S. Tallam and N. Gupta. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. *SIGSOFT Softw. Eng. Notes*, 31(1):35–42, Sep. 2005.

[271] N. Talle. Unit testing framework for Javascript - Unit JS. `http://unitjs.com/`, 2014. [Last access: 30th Apr., 2017].

[272] A. Tarhini, Z. Ismail, and N. Mansour. Regression Testing Web Applications. *Advanced Computer Theory and Engineering, International Conference on*, 0: 902–906, 2008.

[273] TestNG. Welcome. `http://testng.org/`, Dec. 2015. [Last access: 03rd Mar., 2016].

[274] TestNG. Documentation - TestNG Test-Groups. `http://testng.org/doc/documentation-main.html#test-groups`, 2016. [Last access: 11th June, 2016].

[275] The Apache Software Foundation. Apache JMeter. `http://jmeter.apache.org/`, 2016. [Last access: 27th Jan., 2016].

[276] M. M. Tikir and J. K. Hollingsworth. Efficient Instrumentation for Code Coverage Testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 86–96, New York, NY, USA, 2002. ACM.

[277] F. Tip. A Survey of Program Slicing Techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994. Also available as `http://www.franktip.org/pubs/jpl1995.pdf`.

[278] UCLA Compilers Group. JTB - The Java Tree Builder. `http://compilers.cs.ucla.edu/jtb/`, 2016. [Last access: 24th June, 2016].

[279] Vaadin. Vaadin User Interface Components for Business Apps. `https://vaadin.com/`, 2016. [Last access: 23th May, 2016].

[280] F. I. Vokolos and P. G. Frankl. *Pythia: A Regression Test Selection Tool Based on Textual Differencing*, pages 3–21. Springer US, Boston, MA, 1997.

[281] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-Aware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 1–12, New York, NY, USA, 2006. ACM.

[282] Watir. Watir: Web Application Testing in Ruby. `http://watir.com/`, 2014. [Last access: 19th Mar., 2014].

[283] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[284] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A Study of Effective Regression Testing in Practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274, Nov 1997.

[285] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *IEEE Trans. Softw. Eng.*, 42(8):707–740, Aug. 2016.

[286] Xamarin Inc. Get started with Xamarin. `https://www.xamarin.com/`, 2016. [Last access: 05th May, 2016].

[287] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[288] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression Testing for Web Applications Based on Slicing. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, COMPSAC '03, pages 652–656, Washington, DC, USA, 2003. IEEE Computer Society.

[289] S. Yoo and M. Harman. Pareto Efficient Multi-objective Test Case Selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 140–150, New York, NY, USA, 2007. ACM.

[290] S. Yoo and M. Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[291] S. Yoo, M. Harman, and D. Clark. Fault Localization Prioritization: Comparing Information-Theoretic and Coverage-Based Approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19:1–19:29, July 2013.

[292] Y. Yu, J. A. Jones, and M. J. Harrold. An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 201–210, New York, NY, USA, 2008. ACM.

[293] L. Zhang, M. Kim, and S. Khurshid. Localizing Failure-Inducing Program Edits Based on Spectrum Information. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.

[294] L. Zhang, M. Kim, and S. Khurshid. FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs. In *Proceedings of the*

ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 40:1–40:4, New York, NY, USA, 2012. ACM.

[295] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the Gap Between the Total and Additional Test-case Prioritization Strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, Piscataway, NJ, USA, 2013. IEEE Press.

[296] X. Zhao, Z. Wang, X. Fan, and Z. Wang. A Clustering-Bayesian Network Based Approach for Test Case Prioritization. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 3, pages 542–547, July 2015.