© 2015 Milos Gligoric

REGRESSION TEST SELECTION: THEORY AND PRACTICE

BY

MILOS GLIGORIC

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Associate Professor Darko Marinov, Chair
Professor Grigore Roşu
Professor Josep Torrellas
Associate Professor Sarfraz Khurshid, The University of Texas at Austin
Dr. Rupak Majumdar, MPI for Software Systems

ABSTRACT

Software affects every aspect of our lives, and software developers write tests to check software correctness. Software also rapidly evolves due to never-ending requirement changes, and software developers practice regression testing – running tests against the latest project revision to check that project changes did not break any functionality. While regression testing is important, it is also time-consuming due to the number of both tests and revisions.

Regression test selection (RTS) speeds up regression testing by selecting to run only tests that are affected by project changes. RTS is efficient if the time to select tests is smaller than the time to run unselected tests; RTS is safe if it guarantees that unselected tests cannot be affected by the changes; and RTS is precise if tests that are not affected are also unselected. Although many RTS techniques have been proposed in research, these techniques have not been adopted in practice because they do not provide efficiency and safety at once.

This dissertation presents three main bodies of research to motivate, introduce, and improve a novel, efficient, and safe RTS technique, called Ekstazi. Ekstazi is the first RTS technique being adopted by popular open-source projects.

First, this dissertation reports on the first field study of test selection. The study of logs, recorded in real time from a diverse group of developers, finds that almost all developers perform manual RTS, i.e., manually select to run a subset of tests at each revision, and they select these tests in mostly ad hoc ways. Specifically, the study finds that manual RTS is not safe 74% of the time and not precise 73% of the time. These findings showed the urgent need for a better automated RTS techniques that could be adopted in practice.

Second, this dissertation introduces Ekstazi, a novel RTS technique that is efficient and safe. Ekstazi tracks dynamic dependencies of tests on files, and unlike most prior RTS techniques, Ekstazi requires no integration with version-control systems. Ekstazi

computes for each test what files it depends on; the files can be either executable code or external resources. A test need not be run in the new project revision if none of its dependent files changed. This dissertation also describes an implementation of Ekstazi for the Java programming language and the JUnit testing framework, and presents an extensive evaluation of Ekstazi on 615 revisions of 32 open-source projects (totaling almost 5M lines of code) with shorter- and longer-running test suites. The results show that Ekstazi reduced the testing time by 32% on average (and by 54% for longer-running test suites) compared to executing all tests. Ekstazi also yields lower testing time than the existing RTS techniques, despite the fact that Ekstazi may select more tests. Ekstazi is the first RTS tool adopted by several popular open-source projects, including Apache Camel, Apache Commons Math, and Apache CXF.

Third, this dissertation presents a novel approach that improves precision of any RTS technique for projects with distributed software histories. The approach considers multiple old revisions, unlike all prior RTS techniques that reasoned about changes between two revisions – an old revision and a new revision – when selecting tests, effectively assuming a development process where changes occur in a linear sequence (as was common for CVS and SVN). However, most projects nowadays follow a development process that uses distributed version-control systems (such as Git). Software histories are generally modeled as directed graphs; in addition to changes occurring linearly, multiple revisions can be related by other commands such as branch, merge, rebase, cherry-pick, revert, etc. The novel approach reasons about commands that create each revision and selects tests for a new revision by considering multiple old revisions. This dissertation also proves the safety of the approach and presents evaluation on several open-source projects. The results show that the approach can reduce the number of selected tests over an order of magnitude for merge revisions.

To my family

ACKNOWLEDGMENTS

First of all, I would like to thank my adviser, *Darko Marinov*. If it were not for him, I would neither have started nor finished my PhD. Darko accepted me as an intern back in 2007 and as a PhD student later in 2009. I very much enjoyed spending time and working with him (did you know that Darko once went to a restaurant where they literally take shoes off?). I have to thank him for numerous nights of working in the lab or talking on the phone. Although I could never do enough to return all that he has done for me, I hope to advise students with the same passion in the future.

I would like to thank Sarfraz Khurshid, Rupak Majumdar, Grigore Rosu, and Josep Torrellas for their generous help during my graduate studies. They served on my thesis committee and helped me improve presentation of this material.

None of the projects would have been fun and success without my collaborators Gul Agha, Mohammad Amin Alipour, Elton Alves, Andrea Arcuri, Sandro Badame, Farnaz Behrang, Marcelo d'Amorim, Lamyaa Eloussi, Gordon Fraser, Alex Groce, Tihomir Gvero, Alex Gyori, Munawar Hafiz, Daniel Jackson, Vilas Jagannath, Dongyun Jin, Ralph Johnson, Owolabi Legunsen, Sam Kamin, Sarfraz Khurshid, Viktor Kuncak, Steven Lauterburg, Yilong Li, Benjamin Livshits, Qingzhou Luo, Rupak Majumdar, Aleksandar Milicevic, Peter C. Mehlitz, Iman Narasamdya, Stas Negara, Jeff Overbey, Cristiano Pereira, Gilles Pokam, Chandra Prasad, Grigore Rosu, Wolfram Schulte, Rohan Sharma, August Shi, Samira Tasharofi, Danny van Velzen, Andrey Zaytsev, Chaoqiang Zhang, and Lingming Zhang. I look forward to working with them again in the future.

My internship mentors also provided great support and lovely environments: Peter Mehlitz (NASA), Cristiano Pereira (Intel), Gilles Pokam (Intel), Rupak Majumdar (MPI), Benjamin Livshits (MSR), Chandra Prasad (Microsoft), and Wolfram Schulte (Microsoft).

I was honored to work with several exceptional undergraduate and master's students, including Sandro Badame, Chris Baker, Anirudh Balagopal, Jordan Ebel, Lamyaa Eloussi, Josh Friedman, Yilong Li, Dan Schweikert, Rohan Sehgal, Rohan Sharma, Nikhil Unni, and Andrey Zaytsev. In particular, my thanks go to Rohan Sharma and Lamyaa Eloussi for their effort to make each project an exceptional experience.

A good office mate is a must if one is to enjoy time during grad school. I was fortunate to have exceptionally good office mates during my stay at UIUC: Nima Honarmand, Dmytro Suvorov, and Yu Lin. In particular, I would like to thank Yu Lin for never complaining about my stories, typing, interruptions, and meetings. If I could take Yu with me to my next office, I would happily do so.

Andrei Stefanescu was my gym mate for several years. Although I will give my best to continue workouts, I am sure that gym (and sandwiches after the workouts) will not be the same without him.

I would like to thank other colleagues and friends, including Milos Cvetanovic, Brett Daniel, Danny Dig, Farah Hariri, Yun Young Lee, Daniella Marinov, Lara Marinov, Sasa Misailovic, Bosko Nikolic, Vladimir Petrovic, Zaharije Radivojevic, Cosmin Radoi, Ivan Rajic, Milos Vasiljevic, Yuanrui Zhang, and Wenxuan Zhou. Also, I thank many colleagues and friends at my undergraduate institution for the great time that we had together. Vladimir Petrovic designed several nice logos for my (research) projects.

I would like to thank my extended family and Jessy's family for their unrelenting support. I thank Dejan Stevanovic for sending pictures, jokes, and videos.

University of Illinois has an amazing staff, including Holly Bagwell, Candy Foster, Mary Beth Kelley, Viveka P. Kudaligama, Kara MacGregor, Rhonda McElroy, Colin Robertson, Kathy Runck, and Elaine Wilson. Elaine Wilson helped me with all the paperwork for travels and reimbursements. No words are enough to thank her for making my life easier.

Several faculty members at UIUC helped me prepare for my job interviews, including Gul Agha, Chandra Chekuri, Elsa L. Gunter, Indranil Gupta, Andreas Klöckner, Ranjitha Kumar, P. Madhusudan, Manoj M. Prabhakaran, Dan Roth, Rob A. Rutenbar, Hari Sundaram, Mahesh Viswanathan, Tandy Warnow, and Tao Xie. My hosts at the interviews ensured that I feel comfortable and everything runs smoothly.

Parts of this dissertation were published at ASE 2014 [78] (Chapter 2); ICSE Demo 2015 [74] and ISSTA 2015 [75] (Chapter 3); and CAV 2014 [76] and a technical report [77] (Chapter 4). I would like to thank the audience at the conferences for their comments, which were used to improve the presentation in this dissertation. Additionally, I would like to thank anonymous reviewers of the conference papers for their invaluable comments. Last but not least, I am honored that the ISSTA 2015 paper on Ekstazi won an ACM SIGSOFT Distinguished Paper Award.

My research was funded by Saburo Muroga Fellowship, C.L. & Jane W-S. Liu Award for Exceptional Research Promise, C.W. Gear Outstanding Graduate Student Award for Excellence in Research and Service, Mavis Future Faculty Fellowship, IBM X10 Innovation Grant, Universal Parallel Computing Research Center, and National Science Foundation.

Last but not least, I would like to thank my sister *Mirjana*, my mother *Vera*, my father *Zivko*, and *Jessy* for their never-ending love, care, and support.

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ABBREVIATIONS xii

CHAPTER 1 Introduction 1
- 1.1 Thesis Statement 3
- 1.2 Contributions 3
- 1.3 Study of Manual RTS 5
- 1.4 Novel RTS Technique for Two Code Revisions 6
- 1.5 Novel RTS Approach for Distributed Software Histories 8
- 1.6 Dissertation Organization 10

CHAPTER 2 Manual Regression Test Selection 11
- 2.1 Evaluating Manual RTS 11
- 2.2 Manual vs. Automated RTS 23
- 2.3 Discussion 31
- 2.4 Threats to Validity 33
- 2.5 Summary 34

CHAPTER 3 Regression Test Selection with Dynamic File Dependencies 35
- 3.1 Example 35
- 3.2 Technique and Implementation 38
- 3.3 Evaluation 45
- 3.4 Usage 57
- 3.5 Discussion 60
- 3.6 Threats to Validity 62
- 3.7 Summary 64

CHAPTER 4 Regression Test Selection for Distributed Software Histories 65
- 4.1 Example 65
- 4.2 Test Selection Technique 71
- 4.3 Proofs of Theorems 79
- 4.4 Evaluation 82
- 4.5 Summary 89

CHAPTER 5 Related Work 90
5.1 Manual Regression Test Selection 90
5.2 Build Systems and Memoization 91
5.3 Class-based Test Selection 92
5.4 External Resources 93
5.5 Code Instrumentation 93
5.6 Granularity Levels 94
5.7 Other Work on RTS for Two Revisions 95
5.8 Distributed Version Control Systems 96

CHAPTER 6 Conclusions and Future Work 97

REFERENCES 102

LIST OF TABLES

2.1 Statistics for projects used in the study of manual RTS 12
2.2 RTS capabilities of popular IDEs 23

3.1 Statistics for projects used in the evaluation of Ekstazi 47
3.2 Test selection results using Ekstazi 48
3.3 Legend for symbols used in tables 49
3.4 Ekstazi without and with smart checksum 53
3.5 Ekstazi with method and class selection granularity 54
3.6 Test selection with `FaultTracer` and Ekstazi 55
3.7 Current Ekstazi users 56

4.1 Statistics for several projects that use Git 83
4.2 Statistics for projects used in the evaluation of $\mathbb{S}^1_{merge}$, $\mathbb{S}^k_{merge}$, and $\mathbb{S}^0_{merge}$ . . . 84

LIST OF FIGURES

1.1 Part of software history of the Linux Kernel . . . 8

2.1 Algorithm for computing a set of available test methods at each test session . 16
2.2 Distribution of test selection ratio with and without single-test sessions 18
2.3 Relationship between time to execute unselected tests and selection time . . . 21
2.4 The number of tests selected by manual and automated RTS for $\mathcal{P}_{14}$ 27
2.5 Distribution of selected tests for $\mathcal{P}_{14}$ with and without single-test sessions . . . 28
2.6 Relationship of RTS with relative size of code changes for $\mathcal{P}_{14}$ . . . 29
2.7 Distribution for $\mathcal{P}_{14}$ of selection, analysis, and execution time . . . 31
2.8 Example of a pattern when developer alternates selection and regular runs . . 33

3.1 Example test code (left) and code under test (right) . . . 36
3.2 Dependency matrices collected for code in Figure 3.1 . . . 36
3.3 Integration of an RTS technique in a typical build with a testing framework . 38
3.4 Number of available and selected tests (a,c) and end-to-end time (b,d) 50
3.5 End-to-end `mvn test` time for Apache CXF . . . 57

4.1 Example of a distributed software history . . . 66
4.2 Extension of a software history (Figure 4.1) with `git merge` $\mathtt{b_1}$ $\mathtt{b_2}$ . . . 67
4.3 Extension of a software history (Figure 4.1) with `git rebase` $\mathtt{b_1}$ . . . 69
4.4 Extension of a software history (Figure 4.1) with `git cherry-pick` $\mathtt{n_2}$ 70
4.5 Extension of a software history (Figure 4.1) with `git revert` $\mathtt{n_6}$ . . . 70
4.6 $\mathbb{S}^k_{merge}$ may select more tests than $\mathbb{S}^1_{merge}$; $n_1 = n_4$ and $\delta_1(m) \neq \delta_2(m)$ 75
4.7 Example history to show that using lca (n_2) rather than dom (n_1) is not safe . 82
4.8 Percentage of selected tests for real merges using various options . . . 84
4.9 History statistics of projects used for generated software histories . . . 87
4.10 $\mathbb{S}^1_{merge}/\mathbb{S}^0_{merge}$ (speedup) for various numbers of commits in each branch 88

LIST OF ABBREVIATIONS

$\mathcal{A}$	Analysis Phase
$\mathcal{AE}$	Analysis and Execution Phases
$\mathcal{AEC}$	Analysis, Execution, and Collection Phases
API	Application Programming Interface
AST	Abstract Syntax Tree
$\mathcal{C}$	Collection Phase
DVCS	Distributed Version Control System
$\mathcal{E}$	Execution Phase
ECFG	Extended Control-Flow Graph
GUI	Graphical User Interface
IDE	Integrated Development Environment
JVM	Java Virtual Machine
LOC	Lines of Code
RTS	Regression Test Selection
tts	Traditional Test Selection
VCS	Version Control System

CHAPTER 1

Introduction

Software controls every aspect of our lives, e.g., ranging from communication to social networks to entertainment to business to transportation to health. Therefore, software correctness is of utmost importance. Correctness issues – bugs – in software may lead to significant financial losses and casualties. We have witnessed the high cost of bugs far too many times. Prior studies estimate that bugs cost global economy more than $300 billion per year [10,46].

Despite the risk of introducing new bugs while making changes, software constantly evolves due to never-ending requirements. Thus, software developers have to check, at each project revision, not only correctness of newly added functionality, but also that the recent project changes did not break any previously working functionality.

Software testing is the most common approach in industry to check correctness of software. Software developers usually write tests for newly implemented functionality and include these tests in a test suite (i.e., a set of tests for the entire project). To check that project changes did not break previously working functionality, developers practice *regression testing* – running test suite at each project revision.

Although regression testing is important, it is costly because it frequently runs a large number of tests. Some studies [38, 48, 67, 109, 117] estimate that regression testing can take up to 80% of the testing budget and up to 50% of the software maintenance cost. The cost of regression testing increases as software grows. For example, Google reported that their regression-testing system, TAP [65, 146, 149], has had a linear increase in both the number of project changes per day and the average test-suite execution time per change, leading to a quadratic increase in the total test-suite execution time per day. As a result, the increase is challenging to keep up with even for a company with an abundance of computing resources. Other companies and open-source projects also reported long regression testing time [47,94].

Regression test selection (RTS) is a promising approach to speed up regression testing. Researchers have proposed many RTS techniques (e.g., [69, 82, 89–91, 138, 161]); Engström et al. [68] present a survey of RTS, and Yoo and Harman [157] present an extensive survey of regression testing including RTS. A traditional RTS technique takes four inputs—two project revisions[1] (new and old), test suite at the new revision, and dependency information from the test runs on the old revision—and produces, as output, a subset of the test suite for the new revision. The subset includes the tests that can be affected by the changes; viewed dually, the subset excludes the tests that cannot be affected by the changes and thus need not be rerun on the new revision. RTS is *efficient* if selecting tests takes less time than total running time of unselected tests, *precise* if tests that are not affected are also unselected, and *safe* if it guarantees that selected tests exclude only tests whose behavior cannot be affected by the changes.

While RTS was proposed over three decades ago [68, 71, 157], it has not been widely adopted in practice, except for the substantial success of the Google TAP system [65,146,149] and the Microsoft Echelon system [92,93,144]. Unfortunately, TAP performs RTS only *across projects*, e.g., the YouTube project depends on the Guava project, so *all* YouTube and *all* Guava tests are run if anything in Guava changes, but *all* YouTube and *no* Guava tests are run if anything in YouTube changes. In other words, TAP provides no benefit *within a project*. However, most developers work on one isolated project at a time rather than on a project from a huge codebase as done at Google. Such smaller projects would require a finer-grained technique for more precise RTS. On the other side, although Echelon is more precise (as it tracks basic blocks), it only prioritizes [66,139] but does not select tests to run.

The lack of practical RTS tools leaves two options for developers: either automatically rerun all the tests or manually perform test selection. Rerunning all the tests is safe by definition, but it can be quite imprecise and, therefore, inefficient. In contrast, manual test selection, which we will refer to as *manual RTS*, can be unsafe and imprecise: developers can select too few tests and thus miss to run some tests whose behavior differs due to code changes, or developers can select too many tests and thus waste time. In sum, a large number of developers would benefit from an automated RTS technique that works in practice.

[1]We use commit and revision interchangeably to refer to a single node in a software history graph.

The key requirement for an RTS technique to be adopted is that the *end-to-end time* is shorter than the time to run all tests in the testing framework [110, 113], while guaranteeing safety. A typical RTS technique has three phases: the *analysis ($\mathcal{A}$) phase* selects tests to run, the *execution ($\mathcal{E}$) phase* runs the selected tests, and the *collection ($\mathcal{C}$) phase* collects information from the current revision to enable the analysis for the next revision. Most research has evaluated RTS techniques based on the number of selected tests, i.e., implicitly based on the time only for the $\mathcal{E}$ phase; a few papers that do report time (e.g., [124, 154]) measure only $\mathcal{A}$ and $\mathcal{E}$ phases, ignoring the $\mathcal{C}$ phase. To properly compare speed up (or slow down) of RTS techniques, we believe it is important to consider the *end-to-end time* ($\mathcal{A} + \mathcal{E} + \mathcal{C}$) that the developer observes, from initiating the test-suite execution for a new code revision until all test outcomes become available.

1.1 Thesis Statement

Our thesis is three-pronged:

(1) There is a need for an automated RTS technique that works in practice.

(2) It is possible to design and develop a safe *and* efficient *RTS technique that can be adopted in practice.*

(3) It is possible to improve precision *of RTS techniques for projects with distributed software histories.*

1.2 Contributions

To confirm the thesis statement, this dissertation makes the following contributions:

- The dissertation presents the first study of RTS in practice. The study shows that most developers manually select to run only a subset of tests, and they select these tests in mostly ad hoc ways. Manual test selection is both unsafe and imprecise: developers select too few tests and thus miss to run some tests whose behavior differs due to code

changes, or developers select too many tests and thus waste time. Additionally, the study shows that existing automated RTS techniques are inefficient as they take substantial time for analysis and collection phases. In sum, a large number of developers would benefit from an automated RTS technique that works in practice.

- The dissertation introduces a novel technique for RTS, named EKSTAZI, which is safe and efficient. EKSTAZI computes for each test what files it depends on. A test need not be run in the new project revision if none of its dependent files changed. EKSTAZI takes a radically different view from the existing RTS techniques: while the existing RTS techniques sacrifice efficiency for precision by keeping fine-grained dependencies (e.g., method), EKSTAZI sacrifice the precision for efficiency by keeping coarse-grained dependencies (i.e., files). In addition to EKSTAZI being safer than the existing techniques, our evaluation on 32 projects, totaling almost 5M LOC, shows that EKSTAZI is efficient: it reduced the end-to-end time 32% on average (and 54% for longer-running test suites) compared to executing all tests. EKSTAZI also has lower end-to-end time than the state-of-the-research RTS technique [158], despite the fact that EKSTAZI selects more tests. EKSTAZI is the first RTS tool adopted by several popular open-source projects, including Apache Camel, Apache Commons Math, and Apache CXF.

- The dissertation presents a novel approach to improve precision of RTS techniques for projects with distributed software histories. All prior RTS techniques reason about changes only between two revisions – an old revision and a new revision – effectively assuming a development process where changes occur in a linear sequence. However, most projects nowadays use distributed version-control systems. Software histories are generally modeled as directed graphs; in addition to changes occurring linearly, multiple revisions can be related by other commands, e.g., branch, merge, rebase, cherry-pick, revert, etc. Unlike any prior RTS technique, our novel approach reasons about commands that create each revision to select tests for a new revision by considering multiple old revisions. We also prove the safety of the approach and present an evaluation on several open-source projects. The results show that the approach can reduce the number of selected tests over an order of magnitude for merge revisions.

The rest of this chapter describes in more detail these three bodies of research.

1.3 Study of Manual RTS

Despite the importance of RTS, there was no prior research that studied *if* and *how* developers actually perform manual RTS, and how manual and automated RTS compare. Our anecdotal experience shows that many developers select to run only some of their tests, but we do not know how many developers do so, how many tests they select, why they select those tests, what automated support they use for manual RTS in their Integrated Development Environment (IDE), etc. Also, it is unknown how developers' manual RTS practices compare with any automated RTS technique proposed in the literature: how does developers' reasoning about affected tests compare to the analysis of a safe and precise automated RTS technique? The potential need for adoptable automated RTS tools makes it critical to study current manual RTS practice and its effects on safety and precision.

This dissertation presents the results of the first study of manual RTS and a first comparison (in terms of safety, precision, and performance) of manual and automated RTS. Specifically, we address the following research questions:

RQ1. How often do developers perform manual RTS?

RQ2. What is the relationship between manual RTS and size of test suites or amount of code changes?

RQ3. What are some common scenarios in which developers perform manual RTS?

RQ4. How do developers commonly perform manual RTS?

RQ5. How good is current IDE support in terms of common scenarios for manual RTS?

RQ6. How does manual RTS compare with automated RTS, in terms of precision, safety, and performance?

To address the first set of questions about manual RTS (RQ1-RQ5), we extensively analyzed logs of IDE interactions recorded from a diverse group of 14 developers (working

on 17 projects, i.e., some developers worked on multiple projects during our study), including several experts from industry [120]. These logs cover a total of 918 hours of development, with 5,757 test sessions and a total of 264,562 executed tests. A *test session* refers to a run of at least one test between two sets of code changes. We refer to test sessions with a single test as *single-test sessions*, and test sessions with more than one test as *multiple-test sessions*. To address RQ6, we compared the safety, precision, and performance of manual and automated RTS for 450 test sessions of one representative project, using the best available, at the time of the study, automated RTS research prototype [158].

Several of our findings are surprising. Regardless of the project properties (open-source vs. closed-source, small vs. large, few tests vs. many tests, etc.), *almost all developers* performed manual RTS. 62% of all test sessions executed a single test, and of the multiple-test sessions, on average, 59% had some test selection. The pervasiveness of manual RTS establishes the need to study manual RTS in more depth and compare it with automated RTS. Moreover, our comparison of manual and automated RTS [158] revealed that manual RTS can be imprecise (in 73% of the test sessions considered, manual RTS selects more tests than automated RTS) and unsafe (in 27% of the test sessions considered, manual RTS selects fewer tests than automated RTS). Finally, our experiments show that current state-of-the-research automated RTS may provide little time savings: the time taken by an automated RTS tool[2], per session, to select tests was 130.94±13.77 sec (Mean±SD) and the (estimated) time saved (by not executing unselected tests) was 219.86±68.88 sec. These results show a strong need for better automated RTS techniques and tools.

1.4 Novel RTS Technique for Two Code Revisions

Lightweight Technique: We propose EKSTAZI (pronounced "Ecstasy")[3], a novel RTS technique based on *file dependencies*. EKSTAZI is motivated by recent advances in build systems [2, 3, 7–9, 50, 70, 115] and prior work on RTS based on class dependencies [67–69,

[2]This measures only the analysis time to identify the affected tests ($\mathcal{A}$) but *not* the collection time ($\mathcal{C}$).

[3]The word "ekstazi" in the Serbian language has the same meaning as the word "ecstasy" in the English language: a feeling or state of intensely beautiful bliss (`http://en.wiktionary.org/wiki/ecstasy`); the name evolved over time: eXtreme Test Selection → eXtreme Test seleCtion → Ecstasy → EKSTAZI.

97, 108, 124, 142] and external resources [54, 88, 117, 154], as discussed further in Chapter 5. Unlike most prior RTS techniques based on finer-grained dependencies (e.g., methods), EKSTAZI does *not* require integration with version-control systems: EKSTAZI does not explicitly compare the old and new code revisions. EKSTAZI computes for each *test entity* (be it a test method or a test class) what files it depends on; the files can be either executable code (e.g., `.class` files in Java) or external resources (e.g., configuration files). A test need not be rerun in the new revision if none of its dependent files changed.

Adoption Approach: We note that testing frameworks, such as JUnit, are widely adopted and well integrated with many popular build systems, such as Ant or Maven. For example, our analysis of 666 most active, Maven-based[4] Java projects from GitHub showed that at least 520 (78%) use JUnit (and 59 more use TestNG, another testing framework). In addition, at least 101 projects (15%) use a code coverage tool, and 2 projects even use a mutation testing tool (PIT [127]). Yet, no project used automated RTS. We believe that integrating a lightweight RTS technique with an existing testing framework would likely increase RTS adoption. Ideally, a project that already uses the testing framework could adopt RTS with just a minimal change to its build script, such as `build.xml` or `pom.xml`.

Implementation: We implement the EKSTAZI technique in a tool integrated with the JUnit testing framework. Our tool handles many features of Java projects/language, such as packing of `.class` files in `.jar` archives, comparison of `.class` files using smart checksums (e.g., ignoring debug information), instrumentation to collect dependencies using class loaders or Java agents, reflection, etc. Our tool can work out-of-the-box on any project that uses JUnit. The EKSTAZI tool is available from `http://www.ekstazi.org`.

Extensive Evaluation: We evaluate EKSTAZI on 615 revisions of 32 Java projects, ranging from 7,389 to 920,208 LOC and from 83 to 641,534 test methods that take from 8 seconds to 2,565 seconds to execute in the base case, called *RetestAll* (that runs all the tests) [110]. To the best of our knowledge, this is the largest evaluation in any RTS study, and the first to report the end-to-end RTS time, including the $\mathcal{C}$ phase. The experiments show that EKSTAZI reduces the end-to-end time 32% on average (54% for longer-running test suites) compared

[4]We cloned 2000 most active Java projects but filtered those that did not use Maven, because our automated analysis considers only pom.xml files.

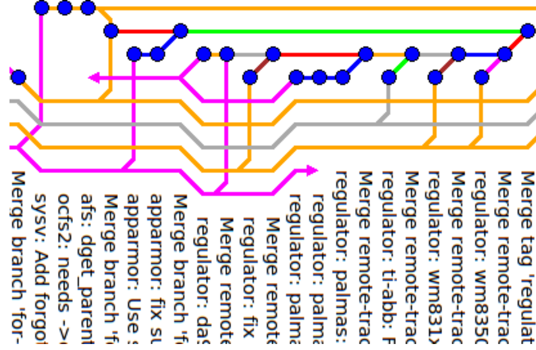


Figure 1.1: Part of software history of the Linux Kernel

to RetestAll. Further, Ekstazi reduces the time 47% on average (66% for longer-running test suites) when the $\mathcal{C}$ phase is performed in a separate, off-line run [48, 63].

We also compare Ekstazi with `FaultTracer` [158], a state-of-the-research RTS tool based on fine-grained dependencies, on a few projects that `FaultTracer` can work on. Not only is Ekstazi faster than `FaultTracer`, but `FaultTracer` is, on average, even slower than RetestAll. We discuss, in Section 3.5, why the main result—that Ekstazi is better than `FaultTracer` in terms of the end-to-end time—is not simply due to `FaultTracer` being a research prototype but a likely general result. In sum, Ekstazi tracks dependencies at our proposed file granularity, whereas `FaultTracer` tracks dependencies at a finer granularity. While Ekstazi does select more tests than `FaultTracer` and has a slightly slower $\mathcal{E}$ phase, Ekstazi has much faster $\mathcal{A}$ and $\mathcal{C}$ phases and thus has a lower end-to-end time.

Ekstazi has already been integrated in the main repositories of several open-source projects where it is used on a regular basis, including in Apache Camel [16], Apache Commons Math [23], and Apache CXF [26].

1.5 Novel RTS Approach for Distributed Software Histories

Previous RTS techniques, which we will call *traditional RTS* techniques, viewed software history as a linear sequence of commits to a centralized version-control system (as was common for CVS or SVN). However, modern software development processes that use distributed version-control systems (DVCSs) do not match this simplistic view. Software version histo-

ries that use DVCSs, such as Git and Mercurial, are complex graphs of branches, merges, and rebases of the code that mirror more complex sharing patterns among developers. For example, Figure 1.1 shows a part of the Linux Kernel Git repository [111]: this software history is a complex graph, with multiple branches being merged. (There is a case in Linux where 30 branches are merged at once.) We empirically find that such complexities are not isolated to the Linux Kernel development: most open-source codebases perform frequent merges. Section 4.4 reports detailed results for a number of open-source projects; we find about third of the commits to be merge-related.

We consider the problem of RTS for codebases that use DVCS commands. One possible *baseline approach* is to apply traditional RTS by picking an arbitrary linearization of the software history. While this technique is safe (recall that a safe technique does not miss tests whose outcome may be affected by the change), we empirically demonstrate that this technique can be very imprecise (recall that an imprecise technique can select many tests whose outcome cannot be affected by the change). Instead, we propose the first approach that explicitly takes into account the history graph of software revisions. We have implemented our approach and show, through an evaluation on several open-source code repositories, that our approach selects on average an order of magnitude fewer tests than the baseline technique, while still retaining safety.

We evaluate our approach both on real open-source code repositories that use DVCS and on distributed repositories that we systematically generate from projects that use a linear sequence of commits. As part of our approach, we propose and compare two *options* for selecting tests at each merge revision of such repositories. These options have different trade-offs in terms of cost (how many traditional RTS analysis, i.e., $\mathcal{A}$, need to be performed to compute the selected tests) and precision (how many tests are selected to be run, while maintaining safety). (Note that the saving in the number of tests reflects in time saving for $\mathcal{E}$ and $\mathcal{C}$ phases.) In particular, we describe a fast option for code merges that does *not* require *any* traditional RTS analysis between two revisions but still achieves a reduction in terms of the number of tests, 10.89× better than a baseline technique that performs one traditional RTS for a merge point. Another option, which performs one traditional RTS analysis for each branch being merged, achieves additional reduction of 2.78× in the number of tests,

but potentially requires many RTS analysis runs. We propose a heuristic to choose one of the options based on the shape of the software history. We also prove safety of the options, assuming that the traditional RTS analysis is safe.

1.6 Dissertation Organization

The rest of this dissertation is organized as follows.

Chapter 2: Manual Regression Test Selection

This chapter presents our study of manual RTS in practice; the results of this study were a part of the motivation for RTS techniques presented in chapters 3 and 4.

Chapter 3: Regression Test Selection with Dynamic File Dependencies

This chapter presents the contributions of the Ekstazi RTS technique for two code revisions, which substantially speeds up end-to-end regression testing time compared to RetestAll technique.

Chapter 4: Regression Test Selection for Distributed Software Histories

This chapter presents the contributions of a novel RTS approach for projects with distributed software histories, which improves efficiency of any RTS technique at many code revisions.

Chapter 5: Related Work

This chapter overviews the various bodies of work that are related to the contributions of this dissertation.

Chapter 6: Conclusion and Future Work

This chapter concludes the dissertation and presents various directions for future work building upon the contributions of this dissertation.

CHAPTER 2

Manual Regression Test Selection

This chapter presents the first study of manual RTS in practice; the results of the study were a part of the motivation for our work presented in chapters 3 and 4. This chapter is organized as follows. Section 2.1 first describes our research methodology and experimental setup, and then discusses how developers actually perform manual RTS. Section 2.2 presents our comparison of manual and automated RTS. Section 2.3 describes potential improvements for manual and automated RTS. Section 2.4 presents threats to validity.

2.1 Evaluating Manual RTS

We first present our methodology for analyzing manual RTS data to answer the research questions RQ1-RQ5 (listed in Section 1.3), and we then summarize our findings.

2.1.1 Methodology

We analyzed the data collected during a previous field study [120], in which the authors of the study unobtrusively monitored developers' IDEs and recorded their programming activities over three months. The collected data has been used in several prior research studies [118, 120, 150] on refactoring and version control; the work presented in this dissertation is the first to focus on the (regression) testing aspects.

To collect data, the study participants were asked to install a record-and-replay tool, `CodingTracker` [51], in their Eclipse [60] (Indigo) IDEs. Throughout the study, `CodingTracker` recorded detailed code evolution data, ranging from individual code edits, start of each test, and test outcome (e.g., pass/fail) up to high-level events like automated refactoring invoca-

Project	Test Sessions			Test [methods]			Selected Tests					Selective
	total	single-test	debug	min	max	*avg*	min	max	*avg*	sum	time^{min}	**Sessions**
$\mathcal{P}_1$	41	20	8	1	7	4.68	1	7	2.59	106	89	28.57%
$\mathcal{P}_2$	218	152	68	1	886	43.70	1	886	9.71	2,116	203	77.27%
$\mathcal{P}_3$	41	28	9	1	530	19.46	1	530	15.61	640	2	38.46%
$\mathcal{P}_4$	94	33	22	170	182	176.23	1	173	103.16	9,697	26	59.02%
$\mathcal{P}_5$	1,231	883	852	1	172	83.00	1	141	13.01	16,019	374	99.71%
$\mathcal{P}_6$	18	7	5	1	13	6.00	1	13	4.11	74	0	18.18%
$\mathcal{P}_7$	55	54	43	1	8	6.47	1	8	1.13	62	34	0.00%
$\mathcal{P}_8$	612	446	306	1	59	34.29	1	44	2.56	1,565	89	92.77%
$\mathcal{P}_9$	443	362	117	1	132	85.86	1	124	5.66	2,508	246	81.48%
$\mathcal{P}_{10}$	178	108	29	1	126	48.54	1	124	14.48	2,577	139	64.29%
$\mathcal{P}_{11}$	129	108	27	1	19	15.29	1	9	1.64	211	53	95.24%
$\mathcal{P}_{12}$	176	121	74	1	121	105.53	1	120	19.39	3,413	153	94.55%
$\mathcal{P}_{13}$	51	36	22	1	18	12.86	1	18	5.53	282	3	0.00%
$\mathcal{P}_{14}$	450	146	103	72	1,012	889.32	1	1,010	113.40	51,031	242	98.36%
$\mathcal{P}_{15}$	156	78	60	1	1,663	13.40	1	1,663	12.98	2,025	9	28.21%
$\mathcal{P}_{16}$	1,666	855	462	1	1,606	1,416.10	1	1,462	103.24	171,990	420	98.40%
$\mathcal{P}_{17}$	198	157	50	1	6	1.83	1	4	1.24	246	23	31.71%
$\sum$	5,757	3,594	2,258	-	-	-	-	-	-	264,562	2,113	-
Ari Mean	338.65	211.41	132.76	-	-	174.27	-	-	-	15,562.47	124.31	59.19%

Table 2.1: Statistics for projects used in the study of manual RTS

tions and test session executions. `CodingTracker` uploaded the collected data to a centralized repository using existing infrastructure [150].

In this study, we only consider data from participants who had more than ten test sessions. Overall, the data encompasses 918 hours of code development activities by 14 developers, of whom five are professional programmers and nine are students. The professional programmers worked in software companies on projects spanning various domains such as marketing, banking, business management, and database management. The students were Computer Science graduate students and senior undergraduate summer interns, who worked on a variety of research projects from six research labs at the University of Illinois. The programming experience of our study participants varied: one developer had less than 5 years, eight developers had between 5–10 years, and five developers had more than 10 years. None of the study participants knew how we would analyze the collected data; in fact, we ourselves did not know all the analyses we would do at the time the data was collected.

In the rest of this section, we discuss the tool used, the projects analyzed, the challenges faced, and the answers we found to the questions RQ1-RQ5.

CodingTracker

`CodingTracker` integrates well with one of the most popular IDEs, Eclipse [99]. Developers do not explicitly interact with `CodingTracker` during their workflow, and thus, the data recorded by `CodingTracker` is as close as possible to what developers normally do. `CodingTracker` collects information about all test sessions. Because test-selection data is available at every test session, we were able to capture developers' manual RTS decisions. Each test session includes a list of executed tests, their execution time, and their status on completion (e.g., pass or fail). Further, `CodingTracker` collects information about *code changes* that happen between test sessions.

Moreover, because test-selection data is available at every test session, we were able to capture developers' manual RTS decisions more realistically and at a finer granularity than one could attempt to infer otherwise, e.g., based only on differences between commits in a version control system (VCS) [13, 57, 58, 79, 153].

While `CodingTracker` logs provide a treasure trove of data, they have limitations. First, `CodingTracker` logs cannot fully confirm that developers performed manual RTS. In theory, developers could have installed some Eclipse plugin that would perform automated RTS for them. However, we are not aware of any automated RTS tool that works in Eclipse. Moreover, we have noticed significant time delays between code changes and the start of test sessions, which likely correspond to developers' *selection times* (i.e., time that developers spend reasoning about which tests to run) and not automated tool runs. Therefore, we assume that developers manually selected the tests in each test session. Second, `CodingTracker` collects information about *code changes but not entire project states*. The original motivation for `CodingTracker` was a study of refactorings [120], which needed only code changes, so a design decision was made for `CodingTracker` to *not* collect the entire project states (to save space/time for storing logs on disk and transferring them to the centralized repository). However, the lack of entire states creates challenges to exactly reconstruct the project as the developer had it for each test session (e.g., to precisely count the number of tests or to compile and run tests for automated RTS). Sections 2.1.1 and 2.2.1 discuss how we address these challenges.

Projects Under Analysis

As mentioned earlier, we analyzed the data from 14 developers working on 17 research and industrial projects, e.g., a Struts web application, a library for natural-language processing, a library for object-relational mapping, and a research prototype for refactoring. Note that some developers worked on several projects in their Eclipse IDE during the three-month study; `CodingTracker` recorded separate data for each project (more precisely, `CodingTracker` tracks each Eclipse workspace) that was imported into Eclipse.

Table 2.1 is a summary of test-related data that we collected[1]. For each project, we first show the number of test sessions. Our analysis showed that a large number of these sessions execute only one test. We refer to such test sessions as *single-test sessions*. Further, we found that many of these single-test sessions execute only one test that had failed in the

[1] Due to the conditions of Institutional Review Board approval, we cannot disclose the true names of these projects.

immediately preceding session. We refer to such sessions as *debug test sessions*. Next, we show the number of *available tests*, i.e., the total number of tests in the project at the time of a test session, discussed in more detail in Section 2.1.1. Then, we show the number of *selected tests*, i.e., a subset of available tests that the developer selected to execute, including the total number of selected tests that the developer executed throughout the study and the total execution time for all test sessions[2]. Finally, we show the percentage of *selective sessions*, i.e., *multiple-test sessions* where the number of selected tests is smaller than the number of available tests. In other words, the developer performed manual RTS in each such test session by selecting to execute only a subset of the tests available in that session; we exclude single-test sessions as they may not be "true" selective sessions - developer knows that not all affected tests are selected.

The total test execution time with manual RTS is substantially lower than it would have been without manual RTS. The sum of the "time^{min}" column in Table 2.1 shows that, when manual RTS is performed, the total test execution time for all developers in our study was 2,113 minutes. In contrast, had the developers always executed all available tests, we estimate[3] that it would have resulted in a total test execution time of 23,806 minutes. In other words, had the developers not performed manual RTS, their test executions would have taken about an order of magnitude more time.

We point out some interesting observations about single-test sessions. First, the projects used in our study span many domains and vary in the number of available and selected tests, but they all have some single-test sessions and some multiple-test sessions. Second, single-test sessions include both debug and non-debug sessions. Non-debug single-test sessions usually happen when introducing a new class/feature, because the developer focuses on the new code. By default, in the rest of the paper, we exclude all single-test sessions from our analyses and only mention them explicitly when some of the subsequent plots or other numbers that we report include single-test sessions.

[2]The reported execution time is extracted from the timestamps recorded on developers' computers. It is likely that developers used machines with different configurations, but we do not have such information.

[3]Note that `CodingTracker` does/can not record the execution time for the unselected tests that were not executed; we estimate the time from the averages of the sessions in which the tests were executed.

```
1  // Inputs: Session info extracted from CodingTracker logs
2  List⟨TestSession⟩ sessions;
3  Map⟨TestSession, Set⟨Pair⟨ClassName, MethodName⟩⟩⟩ executed;
4
5  // Output: Available tests for each test session
6  Map⟨TestSession, Set⟨Pair⟨ClassName, MethodName⟩⟩⟩ available;
7
8  // Compute available tests for each test session
9  ComputeAvailable()
10    Set⟨Pair⟨ClassName, MethodName⟩⟩ 𝔸 = {} // Current available tests
11    available = {}
12
13    foreach s: sessions
14      Set⟨Pair⟨ClassName, MethodName⟩⟩ e = executed(s)
15      if |e| > 1
16        𝔸 = 𝔸 ∖ {(c, m) ∈ 𝔸 | ∃(c, m′) ∈ e}
17      𝔸 = 𝔸 ∪ e
18      available(s) = 𝔸
```

Figure 2.1: Algorithm for computing a set of available test methods at each test session

Challenges

`CodingTracker` was initially designed to study how code evolves over time [120], and thus it recorded only code changes and various file activities but not the entire state of the developers' projects. As a consequence, we could not easily extract the number of available tests for each test session: while `CodingTracker` did record the information about tests that are *executed*/selected, it had no explicit information about tests that were *not* executed. Therefore, we developed an algorithm to estimate the number of available tests (reported in Table 2.1). We designed our algorithm to be conservative and likely *under-estimate* the number of available tests. In other words, developers likely performed even more manual RTS than we report.

Figure 2.1 shows the algorithm. The input is a list of test sessions extracted from the `CodingTracker` logs; each session is mapped to a set of executed tests, and each test is represented as a pair of a test class and test method name. The output is a mapping from test sessions to the set of available tests. Although we extract more information for each test session, e.g., execution time, that information is not relevant for this algorithm.

The algorithm keeps track of the current set of available tests, $\mathbb{A}$, initialized to the empty set (line 10). For each test session, the algorithm adds to $\mathbb{A}$ the tests executed in that session

(line 17); those tests are definitely available. The algorithm also attempts to find which tests may have been removed and are not available any more. For each multiple-test session, the algorithm removes from $\mathbb{A}$ all the tests whose class matches one of the tests executed in the current session s (line 16). The assumption is that executing one test from some class c (in a session that has more than one test) likely means that *all* tests from that class are executed in the session. Thus, any test from the same class that was executed previously but not in the current session was likely removed from the project. This assumption is supported by the fact that Eclipse provides rather limited support for selection of multiple tests from the same class as discussed in Section 2.1.2. For single-test sessions, the algorithm only adds the executed test to $\mathbb{A}$; the assumption is that the same tests remain available as in the previous session, but the developer decided to run only one of the tests. Finally, $\mathbb{A}$ becomes the available set of tests for the current session (line 18). Note that our algorithm does not account for removed test classes, but these are very rare in our data set. For example, we inspected in detail project $\mathcal{P}_{14}$, one of the largest projects, and no test class was deleted.

2.1.2 Investigating Manual RTS

In sum, the results showed that almost all developers in our study performed some manual RTS. They did so regardless of the size of their test suites and projects, showing that manual RTS is widely practiced. Next, we provide details of our findings regarding research questions RQ1-RQ5.

RQ1: How often do developers perform manual RTS?

Developers performed manual RTS in 59.19±35.16% (mean ± SD) of the test sessions we studied (column "Selective Sessions" in Table 2.1). Note that we first compute selective session ratio for each developer, and then we took an *unweighted* arithmetic mean of those ratios (rather than weighting by the number of test sessions), because we do not want developers with the most test sessions to bias the results.

Across all 2,163 multiple-test sessions in our study, the average ratio of selected tests (tests that the developer executed) to available tests (tests that could have been executed),

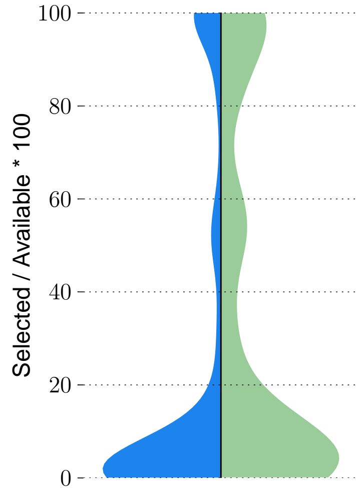


Figure 2.2: Distribution of test selection ratio with and without single-test sessions

i.e., average *test selection ratio*, was only 35.07%. Note that this number is calculated from all test sessions as if they were obtained from a single developer. We show the distribution of test selection ratios for all test sessions for all the developers using violin plots [95] in Figure 2.2. A violin plot is similar to a boxplot but additionally shows probability density of the data at different values. The left part of Figure 2.2 shows the distribution of test selection ratios when single-test sessions are included, while the right part shows the distribution when single-test sessions are excluded. We show only one half of each violin plot; the missing halves are symmetric. It can be observed from the violin plots that manual RTS happens very frequently, and, most of the time, the test selection ratio is less than 20%.

We note here that our finding constitutes the first empirical evidence concerning manual RTS in practice. More importantly, we think that this fact should result in a call-to-arms by the automated RTS community, because poor manual RTS could be hampering developer productivity and impacting negatively on software quality.

RQ2: Does manual RTS depend on size of test suites or amount of code changes?

Developers performed manual RTS regardless of the size of their test suites. We draw this conclusion because almost all developers in our study performed manual RTS, and they had

a wide range of test-suite sizes. The average test-suite size in all 17 projects we studied was 174.27 tests (column "Test [methods]" in Table 2.1); the minimum was 6 tests, and the maximum was 1,663 tests. Considering that these projects are of small to medium size, and because they exhibit manual RTS, we expect that developers of larger projects would perform even more manual RTS.

We also consider the relationship between the size of recent code changes and the number of tests that developers select in each test session. One may expect that developers run more tests after large code changes. We correlate the test selection ratio with the *code change ratio* for all test sessions. The code change ratio is calculated as the percentage of AST node changes [120] since the previous test session over the total AST node changes during the entire study for a particular project. To assess correlation, we measure the Spearman's and Pearson's correlation coefficients[4]. The Spearman's and Pearson's coefficients are 0.28 (0.25 when single-test sessions are included) and 0.16 (0.16 when single-test sessions are included), respectively. In all cases, the p-value was below 0.01[5], which confirms that some correlation exists. However, the low values of coefficients imply a low correlation between the amount of code changes immediately before a test session and the number of manually selected tests in that session. This low correlation was a surprising finding as we had expected a higher correlation between code changes and the number of selected tests.

RQ3: What are common scenarios for manual RTS?

The most common scenario in which developers performed manual RTS was while debugging a single test that failed in the previous session. Recall that we refer to such test sessions as *debug test sessions*. As seen in Table 2.1 (column "debug"), debug test sessions account for 2,258 out of the 5,757 total test sessions considered. One common pattern that we found in the data was that, after one or more tests fail, developers usually start making code changes to fix those failing tests and keep rerunning only those failing tests until they pass. After all the failing tests pass, the developers then run most or all of the available tests to check

[4]Although the data is not normally distributed, and the relationship is not linear, we report the Pearson's coefficient for completeness.

[5]A low p-value indicates that Spearman's or Pearson's coefficient is unlikely 0.

for regressions. Another pattern is when a developer fixes tests one after another, rerunning only a single failing test until it passes. Therefore, even if the developers had a "perfect" automated RTS tool to run after each change, such a tool could prove distracting when running many debug test sessions in sequence. Specifically, even if some code changes affect a larger number of tests, developers may prefer to temporarily run only the single test that they are currently debugging. The existence of other reasons for RTS, besides efficiency improvements, shows a need for a different class of tools and techniques that can meet these actual developer needs; we discuss this further in Section 2.3.

It is also interesting to mention that the *sequences of single-test sessions* (i.e., single-test sessions without other test sessions in between) were much longer than we expected. The mean±SD of the length of single-test session sequences was 6.83±37.00. The longest single-test session sequence contains 99 test sessions, which may indicate that developers avoid running all tests when focusing on new features and debugging.

RQ4: How do developers commonly perform manual RTS?

We found that developers use a number of ad hoc ways for manual RTS. These include: (1) commenting out tests that should not be run, (2) selecting individual *nodes of hierarchy*, by which we refer to the way tests are hierarchically organized in a Java IDE, from test methods to test classes to test packages to entire projects, and (3) creating test scripts, which specify runs of several nodes of hierarchy.

Manual RTS by Commenting: One approach used by the developers was to comment out unit tests they did not want to run. We observed that developers performed this type of selection at different levels of granularity. Some developers commented out individual test methods within a test class, while others commented out entire test classes from JUnit annotations that specify test suites. In both cases, the time overhead incurred by the developer in deciding which tests to run and in commenting out the tests, i.e., *selection time*, is likely to be non-negligible. In other words, selection time is an estimate of the time spent by developers to manually "analyze" and select which tests may be affected. Using the available `CodingTracker` data, we estimate selection time to be the time elapsed from the last

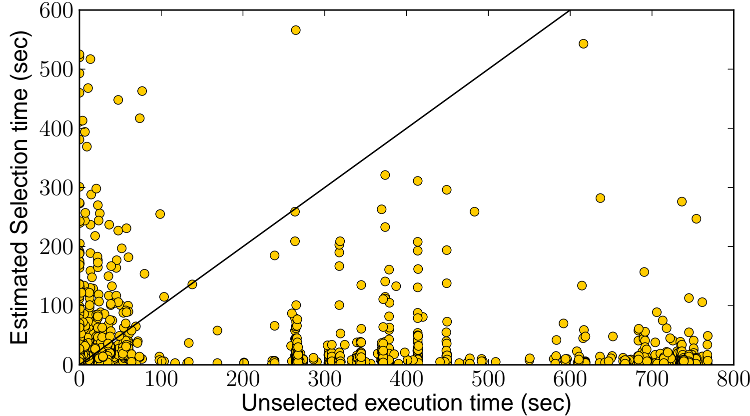


Figure 2.3: Relationship between time to execute unselected tests and selection time

code change that immediately preceded a test session and the start of the test session. We exclude selection time values greater than 10 minutes, as developers may rerun tests after taking a break from work. Our experiments with break times of 5 minutes and 20 minutes did not significantly change any of the outcomes of our study. In Figure 2.3, we show the correlation between selection time and (estimated) time to execute unselected tests (which is the time saved by not executing unselected tests). While the overall time savings due to manual RTS is significant, we found that in 31% of the cases (points above the identity line in Figure 2.3) developers could have saved more time by simply running all the tests.

Manual RTS by Selecting Various Nodes of Hierarchy: Developers also perform test selection by selecting a node of hierarchy in their IDE, e.g., they could select to run only a single test or all the tests from a single class or package. This is a critical RTS limitation in Eclipse—it restricts the developer to select to run only one node of hierarchy (in the limit this node represents the entire project such that the entire test suite for that project is run). In other words, the developer is not able to select to run an *arbitrary* set of tests or test suites. Related but different, in several projects, by browsing through the changes collected by `CodingTracker`, we noticed that developers were writing scripts (".launch" files in Eclipse) to group tests. Using a script has the same limitation as manually selecting a node of hierarchy. These limitations of Eclipse are shared by several popular IDEs as shown in Table 2.2 [98, 121, 151].

RQ5: How good is IDE support for manual RTS?

IDEs provide varying levels of support for performing manual RTS. The IDEs we investigated are: Eclipse[6], IntelliJ IDEA[7], NetBeans[8], and Visual Studio 2010[9].

Support for Arbitrary Manual RTS: Recall from the answer to RQ4 that, in several cases, the developers selected among tests by commenting out the tests within test classes or commenting out test classes within test suites. This likely means that developers would prefer to arbitrarily select tests within nodes of hierarchy. Also, our experience with running the automated RTS tool (as discussed in Section 2.2) shows that all affected tests may not reside in the same node of hierarchy. Thus, it is also important to be able to arbitrarily select tests across these nodes.

Table 2.2 is a summary of available IDE support for selecting tests at different levels of granularity within and across nodes of hierarchy. All the IDEs allow developers to select to run a single test. Moreover, several IDEs offer support for arbitrary selection. IntelliJ allows to arbitrarily select tests by marking (in the GUI) each test to be run subsequently. This may be tedious for selecting among very many tests and is only available for arbitrarily selecting test classes across test packages or test methods within the same class. Visual Studio allows arbitrary selection by specifying regular expressions for test names which may match across multiple nodes of hierarchy. However, not all developers are familiar with regular expressions, and knowledge of all test names in the project is required to write them effectively. Still, based on our study, having this type of support seems very valuable, given that it is needed by the developers. More importantly, Eclipse lacks support for such arbitrary test selection.

Support for RTS across multiple test sessions: We showed in the answer to RQ3 that the most common pattern of manual RTS occurred during debug test sessions. It is likely that the changes made between debug test sessions affect more tests than the test being fixed.

[6]Kepler Service Release 1, build id: 20130919-0819.

[7]Version 12.1.6, build id: IC-129.1359.

[8]Version 7.4, build id: 201310111528.

[9]We selected Visual Studio 2010 rather than the latest version because Visual Studio 2010 was the only IDE that has ever supported automated RTS; interestingly enough, this automated RTS support has been removed from the IDE in subsequent releases.

RTS Capability	Eclipse	NetBeans	IntelliJ	VS 2010
Select single test	+	+	+	+
Run all available tests	+	+	+	+
Arbitrary selection in a node of hierarchy	-	-	±	+
Arbitrary selection across nodes of hierarchy	-	-	±	+
Rerun only previously failing tests	+	+	+	+
Select one from many failing tests	-	-	+	+
Arbitrary selection among failing tests	-	-	+	+

Table 2.2: RTS capabilities of popular IDEs

Indeed, we found this to be the case for project $\mathcal{P}_{14}$. It is possible that the developers do not select other tests affected by the changes due to additional reasoning required to identify such tests. Thus, their test selections during debug test sessions are likely to be unsafe and may lead to extra debug steps at a latter stage. Although Visual Studio provides some level of RTS automation, it has some shortcomings that we discuss in Section 2.3.

One observation from our comparison of IDEs is that they differ in their level of support for the different patterns of manual RTS, but even if we combined the best RTS features from all IDEs investigated, it would still not be sufficient for safe and precise RTS that developers need.

2.2 Manual vs. Automated RTS

We next discuss the results of our comparison of manual and automated RTS, by which we address question RQ6. We compare both approaches in terms of safety, precision, and performance using one of the largest project from our study. As no industry-strength tool for automated RTS was available, we used `FaultTracer` [158], a recently developed state-of-the-research RTS prototype.

2.2.1 Methodology

We investigated in detail the data collected from one of our study participants, with the goal of comparing manual and automated RTS. We chose $\mathcal{P}_{14}$ from Table 2.1 (for reasons

described later in this section). First, we reconstructed the state of $\mathcal{P}_{14}$ at every test session. Recall that `CodingTracker` does *not* capture the entire state of the project for any test session. We had to perform a substantial amount of work to find a code revision that (likely) matched the point where the developer used `CodingTracker`. We acknowledge the help of the $\mathcal{P}_{14}$ developer who helped with this information, especially that the code moved from an internal repository to an external repository. It took several email exchanges to identify the potential revision on top of which we could replay the `CodingTracker` changes while still being able to compile the project and execute the tests. Second, for each test session, we ran `FaultTracer` [158] on the project and compared the tests selected by the tool with the tests selected by the developer. Because `FaultTracer` is a research prototype, it did not support projects (in the general sense of the term "software projects") that are distributed across multiple Eclipse projects (in the specific terminology of what Eclipse calls "projects") even in the same Eclipse workspace. We worked around this limitation by automatically merging all Eclipse projects from $\mathcal{P}_{14}$ into one project that `FaultTracer` could analyze.

Upon replaying the `CodingTracker` logs and analyzing the data, we discovered that the developer often ran multiple test sessions which had no code changes between them. The developer had organized the tests in separate test suites and always selected to run these test suites one at a time, thereby potentially running multiple test sessions in parallel.

To compare manual and automated RTS fairly and consistently, we accounted for the occurrence of multiple test sessions without intervening changes. This is because `FaultTracer` would only select to run tests after detecting code changes between consecutive revisions of the software. Our solution was to merge consecutive test sessions which had no intervening changes. Consider two consecutive test sessions, X and Y, with no intervening changes. Suppose that the tests and their outcomes for X are `[test1:OK, test4:OK]`, and for Y are `[test1:OK, test2:Failure, test3:OK]`. Our merge would produce a union of the tests in X and Y, and if a test happens to have different outcome, the merge would keep the result from X; however, because the test runs happened without intervening changes, it is reasonable to expect that if some tests are rerun, their outcomes should be the same. We checked that, in our entire study, the test runs are largely deterministic and found a tiny percentage of non-deterministic tests (0.6%). The effect of non-deterministic tests

on RTS is a worthwhile research topic on its own [112]. For the sessions X and Y shown above, the merged session would contain the tests `[test1:OK, test2:Failure, test3:OK, test4:OK]`.

Having merged the manual test sessions as described above, the number of test sessions for comparing manual and automated RTS we obtained was 683. We further limited our comparison to the first 450 of these 683 test sessions, due to difficulties in automating the setup of $\mathcal{P}_{14}$ to use `FaultTracer` to perform RTS between successive revisions. As we studied a very large project, which evolved very quickly and had dependencies on environment and many third-party libraries, we could not easily automate the setup across all 683 merged test sessions. The 450 test sessions used constitute the largest consecutive sequence of test sessions which had the same setup. (We discuss other challenges faced in Section 2.2.1.) Across all 450 test sessions considered, $\mathcal{P}_{14}$ has, on average, 83,980 lines of code and 889.32 available tests.

FaultTracer

The inputs to `FaultTracer` are two program revisions (that include test suites)—old revision P and new revision P'—and the execution dependencies of tests at revision P (i.e., a mapping from test to nodes of extended control-flow graph [158] covered by the test). Let $\mathbb{A}$ be the set of tests in P. `FaultTracer` produces, as output, a set of tests $\mathbb{S}_{sel} \subseteq \mathbb{A}$ that are affected by the code changes between P and P'. The unselected tests in $\mathbb{A} \setminus \mathbb{S}_{sel}$ cannot change their behavior. Note that one also has to run new tests that are added in P'.

We chose `FaultTracer` because it represents the state-of-the-research in RTS and implements a mostly safe RTS technique. Also, `FaultTracer` works at a fine-granularity level (which improves its precision), because it tracks dependencies at the level of an extended control-flow graph [158]. To identify code changes, `FaultTracer` implements an enhanced change-impact analysis. In addition, `FaultTracer` targets projects written in Java, the same programming language used in $\mathcal{P}_{14}$, so, there was a natural fit.

However, note that we chose `FaultTracer` from a very limited pool. To the best of our knowledge, there exists no other publicly available tool that performs RTS at such fine gran-

ularity level (e.g., statement, control-flow edge, basic block, etc.). Systems such as Google's TAP system [146, 149] and Microsoft's Echelon system [92, 93, 144] are proprietary. Moreover, TAP implements a coarse-grained analysis technique based on dependencies between modules, which would be overly imprecise for $\mathcal{P}_{14}$ that has only few modules and even the largest project in the `CodingTracker` study consists of only a few modules. On the other hand, Echelon only prioritizes tests but does not select tests for execution.

Project under Analysis

We chose $\mathcal{P}_{14}$ for the following major reasons. First, it was one of the projects with the largest recorded data (in terms of the number of test sessions) of all 17. Hence there was a higher chance of observing a greater variety of test selection patterns. This also means that we had more data points over which to compare manual and automated RTS for the same developer. Second, the developer worked on creating a large and industrially used library, presenting the opportunity to study test selection in a realistic setting. Finally, with the help of the original developer of the project, we were able to gain access to the exact VCS commits of the project which matched the recorded data. At the time of this writing, developers of other projects have either been unable to provide us access to their repositories, or we are unable to reconstruct the revisions of their projects that matched the exact period in the `CodingTracker` recording.

Challenges

Because `CodingTracker` did not capture entire project state, we had to reconstruct the $\mathcal{P}_{14}$'s developer's workspace to be able to build and run tests for our analysis. Using timestamps from the `CodingTracker` logs, we looked for a commit in the developer's VCS which satisfied the following conditions: (1) the time of the commit matches the VCS commit timestamp recorded in the `CodingTracker` logs and (2) the code compiles after checking it out of the VCS and adding required dependencies. Finally, this checked-out revision was imported into Eclipse [60] and used as a basis for replaying the `CodingTracker` logs. By replaying the changes captured by `CodingTracker` on top of this initial state, we obtained the state

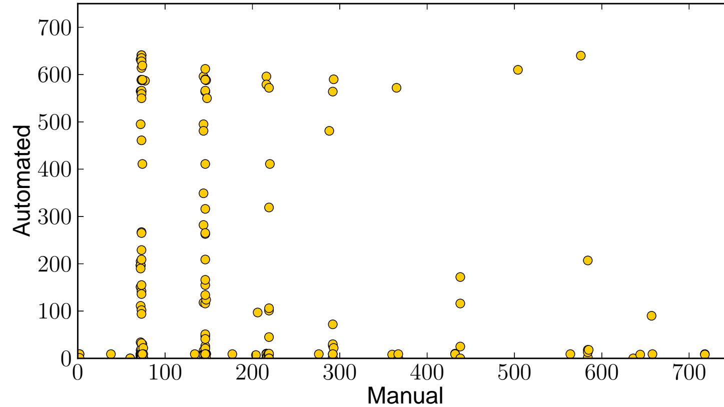


Figure 2.4: The number of tests selected by manual and automated RTS for $\mathcal{P}_{14}$

of the entire project in every succeeding test session. Note that `CodingTracker` captures changes to both the project under test and the testing code, and thus, the reconstructed developer's workspace contained all the tests available at any given test session. We assume that the ability to replay the `CodingTracker` logs from the initial VCS commit till the end of the logs without any error means that it was a likely valid starting point. Thus, the reconstructed workspace is as close to the developer's workspace as it existed while `CodingTracker` monitored the developer's programming activity.

To mitigate these challenges in future studies focusing on RTS, `CodingTracker` would need to be modified to capture the complete initial state of the project as well as any dependencies on external libraries.

2.2.2 Comparing Manual and Automated RTS

The number of selected tests: We plot, in Figure 2.4, the number of tests selected by manual RTS against the number of tests selected by automated RTS (i.e., `FaultTracer`) for each test session. A quick look may reveal that there is a substantial difference between manual and automated RTS, which we further analyze.

Figure 2.5 shows the distribution, across test sessions, of the number of tests selected by manual and automated RTS. We show the distribution for two cases: with ("w/") and without ("w/o") single-test sessions. It can be seen that the median is much lower for the automated tool in both cases. This implies that the developer is imprecise (i.e., selects

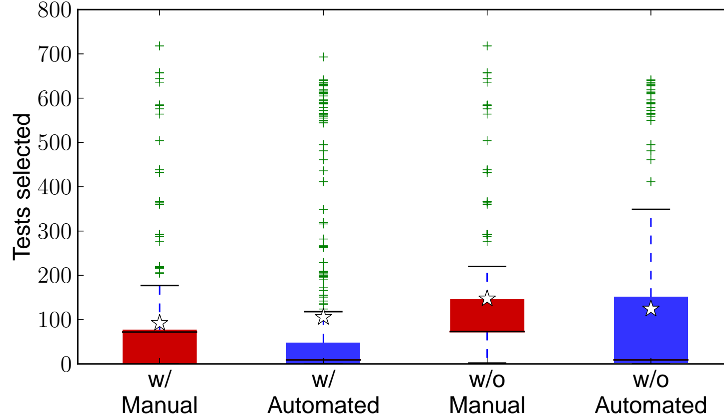


Figure 2.5: Distribution of selected tests for $\mathcal{P}_{14}$ with and without single-test sessions

more than necessary). Further, if single-test sessions are included, we can observe that the arithmetic mean (shown as a star) is lower for manual than automated RTS. However, when single-test sessions are excluded, we can see the opposite. This indicates, as expected, that developer focuses on very few tests while debugging and ignores the other affected tests. Finally, when single-test sessions are excluded from the manually selected tests, we found that many test sessions contain the number of tests equal to the median. Our closer inspection shows this to be due to the lack of support for arbitrary selection in Eclipse, which forced the developer to run all tests from one test class in a node of hierarchy.

Safety and precision: One major consideration in comparing manual and automated RTS is the safety of these approaches relative to one another. In other words, if we assume that the automated tool always selects all the tests affected by a code change, does the developer always select a superset of these? If the answer is in the affirmative, then the developer is practicing safe RTS. On the contrary, if the set of tests selected by the developer does not include all the tests selected by the tool, it means that manual RTS is unsafe (or the tool is imprecise). To compare safety between manual and automated RTS, for every test session, we compare both the number of tests selected and the relationship between the sets of tests selected using both approaches.

Figure 2.4 shows the relationship between the numbers of tests selected by both approaches. The Spearman's and Pearson's correlation coefficients are 0.18 (p-value below 0.01) and 0.00 (p-value is 0.98), respectively. These values indicate a rather low, almost

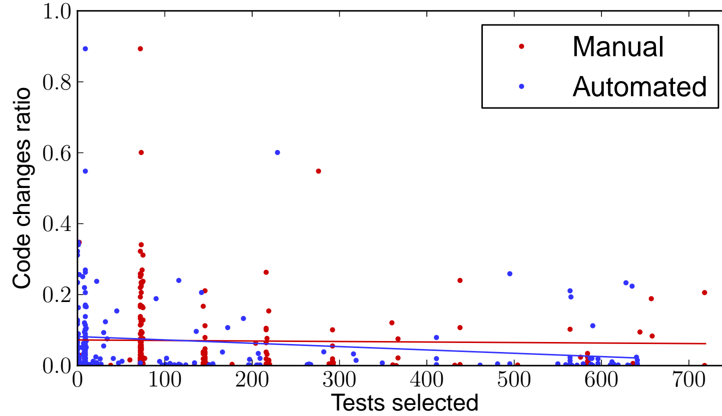


Figure 2.6: Relationship of RTS with relative size of code changes for $\mathcal{P}_{14}$

non-existent, correlation.

We compared the relation between the sets of tests selected using manual and automated RTS. In 74% of the test sessions, the developer missed to select at least one of the tests selected by `FaultTracer`. Assuming that `FaultTracer` is safe, we consider these cases to be unsafe. In the remaining 26% of the test sessions, the developer selected a superset of tests selected by `FaultTracer`. Moreover, in 73% of the test sessions, the developer selected more tests than `FaultTracer`. Assuming that `FaultTracer` is precise, we consider these cases to be imprecise. Note that a developer can be both unsafe and imprecise in the same test session if the developer selects some non-affected tests and does not select at least one of the affected tests. Thus, the sum of the percentages reported here (74% + 73%) is greater than 100%.

Correlation with code changes: In Section 2.1.2, we found that for *all* projects in our study there is low correlation between code change ratio and manual RTS. We revisit that correlation in more detail for the $\mathcal{P}_{14}$ project. To further compare manual and automated RTS, we evaluate whether either of these selection approaches correlates better with code changes. Effectively, we re-check our intuition that the developer is more likely to select fewer tests after smaller code changes. We measured the Pearson's and Spearman's correlation coefficients for both manual and automated RTS. The values for Spearman's coefficients are 0.22 (p-value below 0.01) and 0.01 (p-value is 0.93) for manual and automated RTS, respectively. The values for Pearson's coefficients are 0.08 (p-value is 0.10) and -0.02 (p-value is 0.77) for manual and automated RTS, respectively. While the correlation is low in

all cases, the slightly higher values of correlation coefficients for manual RTS may indicate that (compared to automated RTS) the developer indeed selects fewer tests after smaller changes and more tests after larger changes, as it becomes harder to reason which tests are affected by larger changes. The plot in Figure 2.6 visualizes the relationship, for each test session, between code change ratio and the number of selected tests for both manual and automated RTS. We can observe that manual RTS is less likely to select many tests for small changes (e.g., fewer red dots than blue dots are close to the x-axis around the 600 mark). In the end, the size of semantic effect of a change (as measured by the number of affected tests) is not easy to predict from the size of the syntactic change (as measured by the number of AST nodes changed).

Performance: We finally compare manual and automated RTS based on the time taken to select the tests (i.e., $\mathcal{A}$ phase). Figure 2.7 shows the distribution of selection time (first boxplot), as defined in Section 2.1.1, and analysis time (second boxplot) incurred by `FaultTracer`. We can observe that the developer is faster than the automated RTS tool in selecting which tests to run (the p-value for the Mann-Whitney U test is below 0.01). For comparison, we also show the distribution of estimated execution time for tests that are unselected by `FaultTracer` (third boxplot) and actual execution time for tests selected by `FaultTracer` (fourth boxplot). We ran all our experiments on a 3.40 GHz Intel Xeon E3-1240 V2 machine with 16GB of RAM, running Ubuntu Linux 12.04.4 LTS and Oracle Java 64-Bit Server version 1.6.0_45.

One can observe that `FaultTracer` analysis ($\mathcal{A}$) took substantial time. Although the analysis time (130.94±13.77 seconds) is, on average, less than the time saved by not running unselected tests (219.86±68.88 seconds), it is important to note that one may also want to take into account time to collect necessary dependency information to enable change impact analysis ($\mathcal{C}$); if time taken for analysis plus overhead for collecting dependencies plus running selected tests is longer than time taken for running all the tests, then test selection provides no benefit. This raises the question whether a fine-grained technique, such as the one implemented in `FaultTracer` [158], can be optimized to bring benefits to smaller projects. We believe that research studies on automated RTS should provide more information about their complexity (e.g., time to implement the technique) and efficiency (e.g., analysis time,

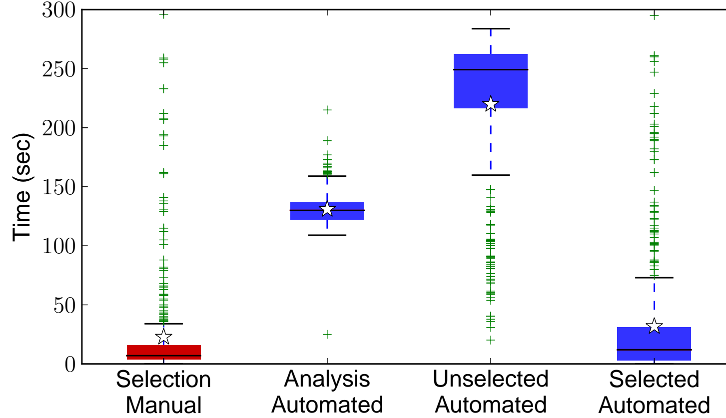


Figure 2.7: Distribution for $\mathcal{P}_{14}$ of selection, analysis, and execution time

collection time, etc.). Previous research focused mostly on the number of selected tests (i.e., safety and precision) rather than on end-to-end time, which is not sufficient for proper comparison and discovering an RTS technique that works in practice.

2.3 Discussion

We briefly discuss test-selection granularity, our experience with an IDE-integrated automated RTS tool, and propose a potential improvement to automated RTS in IDEs.

Test Selection Granularity: We mentioned earlier that systems such as Google TAP [146, 149] and Microsoft Echelon [92, 93, 144] are successfully used for test selection/prioritization. However, these systems are used as part of the gated check-in [72] infrastructure (i.e., all affected regression tests are executed before a commit is accepted into the repository). In other words, they are not used (and are not applicable) on developers' machines where developers commonly work on few modules at a time (and run tests locally). Even developers at either of these companies, let alone many developers who do not develop code at the scale of Google or Microsoft, would benefit from an improved fine-grained test selection. This provides motivation for research on finding the best balance between analysis time, implementation complexity, and benefits obtained from test selection. Improved fine-grained test selection would be more widely

applicable and could be used in addition to coarse-grained test selection systems.

Experience with IDE-integrated automated RTS: We experimented with Visual Studio 2010, the only tool (to the best of our knowledge) that integrates automated RTS with an IDE. We did this to see if such a tool would perform better than manual RTS in terms of safety and precision. Specifically, the Test Impact Analysis (TIA) tool in Visual Studio 2010 [145] was designed to help reduce testing effort by focusing on tests that are likely affected by code changes made since the previous run of the tests. We think this is an excellent step towards improved RTS in developer environments and that similar tools should be developed for other IDEs. We successfully installed TIA and ran it on several simple examples we wrote and on an actual open-source project. However, we found a number of shortcomings with TIA. Most importantly, the tool is *unsafe*: any change not related to a method body is ignored (e.g., field values, annotations, etc.). Also, changes like adding a method, removing a method, or overriding a method remain undetected [129]. Furthermore, TIA does not address any of the issues commonly faced by selection techniques [37,44,55,67,89,91,138,154,156], such as library updates, reflection, external resources, etc. Our opinion is that a safe but imprecise tool would be more appreciated by developers.

Potential improvement of IDEs: Across all projects, we observed that developers commonly select tests during debugging. Thus, one common way by which an IDE might help is to offer two separate modes of running tests, a *regular mode* (without selection) and a *test-selection mode*. In the regular mode, the developer may choose to rerun, after a series of code changes, one or more previously failing tests (while ignoring other affected tests). Once the test passes, the developer may run in the test-selection mode to check for regressions. Notice that the test-selection runs would be separated by a series of regular runs. Consider two test-selection runs, A and B (Figure 2.8). In A, some tests were selected to be run and failed. Developer then performs (regular) runs a_1, a_2, ... a_n, until the previously failing test passes. The test selection run B is then executed to ensure that there are no regressions due to code changes, since A. Note that the analysis performed before running B should consider the difference since A

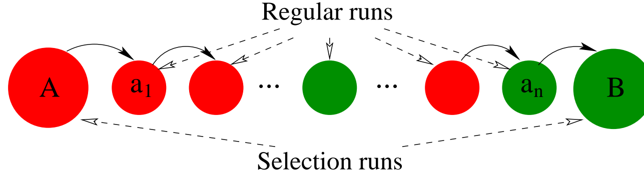


Figure 2.8: Example of a pattern when developer alternates selection and regular runs

and not just the difference between a_n and B; otherwise, tests affected by the changes between A and a_n would not be accounted for. As a simple optimization step, the tool could exclude the tests affected between A and B that were already run after the change that was affecting them.

2.4 Threats to Validity

External: Developers, Projects, and Tools: The results of our study may not generalize to projects outside of the scope of our study. To mitigate this threat, we used 17 projects that cover various domains and 14 developers with different levels of programming experience. Further, these projects vary significantly in size, number of developers, and number of tests. Regarding the comparison of manual and automated RTS, we used the largest project for which we could reconstruct the entire state for many test sessions.

We used `FaultTracer`, a research prototype, to perform automated RTS. Other tools [37, 44, 55, 67, 89, 91, 138, 154, 156] that implement different RTS techniques could have led to different results. We chose `FaultTracer` because it implements a mostly safe and precise RTS technique (with respect to the analyzed changes in $\mathcal{P}_{14}$). To the best of our knowledge, no other publicly available tool for RTS exists (except the proprietary tools that work at coarse-granularity level, which would not be applicable to any of the projects used in our study). Our experience with Visual Studio demonstrated that the implemented approach is unsafe, thus inappropriate for our study.

Finally, the patterns of test selection could differ in other languages. We leave the

investigation of how manual RTS is performed in other languages for future work.

Internal: Implementation Correctness: We extracted data relevant to manual RTS from the study participants' recordings. To extract the data, we wrote analyzers on top of the infrastructure that was used in prior research studies on refactorings [118, 120, 150]. Further, new analyzers were tested and reviewed by at least two authors of our ASE 2014 paper [78].

Construct: IDEs and Metrics: Because `CodingTracker` is implemented as an Eclipse plugin, all developers in our study used Eclipse IDE. Therefore, our study results may not hold for other IDEs. However, because Eclipse is the most popular IDE for Java [99], our results hold for a significant portion of Java developers. We leave the replication of our study using other popular IDEs (both for Java and other languages) for future work.

2.5 Summary

This chapter motivated the need for an automated RTS technique. The analysis of logs obtained in real time from a diverse group of developers showed that almost all developers practice manual RTS, but they select tests in mostly ad hoc ways. As a result, manual RTS is unsafe and imprecise: developers select too few tests and thus miss to run some tests whose behavior differs due to code changes, or developers select too many tests and thus waste time. A large number of developers would benefit from an efficient automated RTS technique.

CHAPTER 3

Regression Test Selection with Dynamic File Dependencies

This chapter presents the EKSTAZI technique that was developed with the aim to enable safe and efficient regression test selection between *two code revisions*. The lack of a practical RTS technique (after three decades of research) and our study of manual RTS (presented in Chapter 2) were the main motivation points for the work presented here. This chapter is organized as follows. Section 3.1 introduces the key terms and illustrates several RTS techniques. Section 3.2 describes our RTS technique that tracks dynamic file dependencies. Section 3.3 presents our extensive evaluation. Section 3.4 describes several common patterns to integrate our tool in projects that could benefit from RTS. Section 3.5 discusses surprising results and handling of various tests. Section 3.6 presents threats to validity.

3.1 Example

We use a synthetic example to introduce the key terms and illustrate several RTS techniques and their trade-offs. Figure 3.1 shows sample code that represents an old revision of a project: two test classes—`TestM` and `TestP`[1]—contain four test methods—`t1`, `t2`, `t3`, and `t4`—for two classes under test—`C` and `D`.

Executing the tests on this revision can obtain a *dependency matrix* that relates each *test entity* to a set of *dependent elements*. We refer to the granularity of test entities as *selection granularity*—this is the level at which tests are tracked and selected (as test methods or test classes), and we refer to the granularity of dependent elements as *coverage granularity*—this is the level at which changes are determined. The dependent elements can be of various

[1]Test classes more commonly match the classes (rather than methods) under test, but this example allows to succinctly present our main points.

```
class TestM {
  void t1() { assert new C().m() == 1; }
  void t2() { assert new D().m() == 1; }
}

class TestP {
  void t3() { assert new C().p() == 0; }
  void t4() { assert new D().p() == 4; }
}
```

```
class C {
  int m() { /* no method calls */ }
  int p() { /* no method calls */ }
}

class D extends C {
  @Override
  int p() { /* no method calls */ }
}
```

Figure 3.1: Example test code (left) and code under test (right)

```
t1: C#C, C#m
t2: D#D, C#C, C#m
t3: C#C, C#p
t4: D#D, C#C, D#p
```

(a) method-method

```
t1: TestM, C
t2: TestM, D, C
t3: TestP, C
t4: TestP, D, C
```

(b) method-class

```
TestM: TestM, C, D
TestP: TestP, C, D
```

(c) class-class

Figure 3.2: Dependency matrices collected for code in Figure 3.1

granularity; for our example, we use methods and classes (but even finer elements can be used, e.g., basic blocks in Microsoft's Echelon [92, 93, 144], or coarser elements can be used, e.g., projects in Google's TAP [65, 146, 149]).

A traditional RTS technique, e.g., `FaultTracer` [158], using methods for both the selection granularity and the coverage granularity would obtain the dependency matrix as in Figure 3.2a, where one dependent element is denoted as ClassName#MethodName. Note that we list the dependent elements for each test entity/row but do not show all the dependent elements as columns, because the matrices are fairly sparse. Ekstazi always uses classes (more generally, it uses files) for the coverage granularity and can use either methods or classes for the selection granularity. Using methods or classes obtains the dependency matrices as in Figure 3.2b or Figure 3.2c, respectively. (In principle, one could use methods for the coverage granularity and classes for the selection granularity, but this was not done traditionally, and we do not consider it.)

In Ekstazi, whenever a test entity depends on `D`, it also depends on `C` (in general, on all superclasses of `D`). Each test entity also depends on its test class, e.g., `t1` depends on `TestM`. Finally, this simple example does not show the test code or the code under test accessing any files, but Ekstazi also tracks files.

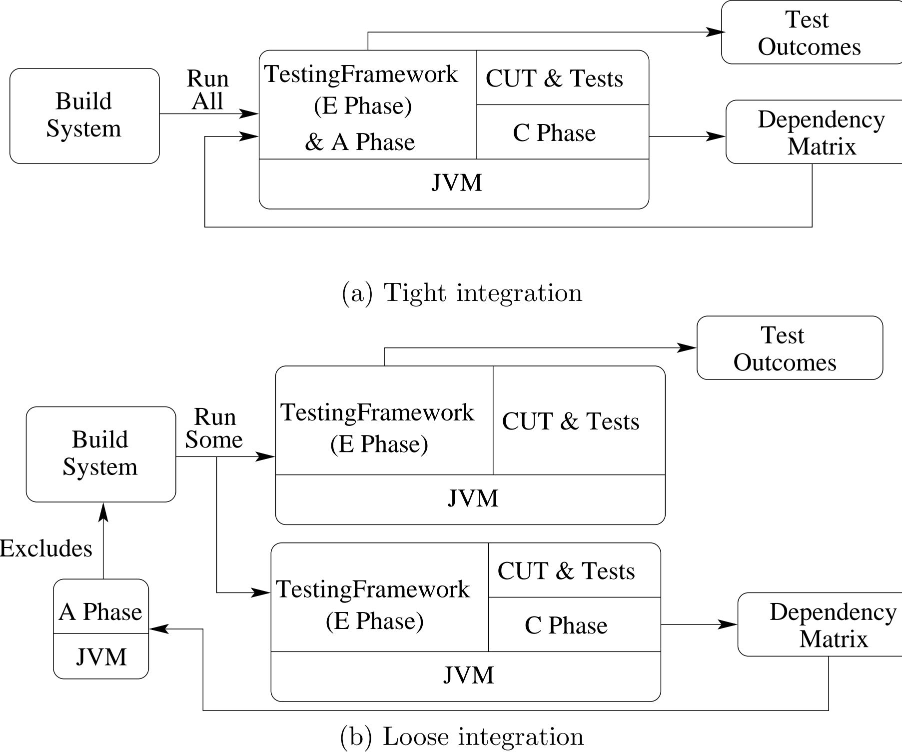
Assume that a new code revision changes only the body of the method `D.p` and thus

only the class `D`. `FaultTracer` would select to run only one test, `t4`. In contrast, Ekstazi at the method granularity would select two tests, `t2` and `t4`, because they both depend on the changed class `D`. Moreover, Ekstazi at the class granularity would select both test classes, `TestM` and `TestP`, and thus all four test methods, because both test classes depend on the changed class `D`.

At a glance, it seems that Ekstazi cannot be better than the traditional techniques, because Ekstazi never selects fewer tests. However, our goal is to optimize the end-to-end time for RTS. Although Ekstazi selects some more tests and thus has a longer execution phase, its use of much coarser dependencies shortens both the analysis and collection. As a result, Ekstazi has a much lower end-to-end time.

Safely using methods as the coverage granularity is expensive. An RTS technique that just intersects methods that are in the set of dependencies with the changes, as we discussed in our simplified description, is *unsafe*, i.e., it could miss to select some test that is affected by the changes. For example, the new revision could add a method `m` in class `D` (that overrides `C.m`); a naive intersection would not select any test, but the outcome of `t2` could change: the execution of this test on the old revision does depend on (the absence of) `D.m`, although the test could not execute that (non-existent) method [89, 129]. For another example, the new revision could change some field accessed from the existing method `C.m`; again, it would be necessary to reason about the change to determine which tests should be selected [129, 158].

As a consequence, an RTS technique that uses methods as the coverage granularity could be safer by collecting more dependencies than just covered methods (hence making the collection expensive and later selecting more tests, making the execution more expensive), and, more critically, it also needs sophisticated, expensive comparison of the old and new revisions to reason about the changes (hence making the analysis phase expensive). In contrast, an RTS technique that uses classes as the coverage granularity can be safer by simply collecting all accessed classes (hence speeding up the collection), and more critically, it can use a rather fast check of the new revision that does not even require the old revision let alone extensively comparing it with the new revision (hence speeding up the analysis phase). However, when tests depend not only on the code under test but also on external files [50,117], collecting only the classes is not safe, and hence Ekstazi uses files as dependencies.



(a) Tight integration

(b) Loose integration

Figure 3.3: Integration of an RTS technique in a typical build with a testing framework

3.2 Technique and Implementation

A typical RTS technique has three phases: the *analysis ($\mathcal{A}$) phase* selects what tests to run in the current revision, the *execution ($\mathcal{E}$) phase* runs the selected tests, and the *collection ($\mathcal{C}$) phase* collects dependencies from the current revision to enable the analysis for the next revision. Ekstazi collects dependencies at the level of files. For each test entity, Ekstazi saves (in the corresponding row of the dependency matrix) the names and checksums of the files that the entity accesses during execution.

In the rest of the section, we first describe the RTS phases in more detail. We then briefly discuss safety, describe the format in which Ekstazi saves the dependencies in the $\mathcal{C}$ phase and uses it in the $\mathcal{A}$ phase, and describe an optimization that is important to make Ekstazi practical. We finally describe Ekstazi integration with a testing framework.

3.2.1 Analysis ($\mathcal{A}$) Phase

The analysis phase in Ekstazi is quite simple (and thus fast). For each test entity, Ekstazi checks if the checksums of all accessed files are still the same (in the new revision) as they were (in the old revision). If so, the test entity is not selected to be run; otherwise, if any checksum differs, the test entity is selected to be run. Note that an executable file can remain the same even when its source file changes (e.g., renaming a local variable); dually, the executable file can change even when its source file remains the same (e.g., due to a change in compilation options). Comparing the checksums requires no sophisticated comparisons of the old and new revisions (which prior RTS research techniques usually perform on the source), and in fact, it does not even need to analyze the old revision (much like a build system can incrementally compile code just by knowing which source files changed). The only check is if the files remained the same.

Ekstazi naturally handles newly added test entities: if the dependency matrix has no information for some entity, then the entity is selected to be run. Initially, on the very first run of Ekstazi, there is no dependency matrix, and hence, it has no information for any entity, so all entities are selected to be run. Ekstazi also naturally handles deleted test entities because Ekstazi does not discover what entities to run, but it only filters what existing non-affected entities not to run among all the entities that are discovered by a testing framework or a build system (which does not discover deleted entities).

3.2.2 Execution ($\mathcal{E}$) Phase

Although one can initiate test execution directly from a testing framework, large projects typically initiate test execution from a build system that invokes the testing framework. Popular build systems (e.g., Ant or Maven) allow the user to specify an `includes` list of all test classes to execute (often specified as regular expressions in build configuration files such as `build.xml` or `pom.xml`) and configuration options to guide the test execution.

Figure 3.3 shows two approaches to integrate an RTS technique in a typical Java project. Ekstazi can work with testing frameworks in both approaches. When tightly integrating the $\mathcal{A}$ and $\mathcal{E}$ phases (Figure 3.3a), the build system finds all test classes and invokes a testing

framework on these classes as if all test entities will run; Ekstazi then checks for each entity if it should be actually run or not.

Loosely integrating the $\mathcal{A}$ and $\mathcal{E}$ phases (Figure 3.3b) can improve performance in some cases. It first determines what test entities *not* to run. This avoids the unnecessary overhead (e.g., loading classes or spawning a new JVM when the build spawns a JVM for each test entity) of preparing to run an entity and finding it should not run. The $\mathcal{A}$ phase makes an `excludes` list of test *classes* that should not run, and the build system ignores them before executing the tests. Without re-implementing the discovery of all tests, the $\mathcal{A}$ phase cannot make a list of all test classes to run (an `includes` list) because it could miss new tests (for which it has no rows in the dependency matrix). Ekstazi makes an `excludes` list from previously collected dependencies and excludes test *classes* rather than test *methods* because most build systems support an `excludes` list of classes. In case of the method selection granularity, the tests methods that are not affected are excluded at the beginning of the $\mathcal{E}$ phase.

Figure 3.3 also shows two approaches to integrate the $\mathcal{E}$ and $\mathcal{C}$ phases. First, the dependencies for the test entities that were not selected cannot change: these entities are not run and their corresponding rows in the dependency matrix do not change. But the test entities that were selected need to be run to determine if they still pass or fail, and thus to inform the user who initiated the test session. Because the dependencies for these entities change, the simplest way to update their rows in the dependency matrix is with *one pass* that both determines the test outcome and updates the rows. However, collecting dependencies has an overhead [152]. Therefore, some settings may prefer to use *two passes*: one pass without collecting dependencies, just to determine the test outcome and inform the user, and another pass to also collect the dependencies. The second pass can be started in parallel with the first pass or can be performed sequentially later.

3.2.3 Collection ($\mathcal{C}$) Phase

The collection phase creates the dependency matrix for the executed test entities. Ekstazi monitors the execution of the tests and the code under test to collect the set of files accessed

during execution of each entity, computes the checksum for these files, and saves them in (the corresponding rows of) the dependency matrix. Ekstazi currently collects *all* files that are either read or written, but it could be even more precise by distinguishing writes that do not create a dependency [86]. Moreover, Ekstazi tracks even files that were attempted to be accessed but did not exist; if those files are added later, the behavior can change.

In principle, we could collect file dependencies by adapting a tool such as Fabricate [70] or Memoize [115]: these tools can monitor any OS process to collect its file dependencies, and thus they could be used to monitor a JVM that runs tests. However, these tools would be rather *imprecise* for at least two reasons. First, they would not collect dependencies per entity when multiple entities run in one JVM. Second, they would not collect dependencies at the level of `.class` files archived in `.jar` files. Moreover, these tools are not portable from one OS to another, and also cannot be easily integrated in a testing framework such as JUnit or a build system such as Maven.

We implemented the $\mathcal{C}$ phase in Ekstazi as a pure Java library that is called from a testing framework and addresses both reasons of imprecision mentioned above. To collect dependencies per test entity, Ekstazi needs to be informed when an entity starts and ends. Ekstazi offers API methods `startCollectingDependencies(String name)`, which clears all previously collected dependencies, and `finishCollectingDependencies(String name)`, which saves all the collected dependencies to an appropriate row in the dependency matrix.

When using method selection granularity, due to common designs of testing frameworks, additional steps are needed to properly collect dependencies. Namely, many testing frameworks invoke a constructor of a test class only once, and then invoke `setUp` method(s) before each test method is invoked. Therefore, Ekstazi appends dependencies collected during constructor invocation and `setUp` methods(s) to the dependencies collected during the execution of each test method.

To precisely collect accessed files, Ekstazi dynamically instruments the bytecode and monitors the execution to collect both explicitly accessed files (through the `java.io` package) and implicitly accessed files (i.e., the `.class` files that contain the executed bytecode). Ekstazi collects explicitly accessed files by monitoring all standard Java library methods that may open a file (e.g., `FileInputStream`). In contrast, files that contain bytecode for

Java classes are not explicitly accessed during execution; instead, a class loader accesses a classfile when needed. If Ekstazi monitored only class loading, and two test entities access the same class, Ekstazi would collect the dependency on that class only for the test entities that accesses the class first, which would be unsafe. Our instrumentation collects a set of objects of the type `java.lang.Class` that a test depends on; Ekstazi then finds for each class where it was loaded from. If a class is not loaded from disk but dynamically created during execution, then it need not be tracked as a dependency, because it cannot change unless the code that generates it changes.

Instrumented Code Points: More precisely, Ekstazi instruments the following code points: (1) start of a constructor, (2) start of a static initializer, (3) start of a static method, (4) access to a static field, (5) use of a class literal, (6) reflection invocations, and (7) invocation through `invokeinterface` (bytecode instruction). Ekstazi needs no special instrumentation for the test class: it gets captured as a dependency when its constructor is invoked. Ekstazi also does not instrument the start of instance methods: if a method of class `C` is invoked, then an object of class `C` is already constructed, which captured the dependency on `C`. An alternative to instrumentation is to use debug interface, however recent work on tracing [105] showed that such an approach does not scale.

3.2.4 Safety

Ekstazi technique is safe for *any* code change and *any* change to the file system. The safety of Ekstazi intuitively follows from the proved safety of RTS based on class dependencies [142] and partial builds based on file dependencies [50]. We leave it as a future work to formally prove that Ekstazi is safe in above mentioned cases. Note that Ekstazi is unsafe if tests execute unmanaged code or access network. In other words, Ekstazi does not collect dependencies outside of a JVM process. Ekstazi with method selection granularity is also unsafe when there are enforced test-order dependencies [39, 87, 159]. Regarding non-deterministic tests (e.g., thread scheduling), Ekstazi collects dependencies for a *single* run and guarantees that the test will be selected if any of its dependencies changes. However, if a dependency changes for another run that was not observed, the test will not be selected.

This is the common approach in RTS [138] because collecting dependencies for all runs (e.g., using software model checking) would be costly. Also, prior studies [61] showed that changes in the test outcome are rare. After all, developers in practice, run each test only once.

3.2.5 Dependency Format

Ekstazi saves dependencies in a simple format similar to the dependency format of build tools such as Fabricate [70]. For each test entity (be it a test method or a test class), Ekstazi saves the dependencies (i.e., one row from the dependency matrix) in a separate file[2] whose name corresponds to the entity name. For example in Figure 3.2b, Ekstazi creates four files `TestM.t1`, `TestM.t2`, `TestP.t3`, and `TestP.t4`. Saving dependencies from all test entities together in one file would save space and could save time for smaller projects, but it would increase time for large projects that often run several test entities in parallel (e.g., spawn multiple JVMs for sets of test classes) so using one file would require costly synchronization on that file. In the future, we plan to explore other ways to persist the dependency matrix, e.g., in a database.

The file, which stores the dependencies, includes the names and checksums of the files that the test entity accesses during execution. These files are the executable files (e.g., standalone `.class` files or `.class` files packed in `.jar` files) or external resources (e.g., configuration files). The checksum effectively hashes the content of the files. For example, consider that some test class `TestP` for a test `t4` is in a jar file called `t.jar`, the code for classes `C` and `D` are in a jar file called `c.jar`, and the test also depends on a file called `config.xml`. If the (hexcode) checksums for those four files are, say, `1a2b`, `0864`, `dead`, and `beef`, then the file with dependencies for `t4` would have content `t.jar!TestP.class 1a2b, c.jar!C.class 0864, c.jar!D.class dead, config.xml beef`. These checksums allow Ekstazi to check changes with no explicit access to the old code revision.

[2]Note that a file that stores dependencies should not be confused with "dependent files", which are the dependencies themselves.

3.2.6 Smart Checksums

Ekstazi's use of file checksums offers several advantages, most notably (1) the old revision need not be available for the $\mathcal{A}$ phase, and (2) hashing to compute checksums is fast. On top of collecting the executable files (`.class`) from the archives (`.jar`) rather than collecting the entire archives, Ekstazi can compute the *smart checksum* for the `.class` files. Computing the checksum from the bytecodes already ignores some changes in the source code (e.g., `i++` and `i+=1` could be compiled the same way). The baseline approach computes the checksum from the entire content of a `.class` file, including all the bytecodes.

However, two somewhat different executable files may still have the same semantics in most contexts. For example, adding an empty line in a `.java` file would change the debug info in the corresponding `.class` file, but almost all test executions would still be the same (unless they explicitly observe the debug info, e.g., through exceptions that check line numbers). Ekstazi can ignore certain file parts, such as compile-time annotations and other debug info, when computing the checksum. The trade-off is that the smart checksum makes the $\mathcal{A}$ and $\mathcal{C}$ phases slower (rather than quickly applying a checksum on the entire file, Ekstazi needs to parse parts of the file and run the checksum on a part of the file), but it makes the $\mathcal{E}$ phase faster (as Ekstazi selects fewer tests because some dependent files match even after they change).

3.2.7 Integrating Ekstazi with JUnit

We implemented the Ekstazi technique in a robust tool for Java and JUnit. We integrated Ekstazi with JUnit because it is a widely used framework for executing unit tests in Java. Regarding the implementation, Ekstazi has to change (dynamically) a part of the JUnit core itself to allow skipping a test method that should not be run. While JUnit provides listeners that can monitor start and end of tests, currently the listeners cannot change the control-flow of tests. Ekstazi supports both JUnit 3 and JUnit 4, each with some limitation. For JUnit 3, Ekstazi supports only methods (not classes) as selection granularity. For JUnit 4, if a project uses a custom runner, Ekstazi supports only classes (not methods); otherwise, if no custom runner is used, Ekstazi supports both classes and methods. Project developers

could hook their custom runner to EKSTAZI by adding proper calls at the start and end of each test entity, and allowing EKSTAZI to determine if a test should be run (Section 3.4). It is important to note that when EKSTAZI does not support some case, it simply offers no test selection and runs all the tests, as RetestAll.

3.3 Evaluation

This section describes an experimental evaluation of EKSTAZI. We (1) describe the projects used in the evaluation, (2) describe the experimental setup, (3) report the RTS results in terms of both the number of selected test entities and the end-to-end time, (4) measure benefits of the smart checksum, (5) evaluate the importance of selection granularity, (6) evaluate the importance of coverage granularity by comparing EKSTAZI with `FaultTracer` [158], and (7) describe a case study of EKSTAZI integration with a popular open-source project.

We ran all the experiments on a 4-core 1.6 GHz Intel i7 CPU with 4GB of RAM, running Ubuntu Linux 12.04 LTS. We used three versions of Oracle Java 64-Bit Server: 1.6.0_45, 1.7.0_45, and 1.8.0_05. Different versions were necessary as several projects require specific older or newer Java version. For each project, we used the latest version of Java that successfully compiled and executed all tests.

3.3.1 Projects

Table 3.1 lists the projects used in the evaluation; all 32 projects are open source. The set of projects was created by three undergraduate students who were not familiar with our study. We suggested starting places that may contain open-source projects: Apache Projects [1], GitHub [4], and GoogleCode [5]. We also asked that each project satisfies several requirements: (1) has the latest available revision (obtained at the time of the first download) build without errors (using one of three Java versions mentioned above), (2) has at least 100 JUnit tests, (3) uses Ant or Maven to build code and execute tests, and (4) uses SVN or Git version-control systems. The first two requirements were necessary to consider compilable, non-trivial projects, but the last two requirements were set to simplify our

automation of the experiments.

Note that EKSTAZI itself does *not* require any integration with VCS, but our experiments do require to automate checking out of various project revisions. To support both Ant and Maven across many project revisions, we do not modify the `.xml` configuration files but replace the appropriate `junit.jar` (in the `lib` for Ant-based projects or in the Maven `.m2` download repo) with our `ekstazi.jar`. Note that this is *not* how one would use EKSTAZI in practice but it is only done for the sake of the experiments; we described in Section 3.4 how users can integrate EKSTAZI with their projects. From about 100 projects initially considered from the three source-code repositories, two-thirds were excluded because they did not build (e.g., due to syntax errors or missing dependencies), used a different build systems (e.g., Gradle), or had too few tests. The students confirmed that they were able to execute JUnit tests in all selected projects.

Table 3.1 tabulates for each project its name, revision (that was the latest available revision of the project at the time of our first download), the number of revisions that could build (out of 20 revisions *before* the specified revision), and the total number of lines of code (as reported by SLOCCount [143]). Table 3.2 tabulates the number of JUnit test methods and classes (averaged across all buildable revisions), and the average (avg) and total ($\sum$) time to execute the entire test suite across all buildable revisions. The remaining columns are discussed in the following sections. Table 3.3 describes symbols used for column titles in several other tables.

The row labeled $\sum$ at the bottom of tables 3.1 and 3.2 shows the cumulative numbers across all projects. In sum, we performed our evaluation on 615 revisions of 32 projects totaling 4,937,189 LOC and 773,565 test methods. To the best of our knowledge, this is the largest dataset used in any RTS study.

We visually separate projects with *short running* and *long running* test suites. While no strict rule defines the boundary between the two, we classified the projects whose test suites execute in less than one minute as short running. The following sections mostly present results for *all* projects together, but in several cases we contrast the results for projects with short- and long-running test suites.

	Project	Repository Location	Revisions		LOC
			SHA	Buildable	
short running	Cucumber	[52]	5df09f85	20	19,939
	JodaTime^M	[104]	f17223a4	20	82,996
	Retrofit	[130]	810bb53e	20	7,389
	CommonsValidator^M	[35]	1610469	20	12,171
	BVal	[15]	1598345	20	17,202
	CommonsJXPath^M	[31]	1564371	13	24,518
	GraphHopper	[81]	0e0e311c	20	33,254
	EmpireDB	[27]	1562914	20	43,980
	River	[34]	1520131	19	297,565
	Functor	[28]	1541713	20	21,688
	JFreeChart	[102]	3070	20	140,575
	CommonsColl4	[18]	1567759	20	52,040
	CommonsLang3	[22]	1568639	20	63,425
	CommonsConfig	[19]	1571738	16	55,187
	PdfBox	[33]	1582785	20	109,951
	GSCollections	[84]	6270110e	20	920,208
long running	ClosureCompiler	[80]	65401150	20	211,951
	CommonsNet	[24]	1584216	19	25,698
	CommonsDBCP	[20]	1573792	16	18,759
	Log4j^M	[32]	1567108	19	30,287
	JGit^M	[103]	bf33a6ee	20	124,436
	CommonsIO	[21]	1603493	20	25,981
	Ivy^M	[30]	1558740	18	72,179
	Jenkins (light)	[100]	c826a014	20	112,511
	CommonsMath	[23]	1573523	20	186,796
	Ant^M	[14]	1570454	20	131,864
	Continuum^M	[25]	1534878	20	91,113
	Guava^M	[85]	af2232f5	16	257,198
	Camel (core)	[17]	f6114d52	20	604,301
	Jetty	[101]	0f70f288	20	282,041
	Hadoop (core)	[29]	f3043f97	20	787,327
	ZooKeeper^M	[36]	1605517	19	72,659
$\sum$	-	-	-	615	4,937,189

Table 3.1: Statistics for projects used in the evaluation of EKSTAZI

	Project	Test [*avg*]		Time [sec]		Test Selection [%]		
		classes	methods	*avg*	$\sum$	e%	$t^{\mathcal{AEC}}$	$t^{\mathcal{AE}}$
short running	Cucumber	49	296	8	169	12	99	76
	JodaTime^{M}	124	4,039	10	214	21	107	75
	Retrofit	15	162	10	217	16	104	90
	CommonsValidator^{M}	61	416	11	230	6	88	78
	BVal	21	231	13	267	13	138	97
	CommonsJXPath^{M}	33	386	15	205	20	94	81
	GraphHopper	80	677	15	303	16	85	59
	EmpireDB	23	113	27	546	18	112	99
	River	14	83	17	335	6	35	18
	Functor	164	1,134	21	439	13	112	90
	JFreeChart	359	2,205	30	618	5	80	64
	CommonsColl4	145	13,684	32	644	9	66	55
	CommonsLang3	121	2,492	36	728	11	60	53
	CommonsConfig	141	2,266	39	633	20	72	58
	PdfBox	94	892	40	813	12	80	63
	GSCollections	1,106	64,614	51	1,036	29	107	90
long running	ClosureCompiler	233	8,864	71	1,429	17	62	50
	CommonsNet	37	215	68	1,300	10	21	21
	CommonsDBCP	27	480	76	1,229	21	46	39
	Log4j^{M}	38	440	79	1,508	6	62	43
	JGit^{M}	229	2,223	83	1,663	22	65	50
	CommonsIO	84	976	98	1,969	12	30	24
	Ivy^{M}	121	1,005	170	3,077	38	53	44
	Jenkins (light)	86	3,314	171	3,428	7	74	71
	CommonsMath	461	5,859	249	4,996	6	77	16
	Ant^{M}	234	1,667	380	7,613	13	24	21
	Continuum^{M}	68	361	453	9,064	10	32	26
	Guava^{M}	348	641,534	469	7,518	13	45	17
	Camel (core)	2,015	4,975	1,296	25,938	5	9	7
	Jetty	504	4,879	1,363	27,275	26	57	49
	Hadoop (core)	317	2,551	1,415	28,316	7	38	22
	ZooKeeper^{M}	127	532	2,565	48,737	20	43	37
$\sum$		7,479	773,565	9,400	182,475	-	-	-
		avg(all)				**14**	**68**	**53**
	avg(for short running) | avg(for long running)					14|**15**	90|**46**	72|**34**

Table 3.2: Test selection results using Ekstazi

c%	Percentage of test classes selected
e%	Percentage of test entities selected
m%	Percentage of test methods selected
$t^{\mathcal{AEC}}$	Time for $\mathcal{AEC}$ normalized by time for RetestAll
$t^{\mathcal{AE}}$	Time for $\mathcal{AE}$ normalized by time for RetestAll

Table 3.3: Legend for symbols used in tables

3.3.2 Experimental Setup

We briefly describe our experimental setup. The goal is to evaluate how Ekstazi performs if RTS is run for each committed project revision. In general, developers may run RTS even between commits (see Chapter 2), but there is no such dataset for the selected projects that would allow *executing* tests the same way that developers executed them in between commits. For each project, our experimental script checks out the revision that is 20 revisions *before* the revision specified in Table 3.1. If any revision cannot build, it is ignored from the experiment. If it can build, the script executes the tests in three scenarios: (1) RetestAll executes all tests (without Ekstazi integration), (2) $\mathcal{AEC}$ executes the tests with Ekstazi while collecting dependencies in all three phases (as a developer would use the tool), and (3) $\mathcal{AE}$ executes the tests with Ekstazi but without collecting dependencies, i.e., only the first two phases (for experiments and comparison with prior work). The script then repeats these steps for all revisions until reaching the latest available revision listed in Table 3.1.

In each step, the script measures the number of executed tests (all tests for JUnit or selected tests for Ekstazi) and the testing time (the execution of all tests for JUnit, the end-to-end time for all $\mathcal{AEC}$ phases of Ekstazi, or just the times for the $\mathcal{AE}$ phases of Ekstazi). The script measures the time to *execute the build command* that the developers use to execute the tests (e.g., `ant junit-tests` or `mvn test`). Finding the appropriate command took a bit of effort because different projects use different build target names, or the entire test suites for the largest projects run too long to perform our experiments on multiple revisions in reasonable time. We sometimes limited the tests to a part of the entire project (e.g., the `core` tests for `Hadoop` in RetestAll take almost 8 hours across 20 revisions, and the full test

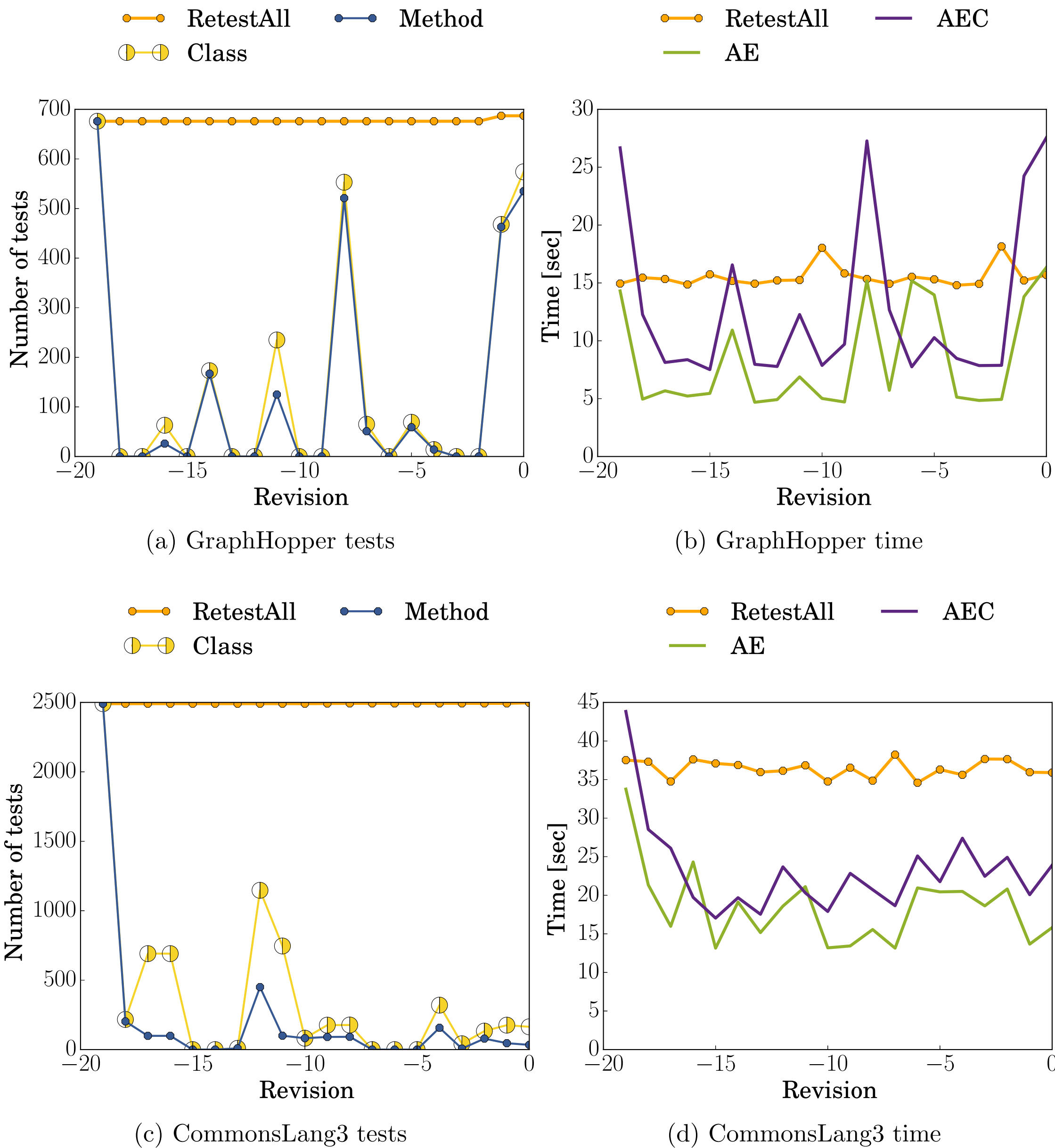


Figure 3.4: Number of available and selected tests (a,c) and end-to-end time (b,d)

suite takes over 17 hours for just one revision). We did not modify anything in the build configuration for running the tests, e.g., whether it uses multiple cores or spawns JVMs. By measuring the time for the build command, we evaluate the speed up that the developers would have observed had they used Ekstazi. Note that the speed up that Ekstazi provides over RetestAll is even *bigger* for the testing itself than for the build command, because the build command has some fixed cost before initiating the testing. But the developer observes

the build command, and hence, we do not measure just the testing time.

Ekstazi has two main options: selection granularity can be class or method, and smart checksum can be on or off. The default configuration uses the class selection granularity with smart checksum. As discussed earlier, due to idiosyncrasies of JUnit 3, Ekstazi does not run the class selection granularity for all projects; those that use the method selection granularity have the superscript M in tables 3.1 and 3.2.

3.3.3 Main RTS Results

The testing time is the key metric to compare RetestAll, Ekstazi $\mathcal{AEC}$, and Ekstazi $\mathcal{AE}$ runs; as an additional metric, we use the number of executed tests. Figure 3.4 visualizes these metrics for two of the projects, `GraphHopper` and `CommonsLang3`. Plots for other projects look similar; the two shown projects include several revisions that are interesting to highlight. For each of the 20 revisions, we plot the total number of test methods (close to 700 in `GraphHopper` and 2,500 in `CommonsLang3`), the number of test methods Ekstazi selects at the method level (blue line), the number of test methods Ekstazi selects at the class level (yellow line), the time for RetestAll (orange line), the time for all $\mathcal{AEC}$ phases of Ekstazi at the method level (purple line), and the time for only $\mathcal{AE}$ phases at the method level (green line). For example, revision −12 for `CommonsLang3` has about 400 and 1,200 test methods selected at the method and class level, respectively. We compute the selection ratio for each revision, in this case ~400/2500 and ~1200/2500, and then average the ratios over all revisions by computing their arithmetic mean; for `CommonsLang3`, the average ratio is about 8% of methods and 11% of classes (Table 3.5). Likewise for times, we compute the ratio of the Ekstazi time over the RetestAll time for each revision and then average these ratios. In many revisions, Ekstazi is faster than RetestAll, but in some cases, it is slower, e.g., revisions −1, −8, and −14 for `GraphHopper`. In all starting revisions, −20, we expect the Ekstazi $\mathcal{AEC}$ to be slower than RetestAll, as Ekstazi runs all the tests and collects dependencies. We also expect Ekstazi $\mathcal{AE}$ runs to be faster than Ekstazi $\mathcal{AEC}$ runs, but there are some cases where the background processes flip that, e.g., revisions −5 and −6 for `GraphHopper`. The background noise also makes the time for RetestAll to fluctuate, but over

a large number of revisions we expect the noise to cancel out and allow a fair comparison of times for RetestAll, Ekstazi $\mathcal{AEC}$, and Ekstazi $\mathcal{AE}$. The noise has smaller effect on long running test suites.

For each revision, we compute for both the $\mathcal{AEC}$ and $\mathcal{AE}$ runs: (1) the ratio of selected tests over all available tests and (2) the ratio of testing time over the time for RetestAll. We next compute the average (arithmetic mean) values for ratios over all revisions. The last three columns in Table 3.2 show the average selection per project; "e%" shows the ratio of test entities (methods or classes) selected, and the times for $\mathcal{AEC}$ and $\mathcal{AE}$ are normalized to the JUnit run without Ekstazi (i.e., RetestAll). For example, for `Cucumber`, Ekstazi selects on average 12% of test entities, but the time that Ekstazi takes is 99% of RetestAll (or 76% if the $\mathcal{C}$ phase is ignored), so it provides almost no benefit. In fact, for some other projects with short-running test suites, Ekstazi is slower than RetestAll; we highlight such cases, e.g., for `JodaTime` in Table 3.2.

Overall, the selection ratio of test entities varies between 5% and 38%, the time for $\mathcal{AEC}$ varies between 9% and 138% (slowdown), and the time for $\mathcal{AE}$ varies between 7% and 99%. On average, across all the projects, the $\mathcal{AEC}$ time is 68%, and the $\mathcal{AE}$ time is 53%. More importantly, all slowdowns are for projects with short-running test suites. Considering only the projects with long-running test suites, Ekstazi reduces the $\mathcal{AEC}$ time to 46% of RetestAll, and reduces the $\mathcal{AE}$ time to 34%. In sum, Ekstazi appears useful for projects whose test suites take over a minute: Ekstazi on average roughly halves their testing time.

3.3.4 Smart Checksums

Recall that smart checksum performs a more expensive comparison of `.class` files to reduce the number of selected test entities (Section 3.2.6). Table 3.4 shows a comparison of Ekstazi runs with smart checksum being off and on, for a diverse subset of projects. While smart checksum improves both the number of selected entities and the end-to-end testing time (on average and in most cases), there are several cases where the results are the same, or the reduction in the testing time is even slightly lower, e.g., both times for `Jenkins` or the $\mathcal{AE}$ time for `CommonsMath`. This happens if projects have no revision (in the last 20 revisions)

Project	**All**			**Smart**		
	m%	$t^{\mathcal{AEC}}$	$t^{\mathcal{AE}}$	m%	$t^{\mathcal{AEC}}$	$t^{\mathcal{AE}}$
Camel (core)	5	9	7	5	9	7
CommonsDBCP	40	60	47	23	43	37
CommonsIO	21	42	32	14	30	24
CommonsMath	6	85	16	6	75	17
CommonsNet	11	28	28	9	26	22
CommonsValidator	7	93	79	6	88	78
Ivy	47	63	52	38	53	44
Jenkins (light)	14	72	69	7	74	71
JFreeChart	6	87	70	5	84	67
avg(all)	17	60	45	13	54	41

Table 3.4: EKSTAZI without and with smart checksum

that modifies only debug info; using smart checksum then leads to a slowdown as it never selects fewer tests but increases the cost of checking and collecting dependencies. We also manually inspected the results for several projects and found that smart checksum can be further improved: some `.class` files differ only in the order of annotations on methods, but Java specification does not attach semantics to this order, so such changes can be safely ignored. In sum, smart checksum reduces the overall testing time.

3.3.5 Selection Granularity

EKSTAZI provides two levels of selection granularity: methods (which selects fewer tests for the $\mathcal{E}$ phase but makes the $\mathcal{A}$ and $\mathcal{C}$ phases slower) and classes (which makes the $\mathcal{A}$ and $\mathcal{C}$ phases faster but selects more tests for the $\mathcal{E}$ phase). Table 3.5 shows a comparison of EKSTAZI runs for these two levels, on several randomly selected projects. Because EKSTAZI does not support method selection granularity for projects that use a custom JUnit runner, we do not compare for such projects. Also, we do not compare for `Guava`; it has a huge number of test methods, and with method selection granularity, our default format for saving dependencies (Section 3.2.5) would create a huge number of files that may exceed limits set by the file system. The class selection granularity improves both $\mathcal{AEC}$ and $\mathcal{AE}$

Project	**Method**			**Class**		
	m%	$t^{\mathcal{AEC}}$	$t^{\mathcal{AE}}$	c%	$t^{\mathcal{AEC}}$	$t^{\mathcal{AE}}$
BVal	16	138	94	13	138	97
ClosureCompiler	20	96	53	17	62	50
CommonsColl4	7	81	60	9	66	55
CommonsConfig	19	76	57	20	72	58
CommonsDBCP	23	43	37	21	46	39
CommonsIO	14	30	24	12	30	24
CommonsLang3	8	63	51	11	60	53
CommonsMath	6	75	17	6	77	16
CommonsNet	9	26	22	10	21	21
Cucumber	13	105	78	12	99	76
EmpireDB	13	117	100	18	112	99
Functor	15	111	100	13	112	90
GraphHopper	19	84	54	16	85	59
GSCollections	16	198	101	29	107	90
JFreeChart	5	84	67	5	80	64
PdfBox	8	85	70	12	80	63
Retrofit	19	113	93	16	104	90
River	6	34	17	6	35	18
avg(all)	13	87	61	14	77	59

Table 3.5: EKSTAZI with method and class selection granularity

times on average and in most cases, especially for `GSCollections`. In some cases where the class selection granularity is not faster, it is only slightly slower. In sum, the class selection granularity reduces the overall testing time compared to the method selection granularity, and the class selection granularity should be the default value.

3.3.6 Coverage Granularity

The results so far show that a coarser level of capturing dependencies is not necessarily worse: although it selects more tests, it can lower the overall time; in fact, the class *selection granularity* does have a lower overall time than the method *selection granularity*. We next evaluate a similar question for *coverage granularity*.

We compare EKSTAZI, which uses the file coverage granularity, with `FaultTracer` [158], which tracks dependencies on the edges of an extended control-flow graph (ECFG). To the

Project	FaultTracer		Ekstazi	
	m%	$t^{\mathcal{AEC}}$	m%	$t^{\mathcal{AEC}}$
CommonsConfig	8	223	19	76
CommonsJXPath	14	294	20	94
CommonsLang3	1	183	8	63
CommonsNet	2	57	9	26
CommonsValidator	1	255	6	88
JodaTime	3	663	21	107
avg(*all*)	5	279	14	76

Table 3.6: Test selection with FaultTracer and Ekstazi

best of our knowledge, FaultTracer was the only available tool for RTS. FaultTracer collects the set of ECFG edges covered during the execution of each *test method*. For comparison purposes, we also use Ekstazi with the method selection granularity. FaultTracer implements a sophisticated change-impact analysis using the Eclipse [60] infrastructure to parse and traverse Java sources of two revisions. Although robust, FaultTracer has several limitations: (1) it requires that the project be an Eclipse project, (2) the project has to have only a single module, (3) it does not track dependencies on external files, (4) it requires that both source revisions be available, (5) it does not track reflection calls, (6) it does not select newly added tests, (7) it does not detect any changes in the test code, and (8) it cannot ignore changes in annotations that Ekstazi ignores via smart checksum [6]. Due to these limitations, we had to discard most of the projects from the comparison, e.g., 15 projects had multiple modules, and for CommonsIO, FaultTracer was unable to instrument the code.

Table 3.6 shows a comparison of FaultTracer and Ekstazi, with the values, as earlier, first normalized to the savings compared to RetestAll for one revision, and then averaged across revisions. The Ekstazi results are the same as in Table 3.2 and repeated for easier comparison. The results show that Ekstazi has a much lower end-to-end time than FaultTracer, even though Ekstazi does select more tests to run. Moreover, the results show that FaultTracer is even slower than RetestAll.

To gain confidence in the implementation, we compared the sets of tests selected by Ekstazi and FaultTracer. Our check confirmed that the Ekstazi results were correct.

Project	**Link at `https://github.com/`**
Apache Camel	apache/camel
Apache Commons Math	apache/commons-math
Apache CXF	apache/cxf
Camel File Loadbalancer	garethahealy/camel-file-loadbalancer
JBoss Fuse Examples	garethahealy/jboss-fuse-examples
Jon Plugins	garethahealy/jon-plugins
Zed	hekonsek/zed

Table 3.7: Current Ekstazi users

In most cases, Ekstazi selected a superset of tests selected by `FaultTracer`. (Note that `FaultTracer` could have incorrectly selected a smaller number of tests than Ekstazi because `FaultTracer` is unsafe for some code changes. However, we have not encountered those cases in our experiments, most likely because we could run `FaultTracer` only with small projects due to issues described above.) In a few cases, Ekstazi (correctly) selected fewer tests than `FaultTracer` for two reasons. First, Ekstazi can select fewer tests due to smart checksum (Section 3.2.6). Second, Ekstazi ignores changes in source code that are not visible at the bytecode level, e.g., local variable rename (Section 3.2.1).

3.3.7 Apache CXF Case Study

Several (Apache) projects (Table 3.7) integrated Ekstazi into their main repositories. Note that Ekstazi was integrated in these projects after we selected the projects for the evaluation, as explained in Section 3.3.2. We evaluated how Ekstazi performed on one of these projects (Apache CXF) over 80 selected recent revisions, after Ekstazi was included in the project. Figure 3.5 shows how Ekstazi compares with RetestAll in terms of the end-to-end time. The plot shows that Ekstazi brought substantial savings to Apache CXF.

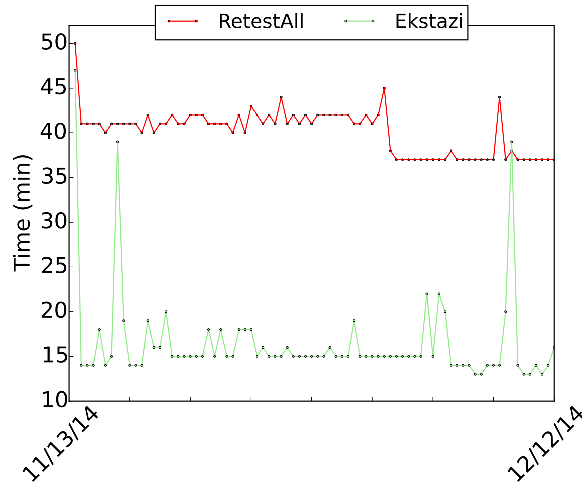


Figure 3.5: End-to-end `mvn test` time for Apache CXF

3.4 Usage

We describe in this section how users can integrate Ekstazi with projects that use Maven or Ant. We also describe programmatic invocation of Ekstazi that could be used to integrate Ekstazi with other testing frameworks (e.g., TestNG). Finally, we describe several options to tune behavior of Ekstazi. Ekstazi is currently distributed as a binary [62].

3.4.1 Integration with Maven

Ekstazi distribution includes a Maven plugin [62], available from Maven central. Only a single step is required to integrate Ekstazi with the existing build configuration files (i.e., `pom.xml`): include Ekstazi in the list of plugins. The plugin should be included in the same list of plugins as Maven Surefire plugin.

```
<plugin>
  <groupId>org.ekstazi</groupId>
  <artifactId>ekstazi-maven-plugin</artifactId>
  <version>${ekstazi.version}</version>
</plugin>
```

where ${ekstazi.version} denotes the version of Ekstazi.

One can also setup a Maven profile to enable test runs with Ekstazi only when explicitly requested, e.g., `mvn test -Pekstazi`. Developers may use Ekstazi on their machines but still

prefer to run all the tests on gated check-in. Using a Maven profile is a common approach for integrating third-party tools, e.g., code coverage tools, into a project.

```
<profile>
  <id>ekstazi</id>
  <activation><property><name>ekstazi</name></property></activation>
  <build>
    <plugins>
      <plugin>
        <groupId>org.ekstazi</groupId>
        <artifactId>ekstazi-maven-plugin</artifactId>
        <version>${ekstazi.version}</version>
        <executions>
          <execution>
            <id>ekstazi</id>
            <goals><goal>select</goal></goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

3.4.2 Integration with Ant

The Ekstazi distribution also includes an Ant task [62] that can be easily integrated with the existing build definitions (i.e., `build.xml`). The Ekstazi Ant task follows the common integration approach; the following three steps are required:

(1) add namespace definitions to the project element:

```
<project ... xmlns:ekstazi="antlib:org.ekstazi.ant">
```

(2) add the Ekstazi task definition:

```
<taskdef uri="antlib:org.ekstazi.ant" resource="org/ekstazi/ant/antlib.xml">
  <classpath path="org.ekstazi.core-${ekstazi.version}.jar"/>
  <classpath path="org.ekstazi.ant-${ekstazi.version}.jar"/>
</taskdef>
```

(3) wrap the existing JUnit target elements with Ekstazi select:

```
<ekstazi:select><junit fork="true" ...> ... </junit></ekstazi:select>
```

3.4.3 Programmatic Invocation

Programmatic invocation provides an extension point to integrate Ekstazi with other testing frameworks (e.g., TestNG). Ekstazi offers three API calls to check if any dependency

is modified, to start collecting dependencies, and to finish collecting dependencies:

```
org.ekstazi.Ekstazi.inst().checkIfAffected("name")
org.ekstazi.Ekstazi.inst().startCollectingDependencies("name")
org.ekstazi.Ekstazi.inst().finishCollectingDependencies("name")
```

where "name" is used as an id to refer to the collected dependencies for a segment of code (e.g., a fully qualified test class name). These primitives can be invoked from any JVM code. For example, to integrate Ekstazi with JUnit, we implement a listener that invokes `startCollectingDependencies` before JUnit executes the first test method in a class and invokes `finishCollectingDependencies` after JUnit executes the last test method in a class.

3.4.4 Options

Ekstazi provides several options to customize its behavior: (1) `forceall` (`boolean`) can be used to force the execution of all tests (even if they are not affected by recent changes) and recollect dependencies, (2) `forcefailing` (`boolean`) can be used to force the execution of the tests that failed in the previous run (even if they are not affected by recent changes), (3) `skipme` (`boolean`) can be set to true to have all tests run without Ekstazi, (4) `root.dir` (`File`) can be used to specify a directory where the dependencies for each test entity are stored, (5) `dependencies.append` (`boolean`) can be set to true to indicate that newly collected dependencies should be appended to the existing dependencies for the same test entity, (6) `hash.algorithm` ({`Adler,CRC32,MD5`}) can be used to specify the algorithm to be used to compute the checksums of collected dependencies, and (7) `hash.without.debuginfo` (`boolean`) can be set to false to indicate that debug Java information should be included when computing the checksum. The Ekstazi options can be specified in `.ekstazirc` file (saved either in home directory or the current working directory), which is loaded when Ekstazi is started.

3.5 Discussion

In this section we briefly discuss our results and the implications of using Ekstazi with various tests.

Coarser Dependencies Can Be Faster: We argue that the cost of `FaultTracer` is due to its *approach* and not just due to its implementation being a *research tool*. The cost of `FaultTracer` stems from collecting fine-grained dependencies, which affects both the $\mathcal{A}$ and $\mathcal{C}$ phases. In particular, the $\mathcal{A}$ phase needs to parse both old and new revisions and compare them. While Orso et al. [124] show how some of that cost can be lowered by filtering classes that did not change, their results show that the overhead of parsing and comparison still ranges from a few seconds up to 4min [124]. Moreover, collecting fine-grained dependencies is also costly. For example, we had to stop `FaultTracer` from collecting ECFG dependencies of `CommonsMath` after one hour; it is interesting to note that `CommonsMath` also has the most expensive $\mathcal{C}$ phase for Ekstazi (~8X for the initial run when there is no prior dependency information). Last but not least, in terms of adoption, `FaultTracer` and similar approaches are also more challenging than Ekstazi because they require access to the old revision through some integration with version-control systems. In contrast, Ekstazi only needs the checksums of dependent files from the old revision.

Sparse Collection: Although Ekstazi collects dependencies at each revision by default, one can envision collecting dependencies at every n-th revision. Note that this approach is safe as long as the analysis phase checks the changes between the current revision and the latest revision for which dependencies were collected. This approach avoids the cost of frequent collection but leads to less precise selection [49, 63].

Duplicate Tests: In a few cases, we observed that Ekstazi did not run some tests during the first run, which initially seemed like a bug in Ekstazi. However, inspecting these cases showed that some test classes were (inefficiently) included multiple times in the same test suite by the original developers. When JUnit (without Ekstazi) runs these classes, they are indeed executed multiple times. But when Ekstazi runs these test

suites, after it executes a test for the first time, it saves the test's dependencies, so when the test is encountered again for the same test suite, all its dependencies are the same, and the test is ignored. Effectively, the developers of these projects could speed up their test suites by rewriting the build configurations or the test suites to not run the same tests multiple times. More precisely, if the same test method *name* is encountered *consecutively* multiple times, EKSTAZI does not ignore the non-first runs but instead unions the dependencies for those invocations, to support parameterized unit tests [148].

Parameterized Tests: Recent versions of JUnit support parameterized unit tests [148]. A parameterized test defines a set of input data and invokes a test method with each input from the set; each input may contain multiple values. This approach is used in data-driven scenarios where only the test input changes, but the test method remains the same. Currently, EKSTAZI considers a parameterized unit test as a single test and unions the dependencies collected when executing the test method with each element from the input data set. In the future, we could explore tracking individual invocations of parameterized tests.

Flaky Tests: Tests can have non-deterministic executions for multiple reasons such as multi-threaded code, asynchronous calls, time dependencies, etc. If a test passes and fails for the same code revision, it is often called a "flaky test" [112]. Even if a test has the same outcome, it can have different dependencies in different runs. EKSTAZI collects dependencies for a *single* run and guarantees that the test will be selected if any of its dependencies from that run changes. However, if a dependency changes for another run that was not observed, the test will not be selected. Considering only one run is the common approach in RTS [138] because collecting dependencies for all runs (e.g., using software model checking) would be costly.

Dependent Tests: Some test suite have (order) dependencies among tests [39,87,159], e.g., if a test `t1` executes before test `t2`, then `t2` passes, but otherwise it fails. EKSTAZI does not detect such dependencies among the tests. This ignoring of dependencies could

lead to test failures if Ekstazi selects a test (e.g., `t2`) that depends on an unselected test (e.g., `t1`). In case when developers intentionally create dependencies (e.g., for performance reasons), RTS would have to be dependency-aware.

Parallel Execution vs. RTS: It may be (incorrectly) assumed that RTS is not needed in the presence of parallel execution. However, even companies with an abundance of resources cannot keep up with running all tests for every revision [65, 146, 149]. Additionally, parallel test execution is orthogonal to RTS. Namely, RTS can significantly speed up test execution even if the tests are run in parallel. For example, tests for four projects used in our evaluation (`ClosureCompiler`, `Hadoop`, `Jenkins`, and `JGit`) execute by default on all available cores; in our case, tests were running on four cores. Still, we can observe a substantial speedup in testing time when Ekstazi is integrated even in these projects. Moreover, RTS itself can be parallelized. In loose integration (Section 3.2.2), we can run $\mathcal{A}$ of all tests in parallel and then run $\mathcal{EC}$ of all tests in parallel. In tight integration, we can run $\mathcal{AEC}$ of all tests in parallel.

Unpredictable Time: One criticism that we heard about RTS is that the time to execute the test suite is not known by the developer in advance and is highly dependent on the changes made since the previous test session. If the $\mathcal{A}$ and $\mathcal{E}$ phases are together, Ekstazi indeed cannot easily estimate the execution time (or the number of tests), because it checks the dependencies for a test entity just before that entity starts the execution. On the other hand, if the $\mathcal{A}$ and $\mathcal{E}$ phases are separate, Ekstazi can estimate the execution time as soon as the $\mathcal{A}$ phase is done, based on the time for each test entity recorded during the previous runs.

3.6 Threats to Validity

In this section we describe several threats to the validity of our evaluation of Ekstazi.

External: The set of projects that we used in the evaluation may not be representative. To mitigate this threat, we performed our experiments on a large number of projects

that vary in size of code and tests, number of developers, number of revisions, and application domain. The set of evaluated projects is by far the largest set of projects used in any RTS study.

We performed experiments on 20 revisions per project; the results could differ if we selected more revisions or different segments from the software history. We considered only 20 revisions to limit the machine time needed for experiments. Further, we consider each segment right before the latest available revision (at the time when we started the experiments on the project).

The reported results for each project were obtained on a single machine. The results may differ based on the configuration (e.g., available memory). We also tried a small subset of experiments on another machine and observed similar results in terms of speed up, although the absolute times differed due to machine configurations. Because our goal is to compare real time, we did not want to merge experimental results from different machines.

Internal: Ekstazi implementation may contain bugs that may impact our conclusions. To increase the confidence in our implementation, we reviewed the code, tested it on a number of (small) examples, and manually inspected several results for both small and large projects.

Construct: Although many RTS techniques have been proposed in the past, we compared Ekstazi only with `FaultTracer`. To the best of our knowledge, `FaultTracer` was the only other available RTS tool. Our focus in comparison is not only on the number of selected tests but primarily on the end-to-end time taken for testing. We believe that the time that the developer observes, from initiating the test-suite execution for the new code revision until all the test outcomes become available, is the most relevant metric for RTS.

3.7 Summary

This chapter introduced a novel, efficient RTS technique, called Ekstazi, that is being adopted in practice. Ekstazi substantially speeds up regression testing (by reasoning about two project revisions). Ekstazi's use of coarse-grain dependencies, coupled with a few optimizations, provides a "sweet-spot" between $\mathcal{AC}$ phases and $\mathcal{E}$ phase, which results in the lowest end-to-end time.

CHAPTER 4

Regression Test Selection for Distributed Software Histories

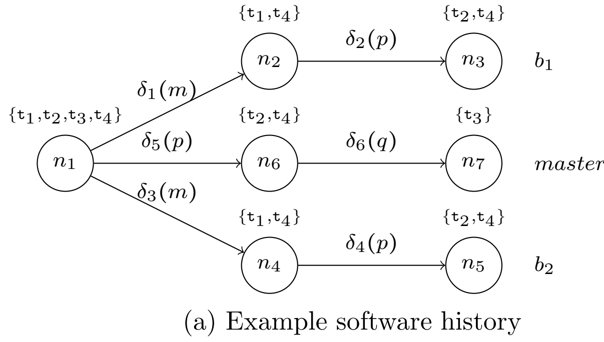
This chapter presents the contributions of the EKSTAZI approach that was developed with the aim to improve efficiency of any RTS technique for software that uses *distributed version control systems*. This chapter is organized as follows. Section 4.1 presents our running example that illustrates three options for our approach. Section 4.2 formalizes our RTS options for distributed software histories. Section 4.3 presents the correctness proofs of the options. Section 4.4 presents our evaluation and discusses the implications of the results.

4.1 Example

We motivate RTS for distributed software histories through an example session using Git, a popular distributed version control system (DVCS).

Distributed software histories: Figure 4.1a visualizes a software history obtained by performing the sequence of Git commands from Figure 4.1c. First, we initialize the software history[1], add two files and make a commit n_1 with these files (lines 1-4). Figure 4.1b shows the abstract representation of the committed files `C` and `T`; file `C` ("*Code*") defines three methods `m`, `p`, and `q` that are checked by four tests $\texttt{t}_1$, $\texttt{t}_2$, $\texttt{t}_3$, and $\texttt{t}_4$ defined in file `T` ("*Test*"). Second, we create a new branch b_1 (line 5), and make and commit changes to methods `m` (lines 6–7) and `p` (lines 8–9). Third, we create another branch b_2 (lines 10–11) and perform a similar sequence of commands as on the first branch (lines 12–15). Finally, we switch to the *master* branch (line 16) and perform a similar sequence of commands (lines 17–20). Although the sequence of commands is similar for each branch, we assume non-conflicting changes on different branches.

[1]`git init` creates the initial node not shown in Figure 4.1a.



(a) Example software history

		Code	Methods in C		
			m	p	q
Tests in T	t_1	{m();}	✓	✗	✗
	t_2	{p();}	✗	✓	✗
	t_3	{q();}	✗	✗	✓
	t_4	{m();p();}	✓	✓	✗

(b) Methods and tests in C and T

```
1  git init // initialize the repository
2  git add C // add C to the repository
3  git add T // add T to the repository
4  git commit -m 'C and T' // commit n1
5  git checkout -b b1 // create branch b1
6  δ1(m) // modify method m in branch b1
7  git commit -am 'Modified m' // commit n2
8  δ2(p) // modify method p in branch b1
9  git commit -am 'Modified p' // commit n3
10 git checkout master // go to master branch
11 git checkout -b b2 // create branch b2
12 δ3(m) // modify method m in branch b2
13 git commit -am 'Modified m' // commit n4
14 δ4(p) // modify method p in branch b2
15 git commit -am 'Modified p' // commit n5
16 git checkout master // go to master branch
17 δ5(p) // modify method p in master branch
18 git commit -am 'Modified p' // commit n6
19 δ6(q) // modify method q in master branch
20 git commit -am 'Modified q' // commit n7
```

(c) Sequence of commands that creates the history on the left

Figure 4.1: Example of a distributed software history

Figure 4.1b further shows which test executes which method; we assume that we have available such a dependency matrix for every revision in the software history. When a method changes, the tests that executed that method are called *modification-traversing* tests. We focus on modifications at a method level for simplicity of presentation; one can track dependencies of other program elements as well [157]. In fact, our implementation tracks dependencies on files, as presented in Chapter 3.

Traditional test selection: Traditional test selection takes as input an old and new revision (together with their test suites), and a dependency matrix for the old revision, and returns a set of tests from the new revision such that each test in the set either is new or traverses at least one of the changes made between the old and the new revision. We have illustrated RTS between two code revisions in Section 3.1, and we formally define it in Section 4.2. Tests that traverse a change can be found from the dependency matrix by taking all the tests that have a checkmark ('✓') for any changed method (corresponding to the appropriate column in Figure 4.1b).

In our running example, all tests are new at n_1, thus all tests are selected. Figure 4.1a

indicates above each node the set of selected tests. At revision n_2, after modifying method m, test selection would take as input n_1 and n_2, and would return all tests that traverse the changed method. Based on our dependency matrix (Figure 4.1b), we can identify that tests t_1 and t_4 should be selected. Following the same reasoning, we can obtain a set of selected tests for each revision in the graph. For simplicity of exposition, we assume that the dependency matrix remains the same for all the revisions. However, the matrix may change if a modification of any method leads to modification in the call graph. In case of a change, the matrix would be recomputed; however, note that for each test that is *not* selected (because it does not execute any changed method), the row in the dependency matrix would *not* change.

Test selection for distributed software histories: Test selection for distributed software histories has not been studied previously, to the best of our knowledge. We illustrate what the traditional test selection would select when a software history (Figure 4.1a) is extended by executing some of the commands available in DVCSs. Specifically, we show that a naive application of the traditional test selection leads to safe but imprecise results (i.e., selects too much), or requires several runs of traditional test-selection techniques, which introduces additional overhead and therefore reduces the benefits of test selection. We consider four commands: *merge*, *rebase*, *cherry-pick*, and *revert*.

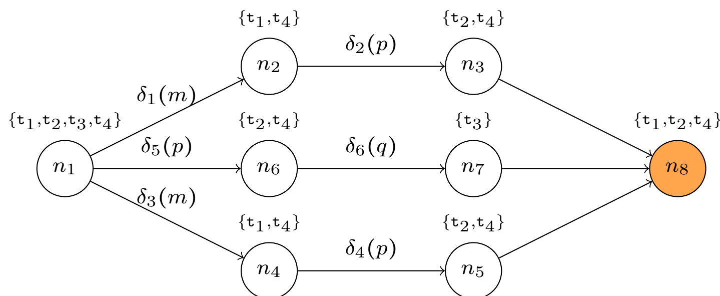


Figure 4.2: Extension of a software history (Figure 4.1) with `git merge b1 b2`

Command: Merge. The merge command joins two or more development branches together. A merge without conflicts and any additional edits is called *auto-merge* and is the most common case in practice. Auto-merge has a property that the changes between the merge point and its parents are a subset of the changes between the parents and the

lowest common ancestors [40, 59] of the parents; we exploit this property in our technique and discuss it further in Section 4.2. If we execute `git merge` $\mathtt{b}_1$ $\mathtt{b}_2$ after the sequence shown in Figure 4.1c, while we are still on the *master* branch, we will merge branches b_1 and b_2 into a new revision n_8 on the *master* branch; this revision n_8 will have three parents: n_3, n_5, and n_7. Figure 4.2 visualizes the software history after the example merge command. The question is what tests to select to run at revision n_8.

We propose three options (and Section 4.4 summarizes how to automatically choose between these options). First, we can use traditional test selection between the new revision (n_8) and its immediate dominator (n_1) [11]. In our example, the changes between these two revisions modify all the methods, so test selection would select all four tests. The advantage of this option is that it runs traditional test selection only once, but there can be many changes, and therefore many tests are selected. Second, we can run the traditional test selection between the new revision and each of its parents and take the intersection of the selected tests. In our example, we would run the traditional test selection between the following pairs: (n_3, n_8), (n_5, n_8), (n_7, n_8); the results for each pair would be: $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{t}_4\}$, $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{t}_4\}$, and $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_4\}$, respectively. The intersection of these sets gives the final result: $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_4\}$. The intuition is that the tests not in the intersection ($\{\mathtt{t}_3\}$) need not be run because their result for the new revision (n_8) can be copied from at least one parent (from n_7 in this case). Although the second option selects fewer tests, it requires running traditional test selection three times, which can lead to substantial overhead. Third, we can collect tests that were modification-traversing on at least two branches (from the branching revision at n_1 to the parents that get merged). In our example, we would select $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_4\}$. As opposed to the two previous options, this third option requires *zero runs* of the traditional test-selection techniques. However, this option is only safe for *auto merge* and requires that the test selection results be stored for previous revisions.

Command: Rebase. Rebase is an alternative way to integrate changes from one branch into another. If we execute `git rebase` $\mathtt{b}_1$ after the sequence shown in Figure 4.1c, we will rewind changes done on *master* branch (δ_5 and δ_6), replay changes from branch b_1 onto *master* (δ_1 and δ_2), and replay the changes from *master* (δ_5 and δ_6). Figure 4.3 visualizes the software history after the example command is executed. Note that the resulting revision (n_9)

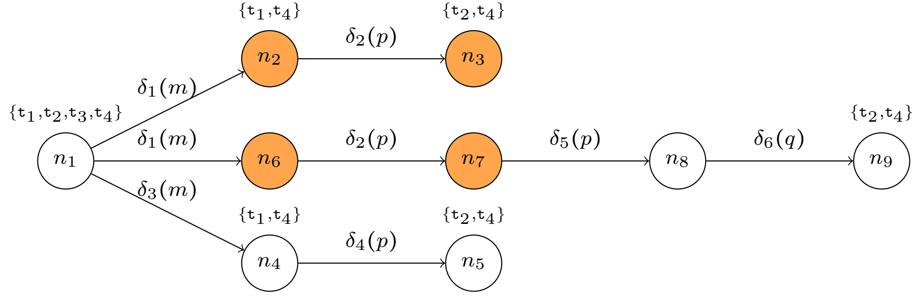


Figure 4.3: Extension of a software history (Figure 4.1) with `git rebase b`$_1$

is the same as if a merge command was executed on *master* branch (`git merge master b`$_1$). The question is the same as for the merge command, what should be selected at n_9.

We propose three options, which are equivalent to the options for the merge command of two branches. First, we can use traditional test selection between the immediate dominator of the new revision and the latest revision on the branch being rebased (n_1), and the new revision (n_9). In our example, the changes between these two revisions modify all the methods, so test selection would select all four tests. Second, we can use traditional test selection between the new revision (n_9) and the latest revisions on branches to/from which we are rebasing (n_3 and n_9) and take the intersection of the selected tests. In our example, we would run the traditional test selection between the following pairs: (n_3, n_9), (n_7, n_9); the results for each pair would be: $\{t_2, t_3, t_4\}$ and $\{t_1, t_2, t_4\}$, respectively. The intersection of these sets gives the final result: $\{t_2, t_4\}$. Note that we use revisions n_3 and n_7 that are available before rebase rewrites the software history on the *master* branch. Third, we can collect tests that were modification-traversing on both branches that are used in rebase. As for the second option, we use the software history before it gets overridden. In our example, we would select $\{t_2, t_4\}$. As for the merge command, the third option works only if there are no conflicts during rebase.

Command: Cherry-pick. Cherry-pick copies the changes introduced by some existing commit, typically from one branch to another branch. If we execute `git cherry-pick n`$_2$ after the sequence shown in Figure 4.1c, we will apply changes (δ_1) made between revisions n_1 and n_2 on top of revision n_7 in *master* branch; the *master* branch will be extended with a new revision n_8. Figure 4.4 visualizes the software history after the command mentioned

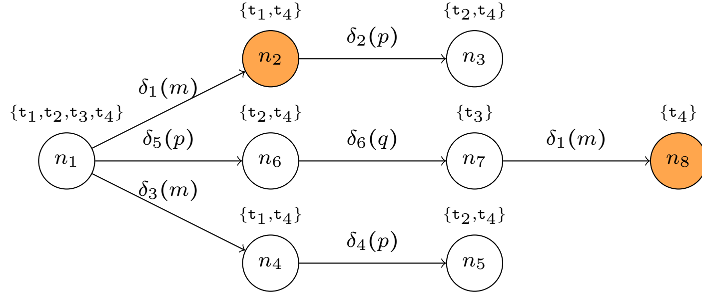


Figure 4.4: Extension of a software history (Figure 4.1) with `git cherry-pick` $\texttt{n}_2$

above. The question is the same as for the merge command, what should be selected at n_8. Naively applying the traditional test selection on revisions n_7 and n_8 would select the same tests as at revision n_2, i.e., $\{\texttt{t}_1, \texttt{t}_4\}$. However, test $\texttt{t}_1$ does not need to be selected at n_8, as this test is not affected by changes on the *master* branch (on which the cherry-picked commit is applied). Therefore, the outcome of $\texttt{t}_1$ at n_8 will be the same as at n_2.

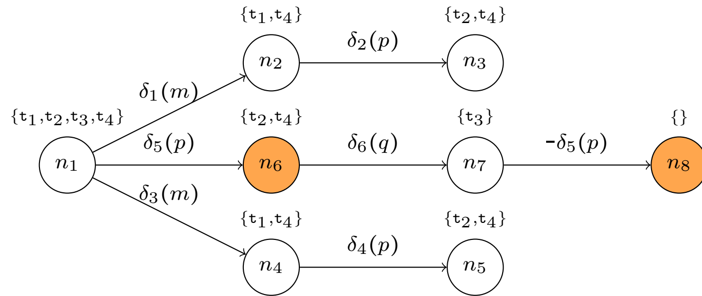


Figure 4.5: Extension of a software history (Figure 4.1) with `git revert` $\texttt{n}_6$

Command: Revert. This command reverts some existing commits. If we execute `git revert` $\texttt{n}_6$ after the sequence shown in Figure 4.1c, we will revert changes made between revisions n_1 and n_6. Figure 4.5 visualizes the software history after the example command is executed. To visualize a change that is reverting a prior change, we use the – sign and the same symbol as for the change being reverted. The *master* branch will be extended with a new revision n_8. Naively applying traditional test-selection techniques between revisions n_7 and n_8 would select the same set of tests as at revision n_6. Instead, if we consider the revert command being executed and changes being made, we can reuse the results of a test from revision n_1 as long as the test is not modification-traversing for any other change after the revision being reverted (n_6). In our example, we can see that the result of all tests obtained

at n_1 can be reused at n_8, and therefore no test has to be selected.

To conclude, naively applying traditional test selection may lead to imprecise results and/or spend too much time on analysis. We believe that our technique, which reasons about the history and commands being executed, leads to a good balance between reduction (in terms of the number of tests being executed) and time spent on analysis.

4.2 Test Selection Technique

4.2.1 Modeling Distributed Software Histories

We model a distributed software history as a directed acyclic graph $G = \langle N, E \rangle$ with a unique root $n_0 \in N$ corresponding to the initial revision. Each node $n \in N$ corresponds to a revision, and each edge corresponds to the parent-child relation among revisions. Each node is created by applying one of the DVCS commands to a set of parent nodes; we assume the command is known. (While the command that creates a node is definitely known at the point of creation, it is not usually kept in the DVCS and cannot always be uniquely determined from the history.) The functions $\mathsf{pred}(n) = \{n' \in N \mid \langle n', n \rangle \in E\}$ and $\mathsf{succ}(n) = \{n' \in N \mid \langle n, n' \rangle \in E\}$ denote the set of parents and children of revision n, respectively. We write $n \leq n'$ if there exists a directed path from n to n' or the two nodes are the same. We write $n \leq^* n'$ to denote the set of all nodes between revisions n and n': $n \leq^* n' = \{n'' \mid n \leq n'' \text{ and } n'' \leq n'\}$. Similarly, we write $n \leq^e n'$ to denote the set of all edges between revisions n and n': $n \leq^e n' = \{\langle n'', n''' \rangle \in E \mid n'', n''' \in n \leq^* n'\}$. The function $\mathsf{sdom}(n) = \{n' \mid n_0 \leq^e n' \cup n' \leq^e n = n_0 \leq^e n \text{ and } n \neq n'\}$ denotes the set of nodes that strictly dominate n. For $n \neq n_0$, the function $\mathsf{imd}(n)$ denotes the unique immediate dominator [11] of n, i.e., $\mathsf{imd}(n) = n'$ such that $n' \in \mathsf{sdom}(n)$ and $\nexists n'' \in \mathsf{sdom}(n)$ such that $n' \in \mathsf{sdom}(n'')$. The function $\mathsf{dom}(n, n')$ denotes the lowest common dominator of n and n', i.e., for a revision n'' such that $\mathsf{pred}(n'') \supseteq \{n, n'\}$, $\mathsf{dom}(n, n') = \mathsf{imd}(n'')$. The function $\mathsf{lca}(n, n')$ denotes the lowest common ancestors [40, 53, 59] (also known as "merge-bases" or "best common ancestors" in Git terminology [73, 96]) for two revisions, i.e., $\mathsf{lca}(n, n') = \{n'' \mid n'' \leq n$ and $n'' \leq n'$ and $\nexists n''' \neq n''$ such that $n''' \leq n$ and $n''' \leq n'$ and $n'' \leq n'''\}$. (We illustrate the

difference between lca and dom in Section 4.3.) The following property holds for all nodes:

$$\mathsf{dom}(n, n') \preceq \mathsf{lca}(n, n') \tag{4.1}$$

4.2.2 Test Selection for Two Revisions

We formalize test selection following earlier work in the area [136, 157] and also model changes and modification-traversing tests used later in our technique. This section focuses on test selection between *two* software revisions. The next sections present our technique for distributed software histories.

Let G be a distributed software history. For a revision n, let $\mathbb{A}(n)$ denote the set of tests *available* at the revision n. Let n and n' be two revisions such that $n \preceq n'$. A *test selection* technique takes as input the revisions n and n' and returns a subset $\mathbb{S}_{sel}(n, n')$ of $\mathbb{A}(n')$. Note that new tests, i.e, $\mathbb{A}(n') \smallsetminus \mathbb{A}(n)$ are always in $\mathbb{S}_{sel}(n, n')$. A test-selection technique is *safe* [135] if every test in $\mathbb{A}(n') \smallsetminus \mathbb{S}_{sel}(n, n')$ has the same outcome when run on the revisions n and n'.

A trivially safe test-selection technique returns $\mathbb{A}(n')$. However, we are interested in selection techniques that select as small a subset as possible. One way to obtain a minimal set is to run each test in $\mathbb{A}(n')$ on the two revisions and keep those that have different outcomes. However, the purpose of the test selection technique is to be more efficient than running all tests. A compromise between minimality and efficiency is provided by the notion of *modification-traversing* tests [136], which syntactically over-approximate the set of tests that may have a different outcome.

Let $\partial(n, n')$ be the set of static code changes between revisions n and n' (which need not be parent-child revisions). Various techniques compute these changes at various levels of granularity (e.g., basic blocks, statements, methods, or other program elements). By extension, we denote the set of changes on all edges from n to n' as

$$\partial^{\star}(n, n') = \bigcup_{\langle n'', n''' \rangle \in n \preceq^{e} n'} \partial(n'', n''')$$

We use the following property:

$$\partial(n, n') \subseteq \partial^\star(n, n') \tag{4.2}$$

It is not an equality because some changes can be reverted on a path from n to n', e.g., consider a graph with three (consecutive) revisions n_1, n_2, and n_3, where all the changes between n_1 and n_2 are reverted between n_2 and n_3: the code at n_3 is exactly the same as the code at n_1, and therefore $\partial(n_1, n_3) = \{\}$.

A test is called *modification-traversing* if its execution on n executes any code element that is modified in n'. For example, in Chapter 3, we introduced a technique where a test is modification-traversing if it depends on at least one file that is modified between n and n'. (Note that "modified" includes all the cases where the existing elements from n are *changed or removed* in n' or where new elements are *added* in n'.) We define a predicate $\varsigma(t, \partial)$ that holds if the test t is modification-traversing for any change in the given set of changes ∂. The predicate can be computed by tracking code paths during a test run and intersecting covered program elements with a syntactic difference between the two revisions. We define a function $\mathsf{mt}(\mathbb{T}, \partial) = \{t \in \mathbb{T} \mid \varsigma(t, \partial)\}$ that returns every test from the set of tests $\mathbb{T}$ that is modification-traversing for any change in the set of changes ∂. Two properties that we will need later are that mt distributes over changes:

$$\mathsf{mt}(\mathbb{T}, \partial_1 \cup \partial_2) = \mathsf{mt}(\mathbb{T}, \partial_1) \cup \mathsf{mt}(\mathbb{T}, \partial_2) \tag{4.3}$$

and thus mt is monotonic with respect to the set of changes:

$$\partial \subseteq \partial' \text{ implies } \mathsf{mt}(\mathbb{T}, \partial) \subseteq \mathsf{mt}(\mathbb{T}, \partial') \tag{4.4}$$

Traditional test selection selects all modification-traversing tests from the old revision that remain in the new revision and the new tests from the new revision:

$$\mathsf{tts}(n, n') = \mathsf{mt}(\mathbb{A}(n) \cap \mathbb{A}(n'), \partial(n, n')) \;\cup\; (\mathbb{A}(n') \setminus \mathbb{A}(n)) \tag{4.5}$$

As $\mathsf{pred}(n')$ is often a singleton $\{n\}$, we also write $\mathsf{tts}(\{n\}, n') = \mathsf{tts}(n, n')$.

4.2.3 Test Selection for Distributed Software Histories

Our technique for test selection takes as inputs (1) the software history $G = \langle N, E \rangle$ optionally annotated with tests selected at each revision, (2) a specific revision $h \in N$ that represents the latest revision (which is usually called `HEAD` in DVCS), and (3) optionally the DVCS command used to create the revision h. It produces as output a set of selected tests $\mathbb{S}_{sel}(h)$ at the given software revision. (We assume that the output of our technique is intersected with the set of available tests at `HEAD`.) We define our technique and prove (in Section 4.3) that it guarantees safe test selection.

Command: Commit: The h revision has one parent, and the changes between the parent and h can be arbitrary, with no special knowledge of how they were created. The set of selected tests can be computed by applying the traditional test selection between the h revision and its parent:

$$\mathbb{S}_{commit}(h) = \mathsf{tts}(\mathsf{pred}(h), h) \tag{4.6}$$

Command: Merge: *Merge* joins two or more revisions and extends the history with a new revision that becomes h. We propose two general options to compute the set of selected tests at h: the first is fast but possibly imprecise, and the second is slower but more precise.

Option 1: This option performs the traditional test selection between the immediate dominator of h and h itself:

$$\mathbb{S}^1_{merge}(h) = \mathsf{tts}(\mathsf{imd}(h), h) \tag{4.7}$$

This option is fast: it computes only one traditional test selection, even if the merge has many parents. However, the number of modifications between the two revisions being compared can be large, leading to many tests being selected unnecessarily. Our empirical evaluation in Section 4.4 shows that this option indeed selects too many tests, discouraging the straightforward use of this option.

Option 2: This option performs one traditional test selection between each parent of the merged revision and the merged revision h itself, and then intersects the resulting sets:

$$\mathbb{S}^k_{merge}(h) = \bigcap_{n \in \mathsf{pred}(h)} \mathsf{tts}(n, h) \tag{4.8}$$

This option can be more precise, selecting substantially fewer tests. However, it has to run k traditional RTS analyses for k parents. Note that we could go to the extreme and, for a given software history with nodes N (whose number is $|N|$), define $\mathbb{S}^{|N|}_{merge}$ that performs one traditional RTS analysis between each revision in the software history and the merge revision, and then takes their intersection: $\mathbb{S}^{|N|}_{merge}(h) = \cap_{n \in N} \mathsf{tts}(n, h)$. This would be the most precise option for the given history and the given traditional RTS, but it would require $|N|$ traditional RTS analyses.

Theorem 1. *$\mathbb{S}^1_{merge}(h)$ and $\mathbb{S}^k_{merge}(h)$ are safe for every merge revision h.*

We prove this Theorem in Section 4.3.

Note that $\mathbb{S}^1_{merge}(h)$ and $\mathbb{S}^k_{merge}(h)$ are incomparable in terms of precision; in general one or the other could be smaller, but in practice $\mathbb{S}^k_{merge}(h)$ is almost always much better (Section 4.4). A contrived example (Figure 4.6) where $\mathbb{S}^1_{merge}(h)$ is smaller is this: starting from a node n_1, branch into n_2 (that changes some method m to m') and n_3 (that changes the same m to m''), and then merge n_2 and n_3 into n_4 such that m in n_4 is the same as m in n_1 (note that such a merge requires manually resolving the conflict of different changes in m' and m''); we have $\partial(n_1, n_4) = \{\}$ while $\partial(n_2, n_4) \cap \partial(n_3, n_4) = \{m\}$, and thus $\mathbb{S}^k_{merge}(n_4)$ would select all tests that depend on m, whereas $\mathbb{S}^1_{merge}(n_4)$ would not select any test.

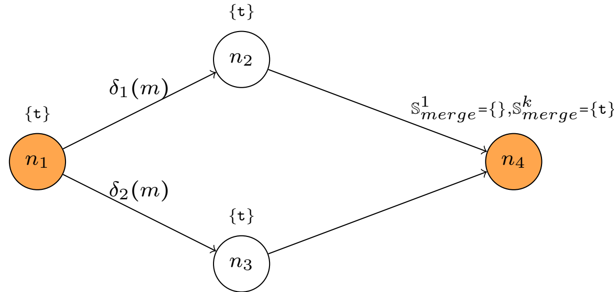


Figure 4.6: $\mathbb{S}^k_{merge}$ may select more tests than $\mathbb{S}^1_{merge}$; $n_1 = n_4$ and $\delta_1(m) \neq \delta_2(m)$

Command: Automerge: A common special case of *merge* is *auto merge*, where revisions are merged automatically without any manual changes to resolve conflicts. (Using the existing DVCS commands can quickly check if a merge is an auto merge.) Empirically (see Table 4.1), auto merge is very common: on average over 90% of revisions with more than one parent are auto merges.

The key property of auto merge is that the merged code revision has a union of all code changes from all branches but has only those changes (i.e., no other manual changes). Formally, given k parents $p_1, p_2, \dots p_k$ that get merged into a new revision h, the changes from each parent p to the merged revision h reflect the changes on all the branches for different parents:

$$\partial(p,h) = \bigcup_{p' \in \mathsf{pred}(h), p' \neq p} \; \bigcup_{l \in \mathsf{lca}(p,p')} \partial(l,p') \tag{4.9}$$

The formula uses lca because of the way Git auto merges branches [73, 96].

For auto merge, we give a test-selection technique, $\mathbb{S}^0_{merge}$, that is based entirely on the software history up to the parents being merged and does not require running any traditional test selection between pairs of code revisions at the point of merge (although it assumes that test selection was performed on the revisions up to the parents of the merge). The set of selected tests consists of the (1) existing tests (from the lowest common dominator of two (different) parents of h) affected by changes on at least two different branches being merged (because the interplay of the changes from various branches can flip the test outcome):

$$\mathbb{S}_{aff}(h) = \bigcup_{p,p' \in \mathsf{pred}(h), p \neq p', d = \mathsf{dom}(p,p')} \Big(\bigcup_{n \in d \leq^* p \setminus \{d\}} \mathbb{S}_{sel}(n) \Big) \cap \Big(\bigcup_{n \in d \leq^* p' \setminus \{d\}} \mathbb{S}_{sel}(n) \Big) \tag{4.10}$$

and (2) new tests available at the merge point but not available on all branches:

$$\mathbb{S}_{new}(h) = \mathbb{A}(h) \setminus \bigcap_{p'' \in \mathsf{pred}(h)} \mathbb{A}(p'') \tag{4.11}$$

Finally, $\mathbb{S}^0_{merge}(h) = \mathbb{S}_{aff}(h) \cup \mathbb{S}_{new}(h)$. The advantage of this option is that it runs *zero* traditional test selections. One disadvantage is that it could select more tests than $\mathbb{S}^k_{merge}$.

Another disadvantage is that it requires storing tests selected at each revision.

Theorem 2. $\mathbb{S}^0_{merge}(h)$ *is safe for every* auto *merge revision* h*.*

Intuitively, $\mathbb{S}^0_{merge}$ is safe because a test that is affected on only one branch need not be rerun at the merge point: it has the same result at that point as on that one branch. The proof is in Section 4.3.

Command: Rebase: *Rebase* replays the changes from one branch into another and then reapplies the local changes; the latest reapplied local change becomes h. We denote the latest revisions on the branch from/to which we rebase as n^{from}_{rebase} and n^{to}_{rebase}, respectively. We propose two options to compute the set of selected tests at h. These options are based on the observation that the merge and rebase commands are used to achieve the same goal (but produce different shapes of the resulting software history). Note that all the rebase options, introduced below, perform as we had done a merge first, compute the set of selected tests, and then perform rebase that changes the shape of the software history. Specifically, we denote h_{merge} ($\equiv h$) to be the result of a merge command on two branches that are used in the rebase command. The partial evaluation of $\mathbb{S}^1_{merge}$ and $\mathbb{S}^k_{merge}$, when the number of branches is equal to two, results in the following options:

$$\mathbb{S}^1_{rebase}(h) = \mathsf{tts}(\mathsf{imd}(h_{merge}), h_{merge}) \tag{4.12}$$

$$\mathbb{S}^2_{rebase}(h) = \mathsf{tts}(n^{to}_{rebase}, h_{merge}) \cap \mathsf{tts}(n^{from}_{rebase}, h_{merge}) \tag{4.13}$$

Similarly, we can define an option for a special case when *rebase* is *auto rebase*:

$$\begin{aligned}\mathbb{S}^0_{rebase}(h_{merge}) = (&\bigcup_{n \in d \leq^* n^{from}_{rebase} \setminus \{d\}} \mathbb{S}_{sel}(n) \cap \bigcup_{n \in d \leq^* n^{to}_{rebase} \setminus \{d\}} \mathbb{S}_{sel}(n)) \\ &\cup (\mathbb{A}(h_{merge}) \setminus \bigcap_{p \in \{n^{from}_{rebase}, n^{to}_{rebase}\}} \mathbb{A}(p))\end{aligned} \tag{4.14}$$

Command: Cherry-pick: *Cherry-pick* reapplies the changes that were performed between a commit n_{cp} and one of its parents $n'_{\mathsf{cp}} \in \mathsf{pred}(n_{\mathsf{cp}})$ (the parent can be implicit for

non-merge $n_{\texttt{cp}}$), and extends the software history (on the branch where the command is applied) with a new revision h. We propose two options to determine the set of selected tests at h. The first option uses the general selection for a commit (the traditional test selection between the current node and its parent): $\mathbb{S}^1_{cherry}(h) = \mathsf{tts}(\mathsf{pred}(h), h)$.

The second option, called $\mathbb{S}^0_{cherry}$, does not require running traditional test selection, but is safe only for *auto cherry-pick*. This option selects all tests that satisfy one of the following four conditions: (1) tests selected between $n'_{\texttt{cp}}$ and $n_{\texttt{cp}}$, and also selected between the point p at which cherry-pick is applied ($\{p\} = \mathsf{pred}(h)$) and $d = \mathsf{dom}(p, n'_{\texttt{cp}})$; (2) tests selected between $n'_{\texttt{cp}}$ and $n_{\texttt{cp}}$, and also selected before $n'_{\texttt{cp}}$ up to d; (3) new tests at $n_{\texttt{cp}}$; and (4) new tests between d and p.

$$\begin{aligned} \mathbb{S}^0_{cherry}(h) = (\mathbb{S}_{sel}(n'_{\texttt{cp}}, n_{\texttt{cp}}) \cap ((\cup_{n \in d \leq^* p \setminus \{d\}} \mathbb{S}_{sel}(n)) \cup (\cup_{n \in d \leq^* n'_{\texttt{cp}} \setminus \{d\}} \mathbb{S}_{sel}(n)))) \\ \cup (\mathbb{A}(n_{\texttt{cp}}) \setminus \mathbb{A}(n'_{\texttt{cp}})) \cup (\mathbb{A}(p) \setminus \mathbb{A}(d)) \end{aligned} \tag{4.15}$$

The intuition for (1) is that the combination of changes that affected tests on both branches, from d to p and from d to $n'_{\texttt{cp}}$, may lead to different test outcomes. The intuition for (2) is that changes before $n_{\texttt{cp}}$ may not exist in the branch on which the cherry-pick is applied and so the outcome of these tests may change. If neither (1) nor (2) holds, the test result can be copied from $n_{\texttt{cp}}$. The formula for cherry pick is similar to that for auto merge but applies to only one commit being cherry picked rather than to an entire branch being merged.

Command: Revert: *Revert* computes inverse changes of some existing commit $n_{\texttt{re}}$ and extends the software history by applying those inverse changes to create a new revision that becomes h. (Reverting a merge creates additional issues that we do not handle specially: one can always run the traditional test selection.) Similar to cherry-pick, we propose two options to determine the set of selected tests. The first option is a naive application of the traditional test selection between h and its parent, i.e., $\mathbb{S}^1_{revert}(h) = \mathsf{tts}(\mathsf{pred}(h), h)$.

The second option, called $\mathbb{S}^0_{revert}$, does not run traditional test selection, but is safe only for *auto revert*. It selects all tests that satisfy one of the following four conditions: (1) tests selected between $n_{\texttt{re}}$ and its parent ($\{p'\} = \mathsf{pred}(n_{\texttt{re}})$), and also selected before the point to which the revert is applied ($\{p\} = \mathsf{pred}(h)$) up to the dominator of p and p' ($d = \mathsf{dom}(p, p')$);

(2) tests selected between $n_{\texttt{re}}$ and its parent p', and also selected before the point that is being reverted (p') up to d; (3) tests that were deleted at the point being reverted (such that in the inverse change tests are added); and (4) new tests between d and p:

$$\begin{aligned} \mathbb{S}^0_{revert}(h) = (\mathbb{S}_{sel}(p', n_{\texttt{re}}) \cap ((\cup_{n \in d \leq^* p \setminus \{d\}} \mathbb{S}_{sel}(n)) \cup (\cup_{n \in d \leq^* p' \setminus \{d\}} \mathbb{S}_{sel}(n)))) \\ \cup (\mathbb{A}(p') \setminus \mathbb{A}(n_{\texttt{re}})) \cup (\mathbb{A}(p) \setminus \mathbb{A}(d)) \end{aligned} \tag{4.16}$$

Intuitively, revert is an inverse of cherry-pick and safe for the same reasons: the unselected tests would have the same outcome at the h revision as at the revision prior to $n_{\texttt{re}}$.

4.3 Proofs of Theorems

Proof. (Theorem 1) $\mathbb{S}^1_{merge}(h)$ is safe whenever a safe traditional test selection is used. This traditional test selection is safe for any pair of revisions and thus is safe for $\mathsf{imd}(h)$ and h that are compared in formula 4.12.

$\mathbb{S}^k_{merge}(h)$ is likewise safe because $\mathsf{tts}(n, h)$ is safe for all nodes n and thus safe for all parents of h. Taking the intersection of selected tests is safe because any test t that is not in the intersection has the same result in revision h as it has for at least one of the parents of that commit, namely the parent(s) whose $\mathsf{tts}(n, h)$ does not contain t. ∎

Proof. (Theorem 2) To prove that $\mathbb{S}^0_{merge}(h)$ is safe, we will establish that $\mathbb{S}^k_{merge}(h) \subseteq \mathbb{S}^0_{merge}(h)$ for any auto merge h.

Let $\mathbb{A} = \mathbb{A}(h)$. We first prove a lemma for the case with no new tests.

Lemma 1. $\mathbb{S}^k_{merge}(h) \subseteq \mathbb{S}^0_{merge}(h)$ *for any auto merge* h *if* $\mathbb{A} = \mathbb{A}(\mathsf{imd}(h))$.

Proof. Consider first the simplest case when a merge has only two parents p and p' that have one lowest common ancestor l. (In general, lowest common ancestor is not unique.) Then formula 4.13 for $\mathbb{S}^k_{merge}$ becomes $\mathsf{mt}(\mathbb{A}, \partial(p, h)) \cap \mathsf{mt}(\mathbb{A}, \partial(p', h))$. Due to the auto merge property 4.9, this can be rewritten as $\mathsf{mt}(\mathbb{A}, \partial(l, p')) \cap \mathsf{mt}(\mathbb{A}, \partial(l, p))$. From formula 4.2 we have that $\partial(l, p) \subseteq \partial^\star(l, p)$ (dually for p'), and from formula 4.1 we further have $\partial^\star(l, p) \subseteq \partial^\star(d, p)$ (dually for p'), where $d = \mathsf{dom}(p, p')$. Due to the mt monotonicity (property

4.4), the latest intersection is a subset of $\mathsf{mt}(\mathbb{A}, \partial^\star(d,p')) \cap \mathsf{mt}(\mathbb{A}, \partial^\star(d,p))$, where $\partial^\star(d,p) = \bigcup_{n' \in d \leq^* p \setminus \{d\}} \bigcup_{n \in \mathsf{pred}(n')} \partial(n,n')$ (and dually for p'). Due to the mt distributivity (property 4.3) (and for cases with one parent, w.l.o.g.), the intersection is $(\cup_{n \in d \leq^* p \setminus \{d\}} \mathsf{mt}(\mathbb{A}, \partial(\mathsf{pred}(n), n)) \cap (\cup_{n \in d \leq^* p' \setminus \{d\}} \mathsf{mt}(\mathbb{A}, \partial(\mathsf{pred}(n), n))$, which is exactly the formula for $\mathbb{S}^0_{merge}$ (Section 4.2.3) when there are no new tests, in which case $\mathbb{S}_{sel}(n) = \mathsf{mt}(\mathbb{A}, \partial(\mathsf{pred}(n), n))$.

Now consider the case where k parents have many lowest common ancestors. We have the following:

$$
\begin{aligned}
\mathbb{S}^k_{merge}(n) &= \bigcap_{p \in \mathsf{pred}(n)} \mathsf{tts}(p,n) && \text{(by 4.13)} \\
&= \bigcap_{p \in \mathsf{pred}(n)} \mathsf{mt}(\mathbb{A}, \partial(p,n)) && \text{(no new tests)} \\
&= \bigcap_{p \in \mathsf{pred}(n)} \mathsf{mt}(\mathbb{A}, \bigcup_{p' \in \mathsf{pred}(n), p' \neq p} \bigcup_{l \in \mathsf{lca}(p,p')} \partial(l,p')) && \text{(by 4.9)} \\
&\subseteq \bigcap_{p \in \mathsf{pred}(n)} \mathsf{mt}(\mathbb{A}, \bigcup_{p' \in \mathsf{pred}(n), p' \neq p, d = \mathsf{dom}(p,p')} \partial(d,p')) && \text{(by 4.1 and 4.4)} \\
&= \bigcap_{p \in \mathsf{pred}(n)} \bigcup_{p' \in \mathsf{pred}(n), p' \neq p, d = \mathsf{dom}(p,p')} \mathsf{mt}(\mathbb{A}, \partial(d,p')) && \text{(by 4.3)} \\
&= \bigcup_{p,p' \in \mathsf{pred}(n), p \neq p', d = \mathsf{dom}(p,p')} \mathsf{mt}(\mathbb{A}, \partial(d,p)) \cap \mathsf{mt}(\mathbb{A}, \partial(d,p')) && \text{(distribute } \cap \text{ over } \cup) \\
&\subseteq \bigcup_{p,p' \in \mathsf{pred}(n), p \neq p', d = \mathsf{dom}(p,p')} \mathsf{mt}(\mathbb{A}, \partial^\star(d,p)) \cap \mathsf{mt}(\mathbb{A}, \partial^\star(d,p')) && \text{(by 4.4)} \\
&= \bigcup_{p,p' \in \mathsf{pred}(n), p \neq p', d = \mathsf{dom}(p,p')} \mathsf{mt}(\mathbb{A}, \cup_{n \in d \leq^* p \setminus \{d\}} \partial(\mathsf{pred}(n), n)) \cap \mathsf{mt}(\mathbb{A}, \cup_{n \in d \leq^* p' \setminus \{d\}} \partial(\mathsf{pred}(n), n)) \\
& && \text{(by def.)} \\
&= \bigcup_{p,p' \in \mathsf{pred}(n), p \neq p', d = \mathsf{dom}(p,p')} (\cup_{n \in d \leq^* p \setminus \{d\}} \mathsf{mt}(\mathbb{A}, \partial(\mathsf{pred}(n), n))) \cap (\cup_{n \in d \leq^* p' \setminus \{d\}} \mathsf{mt}(\mathbb{A}, \partial(\mathsf{pred}(n), n))) \\
& && \text{(by 4.3)} \\
&= \bigcup_{p,p' \in \mathsf{pred}(n), p \neq p', d = \mathsf{dom}(p,p')} (\cup_{n \in d \leq^* p \setminus \{d\}} \mathbb{S}_{sel}(n)) \cap (\cup_{n \in d \leq^* p' \setminus \{d\}} \mathbb{S}_{sel}(n)) && \text{(no new tests)} \\
&= \mathbb{S}_{aff}(h) && \text{(by 4.14)}
\end{aligned}
$$

■

Continuing with the proof for the main theorem, consider now the general case when new tests can be added. We have the following:

$$
\begin{aligned}
\mathbb{S}^k_{merge}(n) &= \bigcap_{p \in \mathsf{pred}(n)} \mathsf{tts}(p, n) && \text{(by 4.13)} \\
&= \bigcap_{p \in \mathsf{pred}(n)} \mathsf{mt}(\mathbb{A}, \partial(p, n)) \cup (\mathbb{A} \setminus \mathbb{A}(p)) && \text{(by 4.5)} \\
&= \bigcup_{s \in 2^{\mathsf{pred}(n)}} \bigcap_{p \in s} \mathsf{mt}(\mathbb{A}, \partial(p, n)) \cap \bigcap_{p' \in \mathsf{pred}(n) \setminus s} \mathbb{A} \setminus \mathbb{A}(p') && \text{(distribute} \cup \text{over} \cap\text{)} \\
&= L1 \cup \bigcup_{s \in 2^{\mathsf{pred}(n)}, s \neq \mathsf{pred}(n)} \bigcap_{p \in s} \mathsf{mt}(\mathbb{A}, \partial(p, n)) \cap \bigcap_{p' \in \mathsf{pred}(n) \setminus s} \mathbb{A} \setminus \mathbb{A}(p') \\
& && \text{(extract one term, } L1 = \big(\textstyle\bigcap_{p \in \mathsf{pred}(n)} \mathsf{mt}(\mathbb{A}, \partial(p, n))\big)\text{)} \\
&\subseteq L1 \cup \bigcup_{s \in 2^{\mathsf{pred}(n)}, s \neq \mathsf{pred}(n)} \bigcap_{p' \in \mathsf{pred}(n) \setminus s} \mathbb{A} \setminus \mathbb{A}(p') && \text{(subset intersection)} \\
&= L1 \cup \bigcup_{s' \in 2^{\mathsf{pred}(n)}, s' \neq \{\}} \bigcap_{p \in s'} \mathbb{A} \setminus \mathbb{A}(p) && \text{(rename complement)} \\
&\subseteq L1 \cup \bigcup_{p \in \mathsf{pred}(n)} \mathbb{A} \setminus \mathbb{A}(p) && \text{(subset intersection)} \\
&= L1 \cup (\mathbb{A} \setminus \bigcap_{p \in \mathsf{pred}(n)} \mathbb{A}(p)) && \text{(De Morgan's law)} \\
&\subseteq \bigcup_{p, p' \in \mathsf{pred}(n), p \neq p', d = \mathsf{dom}(p, p')} (\cup_{n \in d \leq^* p \setminus \{d\}} \mathbb{S}_{sel}(n)) \cap (\cup_{n \in d \leq^* p' \setminus \{d\}} \mathbb{S}_{sel}(n)) \\
&\bigcup (\mathbb{A} \setminus \bigcap_{p \in \mathsf{pred}(n)} \mathbb{A}(p)) && \text{(by Lemma 1)} \\
&= \mathbb{S}^0_{merge}(n) && \text{(by 4.14 and 4.11)}
\end{aligned}
$$

■

Theorem 3. $\mathbb{S}^0_{merge}(h)$ *would be unsafe if using the lowest common ancestors instead of the lowest common dominator:*

$$
\mathbb{S}^{0-\textbf{unsafe}}_{merge}(h) = \bigcup_{p, p' \in \mathsf{pred}(h), p \neq p'} \bigcup_{l, l' \in \mathsf{lca}(p, p')} \Big(\bigcup_{n \in l \leq^* p \setminus \{l\}} \mathbb{S}_{sel}(n)\Big) \cap \Big(\bigcup_{n \in l' \leq^* p' \setminus \{l'\}} \mathbb{S}_{sel}(n)\Big)
\bigcup (\mathbb{A}(h) \setminus \bigcap_{p'' \in \mathsf{pred}(h)} \mathbb{A}(p'')) \tag{4.17}
$$

Note that p and p' must differ, while l and l' may be the same.

Proof. The formula could seemingly be safe because Git performs auto merge based on lca (property 4.9). However, Figure 4.7 shows an example version history where this selection would be unsafe. Suppose that the revision n_1 has a test t that depends on a method m.

This method is changed between revisions n_1 and n_3, and between revisions n_1 and n_4. So the test t would be selected at n_3 and n_4. But this method is *not* changed between revisions n_1 and n_2. At the merge points n_5 and n_6, t would not be selected because it was selected on only one branch each, n_3 and n_4, respectively, rather than on both branches. So far this is safe. However, when merging n_5 and n_6, t would *not* be selected, because $\mathsf{lca}(n_5, n_6) = \{n_2\}$, and the test was not selected within the subgraph n_2, n_5, and n_6. However, this test t should be selected because m was modified on two different paths that reach n_7, and thus these different changes could interplay in such a way that t fails in n_7 even if it passes in both n_5 and n_6.

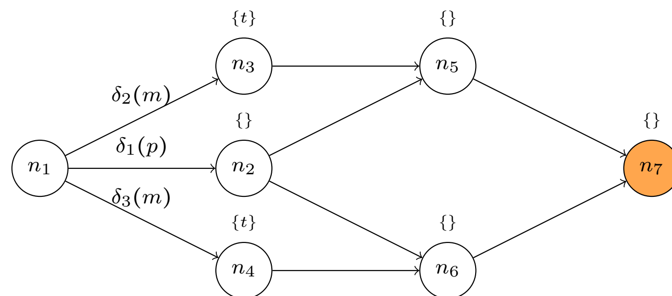


Figure 4.7: Example history to show that using lca (n_2) rather than dom (n_1) is not safe

■

To avoid being unsafe, our actual $\mathbb{S}^0_{merge}(h)$ uses the lowest common dominator rather than the lowest common ancestors.

4.4 Evaluation

We performed several experiments to evaluate the effectiveness of our technique. First, we demonstrate the importance of having a test-selection technique for distributed software histories by analyzing software histories of large open-source projects and reporting statistics about these histories. Second, we evaluate the effectiveness of our test-selection technique by comparing the number of tests selected using $\mathbb{S}^1_{merge}$, $\mathbb{S}^k_{merge}$, and $\mathbb{S}^0_{merge}$ on a number of software histories (both real and systematically generated), i.e., we consider how much

Project	SHA	Size MB	Authors†	(C)ommits	(M)erges	(R)ebases†	Cherrypicks & Reverts (CR)	(M+R+CR) /C	M/C on master	Auto-merges %
Activator	a3bc65e	1.2	14	1499	446	10	29	32.35	93.93	95.73
TimesSquare	d528622	0.31	22	145	50	1	1	35.86	65.71	96.00
Astyanax	ba58831	2.0	59	725	134	3	14	20.82	23.04	94.02
Bootstrap	c75f8a5	3.2	474	6893	1573	21	557	31.20	25.75	83.21
Cucumber	5416686	1.2	145	2495	413	21	148	23.32	15.92	77.48
Dagger	b135011	0.68	32	531	219	1	1	41.61	83.41	100.00
Dropwizard	a01bfd7	1.9	96	1855	238	4	9	13.53	16.23	95.37
EGit	701685b	9.5	70	3574	733	1502	30	63.37	28.32	98.22
Essentials	59b501b	27.0	75	3984	565	69	137	19.35	11.59	95.22
Git	0ecd94d	17.2	1140	35120	7785	10511	1973	57.71	42.62	83.22
GraphHopper	e2805e4	7.2	13	1265	59	3	59	9.56	3.64	55.93
Jenkins	1c0077d	42.6	400	16950	1038	213	327	9.30	4.83	85.93
Jersey	f5a82fa	14.6	28	1320	326	249	12	44.46	35.14	99.07
Jetty	180d9a5	15.5	22	7438	1024	45	999	27.80	6.98	78.62
JGit	7995d87	9.0	83	2801	615	774	24	50.44	33.35	97.48
JUnit	9917b9f	3.2	78	1617	250	47	129	26.34	21.33	83.20
LinuxKernel	e62063d	484.5	11133	400479	27472	151569	–	–	30.12	–
LinuxKVM	b796a09	406.2	8542	273639	17483	107768	–	–	8.92	–
OkHttp	5538ed2	1.1	26	513	212	1	0	41.52	80.16	100.00
TripPlanner	5e7afa5	83.1	60	5168	333	54	131	10.02	5.47	85.88
OrionClient	ffec158	11.9	51	6628	902	218	40	17.50	10.31	93.68
Picasso	29e3461	1.3	33	470	174	11	2	39.78	62.26	98.85
Retrofit	5bd3c1e	0.62	61	631	216	4	2	35.18	58.54	99.07
RxJava	ae073dd	1.7	39	1212	267	3	39	25.49	49.74	89.51
Min	-	0.31	13	145	50	1	0	9.30	3.64	55.93
Max	-	484.5	11133	400479	27472	151569	1973	63.37	93.93	100.00
Median	-	5.20	60.50	2175.00	373.00	19.49	34.50	31.77	27.03	94.62
Ari. mean	-	47.77	945.66	32373.00	2605.29	11379.25	211.95	31.76	34.05	90.25
Geo. mean	-	5.69	79.04	2275.60	415.71	45.60	21.11	18.91	20.49	51.93
Std. Dev.	-	121.71	2717.26	93955.04	6343.27	36281.83	447.38	14.19	26.56	10.30

† We use a heuristic to determine the number of authors and rebases

Table 4.1: Statistics for several projects that use Git

test selection would have saved had it been run on the revisions in the history. Third, we compare $\mathbb{S}^1_{cherry}$ and $\mathbb{S}^0_{cherry}$ on a number of real cherry-pick commits.

We do not evaluate the proposed technique for rebase and revert commands because software histories do not keep track of the commands that created each revision. In particular, we cannot identify precisely whether a revision in the history was created by actually running a special command (such as rebase or revert) or by developers manually editing the code and using a general commit command. In actual practice [144, 146], the developers

Project	**Test [methods]**		**Time [sec]**	
	min	max	min	max
Cucumber (core)	156	308	10	14
GraphHopper (core)	626	692	14	20
JGit	2231	2232	106	116
Retrofit	181	184	10	10

Table 4.2: Statistics for projects used in the evaluation of $\mathbb{S}^1_{merge}$, $\mathbb{S}^k_{merge}$, and $\mathbb{S}^0_{merge}$

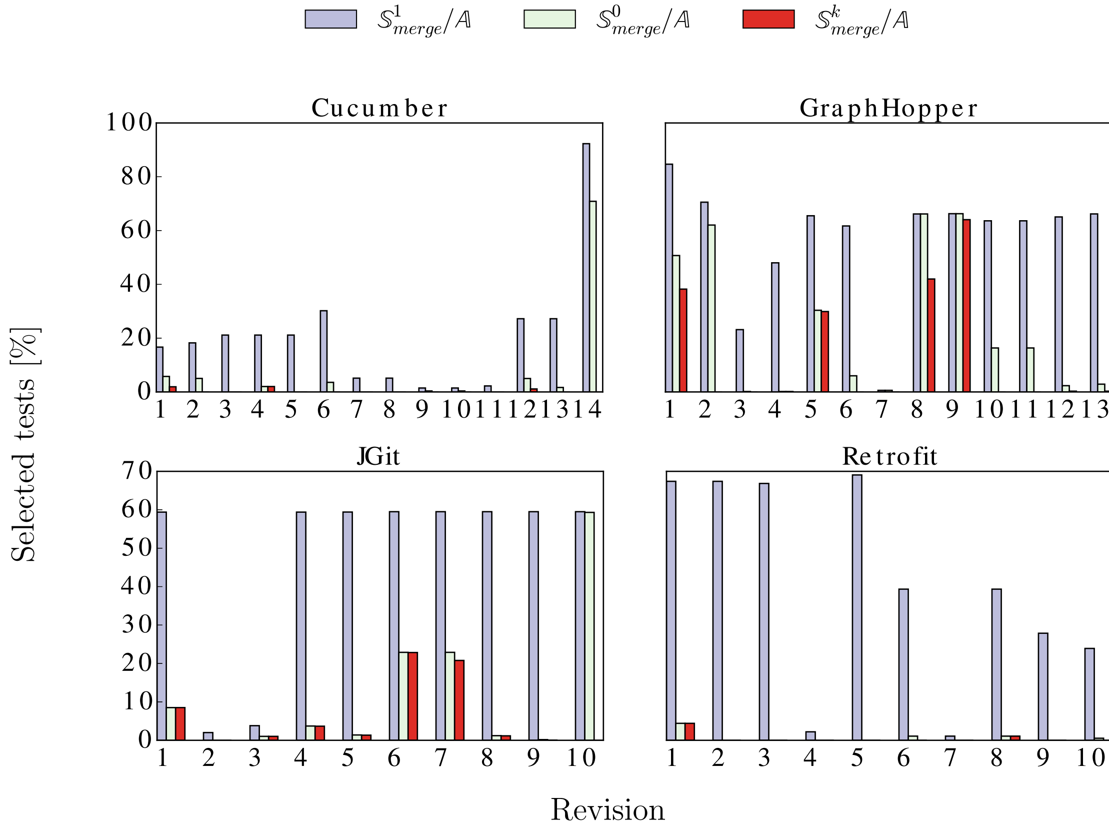


Figure 4.8: Percentage of selected tests for real merges using various options

would use our technique when they create a new software revision and the command being executed is known. Further, note that the proposed technique for rebase command is based on the technique for merge command, so the test selection for these commands should on average have similar savings.

Real software histories are highly non-linear: We collected statistics for software histories of several open-source projects that use Git. To check whether software histories

are non-linear across many project types, we chose projects from different domains (e.g., Cucumber is a tool for running acceptance tests, JGit is a pure Java implementation of the Git version-control system, etc.), implemented in different languages, of various sizes, having different number of unit tests and developers. Table 4.1 shows the collected statistics for 24 projects. The key column is (M+R+CR)/C that shows the ratio of the number of merges, rebases[2], cherry-picks, and reverts over the total number of commits for the entire software history. The ratio can be as high as 63.37% and is 31.76% on average. Stated differently, we may be able to improve test selection for about a third of the commits in an average DVCS history. Additionally, we collected a similar ratio only for the *master* branch, because most development processes run tests for all commits on that branch but not necessarily on other branches (e.g., see the Google process for testing commits [146]). While this ratio included only merges (and not rebases, cherry-picks, or reverts), its average is even higher for the *master* branch than for the entire repository (34.05% vs. 31.76%), which increases the importance of test selection for distributed software histories. Finally, to confirm that the ratio of merges is independent of the DVCS, we collected statistics on three projects that use Mercurial—OpenJDK, Mercurial, and NetBeans—and the average ratio of merges was 20%, which is slightly lower than the average number for Git but still significant.

Implementation: We implemented a tool in Java to perform test selection proposed in Section 4.2. The tool is independent of the DVCS being used and scales to large projects. Because any test-selection technique for distributed histories requires a traditional test selection between two revisions (tts) for linear histories, and because there is no other available tool for the traditional test selection that scales to the large projects used in our study, we used the Ekstazi tool described in Chapter 3 as tts. Note that our technique for distributed histories is orthogonal to the RTS technique used to select tests between two code revisions.

Real merges: Our first set of experiments evaluates our technique on the actual software histories. We used software histories of four large open-source projects (downloaded from GitHub): Cucumber, GraphHopper, JGit, and Retrofit. We selected these projects as their

[2]Note that we approximate the number of rebases by counting commits with different author and committer field.

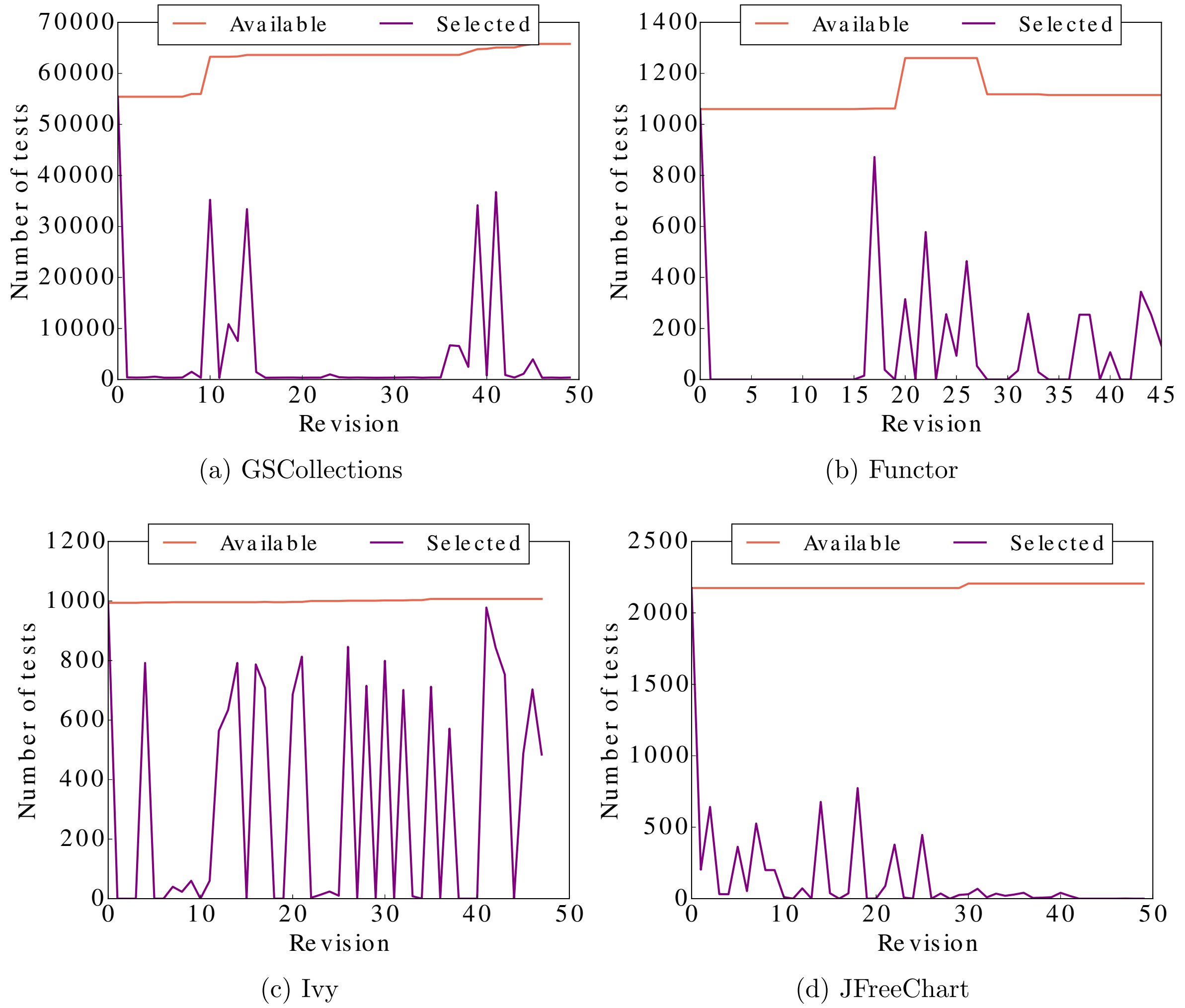
setup was not too complex[3], and they differ in size, number of authors, number of commits, and number of merges. Our experimental setup was the following. For each project, we identify the last merge commit in the current software history and then run our test-selection tool on all the merge commits whose immediate dominator was in the 50 commits before the last merge commit.

At every merge commit, we run all three options—$\mathbb{S}^1_{merge}$, $\mathbb{S}^k_{merge}$, and $\mathbb{S}^0_{merge}$—and compare the number of tests they select. Testing literature [43, 69, 138, 154, 157] commonly measures the speedup of test selection as the ratio of the number of selected tests over the number of available tests ($\mathbb{S}_{sel}/\mathbb{A}$). In addition, Table 4.2 reports the min and max number of available tests across the considered merge commits, and the min and max total time to execute these tests. All tests in these projects are unit tests and take a similar amount of time to execute, so computing the ratio of the numbers of tests is a decent approximation of the ratio of test execution times. We do not measure the real end-to-end time because our implementation of $\mathbb{S}^0_{merge}$ is a prototype that uses a rather unoptimized implementation of Git operations.

Figure 4.8 plots the results for these four projects. In most cases, $\mathbb{S}^k_{merge}$ and $\mathbb{S}^0_{merge}$ achieve substantial saving compared to $\mathbb{S}^1_{merge}$. (Calculated differently, the average speedup of $\mathbb{S}^0_{merge}$ over $\mathbb{S}^1_{merge}$ was 10.89× and $\mathbb{S}^k_{merge}$ over $\mathbb{S}^0_{merge}$ was 2.78×.) Although $\mathbb{S}^0_{merge}$ achieved lower saving than $\mathbb{S}^k_{merge}$ in a few cases (that we discuss below in more detail), it is important to recall that $\mathbb{S}^k_{merge}$ requires k runs of traditional test selection, while $\mathbb{S}^0_{merge}$ requires 0 runs.

We inspected in more detail the cases where $\mathbb{S}^k_{merge}/\mathbb{S}^0_{merge}$ was low. For `GraphHopper` (revisions 2, 10, and 11), two branches have a large number of exactly the same commits (in particular, one branch has 11 commits and another has 10 of those 11 commits, which were created with some cherry-picking); when these branches were merged, the differences between the merged revision and parents were rather small, resulting in a few tests being selected by $\mathbb{S}^k_{merge}$, although the changes between the parents and the dominator were rather big, resulting in many tests being selected by $\mathbb{S}^0_{merge}$. For `JGit` (revision 10) and `Cucumber` (revision 14), some new tests were added on one branch before merging it with another;

[3] We have to build and run tests over a large number of commits, and dependencies in many real projects make running tests from older commits rather non-trivial.



(a) GSCollections

(b) Functor

(c) Ivy

(d) JFreeChart

Figure 4.9: History statistics of projects used for generated software histories

$\mathbb{S}^0_{merge}$ is currently rather conservative in selecting (all) new tests, but new tests are not added frequently in practice.

Based on this inspection, we propose the following heuristic for choosing the best option for test selection at a merge revision:

$\mathbb{S}_{\texttt{merge}}(\texttt{h})$ = **if** (`automerge` & `selection done at every commit`)

if (`many new tests`) $\mathbb{S}^{\texttt{k}}_{\texttt{merge}}(\texttt{h})$ **else** $\mathbb{S}^{0}_{\texttt{merge}}(\texttt{h})$

else if (`short branches`) $\mathbb{S}^{1}_{\texttt{merge}}(\texttt{h})$ **else** $\mathbb{S}^{\texttt{k}}_{\texttt{merge}}(\texttt{h})$

Systematically generated merges: Our second set of experiments systematically compares the merge selection options on a set of graphs generated to represent *potential* software histories. Specifically, for a given number of nodes k, we generate all the graphs where nodes

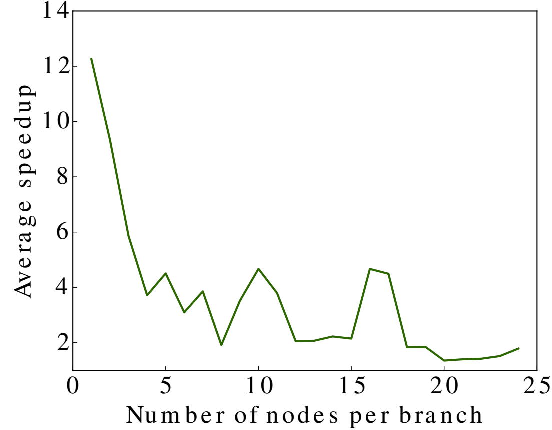


Figure 4.10: $\mathbb{S}^1_{merge}/\mathbb{S}^0_{merge}$ (speedup) for various numbers of commits in each branch

have the out degree (branching) of at most two, each branch contains between 1 and $k/2-2$ nodes, all the branches have the same number of nodes, and there are no linear segments on the master branch (except the last few nodes that remained after generating the branches). In other words, the generated graphs are diamonds of different length. For example for $k = 7$, we have the following two graphs: ·<:>·<:>· and ·<: = :>·–·. The total number of merges for the given number of nodes k is $\lfloor (k-1)/3 \rfloor + \lfloor (k-1)/5 \rfloor + \ldots + \lfloor (k-1)/(k-1) \rfloor$.

In addition to generating history graphs, we need to assign code and tests to each node of the graph. As random code or tests could produce too unrealistic data, we use the following approach: (1) we took the latest 50 revisions of four large open-source projects with *linear* software histories: `JFreeChart` (SVN: 3021), `GSCollections` (Git: 28070efd), `Ivy` (SVN: 1550956), and `Functor` (SVN: 1439120) (Figure 4.9 shows the number of available and selected tests for all projects), (2) we assigned a revision from the linear history to a node of the graph by preserving the relative ordering of revisions such that a linear extension of the generated graph (partial order) matches the given linear history (total order). Using the above formula to calculate the number of merges for generated graph, for 50 revisions, there are 68 merges (in 24 graphs); as we have four projects, the total number of merges is 272.

After the software histories are fully generated, we perform test selection on each of the graphs for each of the projects and collect the number of tests selected by all three options at each merge commit. As for the experiments on real software histories, we calculate the speedup as the ratio of the number of tests. Figure 4.10 shows the average speedup (across

all four projects) for various number of nodes per branch. As expected, with more commits per branch, the speedup decreases, because the sets of changes on each branch become bigger and thus their intersection (as computed by our $\mathbb{S}^0_{merge}$ option) becomes larger. However, the speedup remains high for quite long branches. In fact, this speedup is likely an under-approximation of what can be achieved in real software projects because the assignment of changes across branches may not be representative of actual software histories: many related changes may be sprinkled across branches, which leads to a smaller speedup. Also, linear software histories are known to include more changes per commit [12]. We can see from the comparison of absolute values of the speedups in Figure 4.10 and Figure 4.8 that real software histories have an even higher speedup than our generated histories.

Real cherry-picks: We also compared $\mathbb{S}^1_{cherry}$ and $\mathbb{S}^0_{cherry}$ on 7 cherry-picks identified in the `Retrofit` project. No other revision from the other three projects in our experiments used a cherry-pick command. For 6 cases, $\mathbb{S}^0_{cherry}$ selected 7 tests more than $\mathbb{S}^1_{cherry}$, but all these tests were new. As mentioned earlier, our current technique is rather conservative in selecting new tests; in future, we plan to improve our technique by considering dependency matrices across branches. In the remaining case, $\mathbb{S}^0_{cherry}$ selected 43% fewer tests (42 vs. 73 tests) than $\mathbb{S}^1_{cherry}$.

4.5 Summary

The results show that non-linear revisions are frequent in real software repositories, and that various options we introduced can provide different trade-offs for test selection (e.g., $\mathbb{S}^1_{merge}$, $\mathbb{S}^k_{merge}$, and $\mathbb{S}^0_{merge}$ each have their advantages and disadvantages). Carefully designed combinations of these selection techniques (such as $\mathbb{S}_{merge}$ on page 87 that combines $\mathbb{S}^1_{merge}$, $\mathbb{S}^k_{merge}$, and $\mathbb{S}^0_{merge}$) can provide a substantial speedup for test selection in particular and regression testing in general.

CHAPTER 5

Related Work

This chapter presents an overview of the work related to the contributions of this dissertation. There has been a lot of work on regression testing in general, as surveyed in two reviews [43, 157], and on regression test selection (RTS) in particular, as further surveyed in two more reviews [68, 69]. Those papers review 30+ years of history of research on RTS, but other papers also point out that RTS remains an important research topic for the future [123]. This chapter is organized as follows. Section 5.1 discusses work related to our study of manual RTS. Sections 5.2, 5.3, and 5.4 discuss work that directly inspired our Ekstazi technique between two code revisions, including: recent advances in build systems, RTS based on class dependencies, and RTS for external resources. Section 5.5 presents prior work on collecting code coverage. Section 5.6 describes several regression testing techniques that use different levels of selection granularity and coverage granularity. Section 5.7 presents other related work on regression testing between two project revisions. Section 5.8 discusses prior work on studying and analyzing distributed software histories.

5.1 Manual Regression Test Selection

The closest work to our study of manual RTS are studies of testing practices, studies of usage profiles, and studies of logs recorded in real time.

Our study is different in scope, emphasis, and methodology from the work of Greiler et al. [83], who recently conducted a study of testing practices among developers. We did not limit our scope to a specific class of software, while they focus on testing component-based software. Their emphasis is on answering important questions about the testing practices that are (not) adopted by organizations and discovering reasons *why* these practices are

not adopted. On the other hand, we focus on *how* developers perform RTS. Finally, their approach utilizes interviews and surveys, but we analyzed data collected from developers in real time.

Regarding the empirical study of RTS techniques, the closest work to ours is the use of field study data by Orso et al. [122]. They collected usage profile data from *users* of deployed software for tuning their Gamma approach for RTS and impact analysis. We study data collected from *developers* to gain insight on improving manual RTS. The data was previously used for analyzing whether VCS commit data is imprecise and incomplete when studying software evolution [120], for comparing manual and automated refactorings [118], and for mining fine-grained code change patterns [119]. Although our work is based on previously used data, this is the first use of the data for studying how developers perform testing.

5.2 Build Systems and Memoization

Our Ekstazi regression test selection, based on file dependencies, is related to build systems, in particular to incremental builds and memoization that also utilize file dependencies.

Memoize [115] is used to speed up builds. Memoize is a Python-based system that, given a command, uses `strace` on Linux to monitor all files opened (and the mode in which they are opened, e.g., read/write) while that command executes. Memoize saves all file paths and file checksums, and ignores subsequent runs of the same command if no checksum changed. Fabricate [70] is an improved version of Memoize that also runs on Windows and supports parallel builds. Other build systems, such as Vesta [9] and SCons [8], capture dependencies on files that are attempted to be accessed, even if they do not exist; this is important because the behavior of the build scripts can change when these files are added later. For automatic memoization of Python code, Guo and Engler proposed IncPy [86] that memoizes calls to functions (and static methods). IncPy supports functions that access files, i.e., it stores the file checksums and re-executes a function if any of its inputs or files changes. Our insight is to view RTS as memoization: if none of the dependent files for some test changed, then the test need not be run. By capturing file dependencies for each test entity, Ekstazi provides scalable and efficient RTS that integrates well with testing frameworks. As discussed

throughout the dissertation, EKSTAZI differs from build systems and memoization in several aspects: capturing dependencies for each test entity even when all entities are executed in the same JVM, supporting test entity granularities, smart checksum, and capturing files inside archives.

5.3 Class-based Test Selection

EKSTAZI is related to work on class firewall, work that collects class dependencies, and work that proves safety of RTS and incremental builds.

Hsia et al. [97] were the first to propose RTS based on class firewall [108], i.e., the statically computed set of classes that may be affected by a change. Orso et al. [124] present an RTS technique that combines class firewall and dangerous edges [138]. Their approach works in two phases: it first finds relations between classes and interfaces to identify a subgraph of the Java Interclass Graph that may be affected by the changes, and then selects tests via an edge-level RTS on the identified subgraph. The EKSTAZI approach differs in that it collects all dependencies dynamically, which is more precise than computing them statically.

Skoglund and Runeson first performed a large case study on class firewall [141] and then [142] proposed an improved technique that removes the class firewall and uses a change-based RTS technique that selects only tests that execute modified classes. They give a paper-and-pencil proof that their improved technique is safe under certain assumptions. More recently, Christakis et al. [50] give a machine-verifiable proof that memoization of partial builds is safe when capturing dependencies on all files, under the relaxed assumption that code behaves deterministically (e.g., there is no network access). Compared to prior work, EKSTAZI captures all files (including classes), handles addition and changes of test classes, applies smart checksum, supports reflection, and has both class and method selection granularities. Moreover, we integrated EKSTAZI with JUnit and evaluated it on a much larger set of projects, using the end-to-end testing time. Finally, EKSTAZI includes an approach to improve RTS techniques for software that uses DVCS.

5.4 External Resources

EKSTAZI is also related to prior work on RTS for database programs and work on RTS in the presence of configuration changes.

Haraty et al. [88] and Daou [54] proposed RTS techniques for database programs that work in two phases: they first identify code changes and database component changes using the firewall technique, and then select the tests that traverse these changes. They additionally reduce the selected tests further using two algorithms: one based on control flow and the other based on the firewall technique applied on the inter-procedural level. Willmor and Embury [154] proposed two RTS techniques for database programs, one that captures interaction between a database and the application, and the other based solely on the database state. Kim et al. [106] proposed RTS for ontology-driven systems; the technique creates representations of the old and new ontology and selects tests that are affected by changes. Nanda et al. [117] proposed RTS for applications with configuration files and databases. Compared to prior work, we explored several ways to integrate RTS with the existing testing frameworks, and EKSTAZI captures all files. At the moment EKSTAZI offers no special support for dependencies other than files (e.g., databases and web services). In other words, EKSTAZI treats an entire database as one file: if any record in the database changes, then every test that accesses anything in the database will be selected to be rerun.

5.5 Code Instrumentation

Collecting dependencies in RTS ($\mathcal{C}$ phase) is similar to collecting structural code coverage. In particular, prior work on optimizing code coverage collection is closely related.

Jazz [116] is an approach to reduce the runtime overhead when collecting code coverage; Jazz dynamically adds and removes instructions that collect coverage information. Santelices and Harrold [140] presented DUA-Forensics, a fast approach to compute approximate definition-use associations from branch coverage. Kumar et al. [107] presented analyses to reduce the number of instrumentation points and the cost of code that saves coverage information. Unlike existing code coverage techniques, EKSTAZI collects file dependencies/-

coverage. To reduce the runtime overhead, Ekstazi instruments a small number of carefully selected points (Section 3.2.3). In the future, like Jazz, we plan to consider dynamically removing instrumentation code. It is unclear if this optimization will provide any benefit when multiple test entities are executed in the same process (i.e., in the same Java Virtual Machine), as code has to be instrumented at the beginning of each test entity.

5.6 Granularity Levels

Prior work proposed or evaluated various selection and coverage granularities for several regression testing techniques, including RTS.

Rothermel et al. [133, 134] showed significant impact of test suite granularity, i.e., size of tests in a test suite, on the cost and benefits of several regression testing techniques, including RTS. In our experiments, we do not control for the size of test methods, but we use the test classes that are manually written by the developers of the projects. However, we evaluated Ekstazi with method and class selection granularity (which may correspond to various test suite granularities). Our results show that, although finer granularity may select fewer tests, the coarser granularity provides bigger reduction in end-to-end time.

Bible at al. [41], Elbaum et al. [64], and Di Nardo et al. [56] compared different coverage granularity levels (e.g., statement vs. function) for regression testing techniques. While some results were independent of the levels, the general conclusion is "coarser granularity coverage criteria are more likely to scale to very large systems and should be favoured unless significant benefits can be demonstrated for finer levels" [56]. Ekstazi uses a coarse granularity, i.e., files, for coverage granularity, and the experiments show better results than for `FaultTracer` based on a finer granularity.

Echelon from Microsoft [144] performs test prioritization [157] rather than RTS. It tracks fine-grained dependencies based on basic blocks and accurately computes changes between code revisions by analyzing compiled binaries. Many research RTS techniques [157] also compute fine-grained dependencies like Echelon, but in contrast to Echelon, compare source code of the revisions. Because Echelon is not publicly available, our evaluation used `FaultTracer` [158], a state-of-the-research RTS tool.

Ren et al. [129] described the Chianti approach for change-impact analysis. Chianti collects method dependencies for each test and analyzes differences at the source code level. Chianti reports the tests that are affected by changes and determines for each test the changes that affect the behavior. We show that fine-grained coverage granularity can be expensive, and propose EKSTAZI, a novel RTS technique, which tracks dependencies on coarse-grained dependencies – files.

5.7 Other Work on RTS for Two Revisions

Other work related to EKSTAZI includes work on a framework for comparing RTS techniques, RTS techniques that collect dependencies statically, RTS for other project domains, test suite evolution, and RTS prediction models.

Rothermel and Harrold [137] proposed four metrics for comparing RTS techniques: efficiency, precision, safety, and generality (i.e., applicability of an RTS technique to a broad set of projects). Unlike their work, which defines efficiency in terms of RTS analysis time, we define efficiency in terms of end-to-end time, which is the time observed by developers. We demonstrate EKSTAZI's generality by evaluating it with a large number of projects. Zheng et al. [160] proposed a fully static RTS technique that does not collect dependencies but rather constructs a call graph (for each test) and intersects the graph with the changes. EKSTAZI collects dependencies dynamically, and measures the end-to-end time. Xu and Rountev [155] developed a regression test selection technique for AspectJ programs. Although their approach is more precise than EKSTAZI, they use fine-grained coverage granularity and therefore inherit the high cost of other fine-grained techniques. Pinto et al. [126] and Marinescu et al. [114] studied test-suite evolution and other execution metrics over several project revisions. While we did not use those same projects (e.g., Marinescu et al.'s projects are in C), we used 615 revisions of 32 projects. Several prediction models [90, 132] were proposed to estimate if RTS would be cheaper than RetestAll. Most models assume that the $\mathcal{C}$ phase is run separately and would need to be adjusted when RTS is integrated and the end-to-end time matters. We focus EKSTAZI on the end-to-end time.

5.8 Distributed Version Control Systems

Others [12, 42, 45, 125, 131] analyzed distributed software histories to study commits, described pitfalls of mining distributed histories (e.g., DVCS commands are not recorded), and suggested improvements to DVCS. No prior work related to (regression) testing or verification analyzed or reasoned about distributed software histories. We proposed the first RTS approach that improves precision of any RTS technique for projects with distributed software histories. Our approach for *distributed histories* is compatible with all traditional RTS techniques for *linear histories* as we abstract them in the core `mt` and `tts` functions (Section 4.2.2). We use traditional RTS when a revision is created by a commit command, and we reason about software history, modification-traversing tests, and commands being executed when a revision is created by other DVCS commands (merge, rebase, cherry-pick, and revert). Our results show that our proposed approach improves precision more than an order of magnitude.

CHAPTER 6

Conclusions and Future Work

Regression testing is important for checking that software changes do not break previously working functionality. However, regression testing is costly as it runs many tests for many revisions. Although RTS is a promising approach to speed up regression testing, and was proposed over three decades ago, no RTS technique has been widely adopted in practice, due to efficiency and safety issues. The contributions of this dissertation address these problems.

First, we studied logs recorded in real time from a diverse group of developers to understand the impact of the lack of practical RTS techniques. The study shows that almost all developers perform manual RTS, and they select tests in mostly ad hoc ways (potentially missing bugs or wasting time). Second, we proposed Ekstazi RTS technique, which takes a radically different view from prior RTS techniques: keeping track on coarse-grained dependencies can lead to faster end-to-end time than keeping track on fine-grained dependencies. Ekstazi tracks dependencies on files, guarantees safety in more cases than prior techniques, and provides a substantial reduction in regression testing time. Ekstazi balances the time for the analysis and collection phases, rather than focusing solely on reducing the number of selected tests for the execution phase. Third, we proposed a novel approach that improves precision of any RTS technique for projects with distributed software histories. Unlike any prior RTS technique, the approach takes into account version histories arising out of distributed development, and includes several options that trade off the number of RTS analysis runs and the number of selected tests (which reflects in the execution and collection time).

We now present our plans for possible future work that can build upon our current contributions and results as described in chapters 2, 3, and 4:

Public Release of Experimental Data: We intend to release our dataset and scripts, which are used in our experiments, to allow researchers to reproduce our results.

We have already released some additional information related to our experiments at `http://www.ekstazi.org/research.html`.

Regression Test Selection using File and Method Dependencies: Changes between two project revisions commonly update only method bodies. A test selection technique that uses method coverage granularity would be safe for these revisions. However, such technique would be unsafe for any other revision (e.g., annotation or field updates, or even method additions or deletions). We plan to combine file coverage granularity and method coverage granularity.

Restructuring Test Classes: Several test methods, for the same code under test, are commonly grouped in a single test class. Therefore, it is likely that these test methods have similar dependencies. Indeed, based on our evaluation (Section 3.3.5), using test class granularity leads to better results than using test method granularity, i.e., the overhead for class granularity is smaller, and the number of selected tests is not significantly larger. In the future, we would like to group all test methods that have similar dependencies, even if they reside in different test classes/packages. Also, we would like to split test classes, where several test methods have greatly different dependencies.

Safety of Ekstazi Instrumentation: To collect classes covered by each test method/-class, Ekstazi instruments several places in code (e.g., constructors, static blocks, access to static fields, etc.). We would like to produce a machine-verifiable proof that our instrumentation guarantees safe test selection.

Various Testing Frameworks and JVM-based Languages: JUnit is the most widely used testing framework (for Java), however, it is not the only testing framework available. We plan to support other testing frameworks for Java (e.g., TestNG [147] and randomizedtesting [128]) by inserting Ekstazi hooks at appropriate places (Section 3.2.3). One of the challenges is to identify places where the hooks should be inserted. Note that we have to collect dependencies during the construction of a test entity and during the execution of the test entity.

Other Programming Languages: At the moment, to the best of our knowledge, there is no (publicly available) tool, which collects dynamic dependencies, for RTS for any language. We intend to explore if our technique (based on the file dependencies) is applicable to other languages (e.g., Python). Our initial goal is to build a language-agnostic technique on top of Fabricate [70], which monitors all opened files during the execution of a process. In addition, we plan to integrate the language-agnostic technique with EKSTAZI to improve RTS precision whenever a project uses Java.

Test Prioritization based on File Level Dependencies: Test prioritization is another commonly studied regression testing technique. The goal of test prioritization is to order tests such that if there are bugs in the code under tests, the tests that would fail due to these bugs are executed earlier in the order. Prior work showed that several heuristics can achieve better results than random ordering, e.g., ordering base on the number of times that a test failed in the history or ordering based on the number of statements that a test covers. We intend to explore test ordering based on the number of files that a test covers; files covered by each test can be obtained by EKSTAZI.

Further Analysis of Ekstazi Results: The experiments that we conducted to evaluate EKSTAZI resulted in a large body of data which led to some interesting observations. While we reported the results that are common for RTS studies (e.g., the percentage of selected tests), there are many other results that can be extracted from the collected data. These additional results can help us understand the benefits and limitations of our technique. Specifically, we plan to explore: (1) if long running tests are likely to be selected more often than short running tests and (2) what are similarities in the set of dependencies among test entities.

Impact of Ekstazi on Software Development: We hypothesize that the use of EKSTAZI could impact software development, e.g., developers may start making smaller changes or making their code more modular with the goal to minimize regression testing time. We plan to explore software histories of projects that integrated EKSTAZI and compare software development before and after the integration. Also, we plan to

investigate if the developers' changes could have been done in a different order that would have achieved additional savings in terms of test execution time.

Considering that several open-source projects adopted Ekstazi, we hope that we can enter a new era of software development where projects embrace RTS to speed up their testing. While the Ekstazi techniques are likely to be improved upon, the use of RTS can help developers to improve the quality of their software. We expect to see new, interesting results that improve both theory and practice of regression test selection.

Ekstazi

REFERENCES

[1] *Apache Projects*, https://projects.apache.org.

[2] *Buck*, http://buckbuild.com.

[3] *Build in the cloud*, http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html.

[4] *GitHub*, https://github.com.

[5] *Google Code*, https://code.google.com.

[6] *Java Annotations*, http://docs.oracle.com/javase/7/docs/technotes/guides/language/annotations.html.

[7] *Ninja*, https://martine.github.io/ninja.

[8] *SCons*, http://www.scons.org.

[9] *Vesta*, http://www.vestasys.org.

[10] *Cambridge university study states software bugs cost economy $312 billion per year*, http://www.prweb.com/releases/2013/1/prweb10298185.htm.

[11] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[12] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic, *What's a typical commit? A characterization of open source software repositories*, International Conference on Program Comprehension, 2008, pp. 182–191.

[13] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo, *An automatic approach to identify class evolution discontinuities*, International Workshop on Principles of Software Evolution, 2004, pp. 31–40.

[14] *Apache Ant*, http://svn.apache.org/repos/asf/ant/core/trunk.

[15] *Apache BVal*, https://svn.apache.org/repos/asf/bval/trunk.

[16] *Apache Camel - Building*, http://camel.apache.org/building.html.

[17] *Apache Camel*, https://git-wip-us.apache.org/repos/asf/camel.git.

[18] *Apache Commons Collections*, http://svn.apache.org/repos/asf/commons/proper/collections/trunk.

[19] *Apache Commons Config*, http://svn.apache.org/repos/asf/commons/proper/configuration/trunk.

[20] *Apache Commons DBCP*, http://svn.apache.org/repos/asf/commons/proper/dbcp/trunk.

[21] *Apache Commons IO*, http://svn.apache.org/repos/asf/commons/proper/io/trunk.

[22] *Apache Commons Lang*, http://svn.apache.org/repos/asf/commons/proper/lang/trunk.

[23] *Apache Commons Math*, https://github.com/apache/commons-math.

[24] *Apache Commons Net*, http://svn.apache.org/repos/asf/commons/proper/net/trunk.

[25] *Apache Continuum*, https://svn.apache.org/repos/asf/continuum/trunk.

[26] *Apache CXF*, https://github.com/apache/cxf.

[27] *Apache EmpireDB*, https://svn.apache.org/repos/asf/empire-db/trunk.

[28] *Apache Functor*, http://svn.apache.org/repos/asf/commons/proper/functor/trunk.

[29] *Apache Hadoop*, https://github.com/apache/hadoop-common.

[30] *Apache Ivy*, https://svn.apache.org/repos/asf/ant/ivy/core/trunk.

[31] *Apache JXPath*, http://svn.apache.org/repos/asf/commons/proper/jxpath/trunk.

[32] *Apache Log4j*, http://svn.apache.org/repos/asf/logging/log4j/trunk.

[33] *Apache PDFBox*, http://svn.apache.org/repos/asf/pdfbox/trunk.

[34] *Apache River*, http://svn.apache.org/repos/asf/river/jtsk/trunk.

[35] *Apache Commons Validator*, http://svn.apache.org/repos/asf/commons/proper/net/trunk.

[36] *Apache ZooKeeper*, http://svn.apache.org/repos/asf/zookeeper/trunk.

[37] Thomas Ball, *On the limit of control flow analysis for regression test selection*, International Symposium on Software Testing and Analysis, 1998, pp. 134–142.

[38] Boris Beizer, *Software testing techniques (2nd ed.)*, Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[39] Jonathan Bell and Gail E. Kaiser, *Unit test virtualization with VMVM*, International Conference on Software Engineering, 2014, pp. 550–561.

[40] Michael A. Bender, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin, *Finding least common ancestors in directed acyclic graphs*, Symposium on Discrete Algorithms, 2001, pp. 845–853.

[41] John Bible, Gregg Rothermel, and David S. Rosenblum, *A comparative study of coarse- and fine-grained safe regression test-selection techniques*, Transactions on Software Engineering and Methodology **10** (2001), no. 2, 149–183.

[42] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu, *The promises and perils of mining Git*, International Working Conference on Mining Software Repositories, 2009, pp. 1–10.

[43] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran, *Regression test selection techniques: A survey*, Informatica (Slovenia) **35** (2011), no. 3, 289–321.

[44] Lionel Briand, Yvan Labiche, and Siyuan He, *Automating regression test selection based on UML designs*, Journal of Information and Software Technology **51** (2009), no. 1, 16–30.

[45] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig, *How do centralized and distributed version control systems impact software changes?*, International Conference on Software Engineering, 2014, pp. 322–333.

[46] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen, *Reversible debugging software*, Technical report, 2013, University of Cambridge, Judge Business School.

[47] Lianping Chen, *Continuous delivery: Huge benefits, but challenges too*, Software **32** (2015), no. 2, 50–54.

[48] Pavan Kumar Chittimalli and Mary Jean Harrold, *Re-computing coverage information to assist regression testing*, International Conference on Software Maintenance, 2007, pp. 164–173.

[49] ———, *Recomputing coverage information to assist regression testing*, Transactions on Software Engineering **35** (2009), no. 4, 452–469.

[50] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte, *Formalizing and verifying a modern build language*, International Symposium on Formal Methods, 2014, pp. 643–657.

[51] *CodingTracker*, `http://codingtracker.web.engr.illinois.edu`.

[52] *Cucumber JVM*, `https://github.com/cucumber/cucumber-jvm`.

[53] Artur Czumaj, Mirosaw Kowaluk, and Andrzej Lingas, *Faster algorithms for finding lowest common ancestors in directed acyclic graphs*, Theoretical Computer Science **380** (2007), no. 1-2, 37–46.

[54] Bassel A. Daou, *Regression test selection for database applications*, Advanced Topics in Database Research **3** (2004), 141–165.

[55] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, *Hints on test data selection: Help for the practicing programmer*, Computer **11** (1978), no. 4, 34–41.

[56] Daniel Di Nardo, Nadia Alshahwan, Lionel C. Briand, and Yvan Labiche, *Coverage-based test case prioritisation: An industrial case study*, International Conference on Software Testing, Verification, and Validation, 2013, pp. 302–311.

[57] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson, *Automated detection of refactorings in evolving components*, European Conference on Object-Oriented Programming, 2006, pp. 404–428.

[58] Danny Dig and Ralph Johnson, *How do APIs evolve? A story of refactoring*, Journal of Software Maintenance and Evolution **18** (2006), no. 2, 83–107.

[59] Stefan Eckhardt, Andreas Michael Mühling, and Johannes Nowak, *Fast lowest common ancestor computations in dags*, European Symposium on Algorithms, 2007, pp. 705–716.

[60] *Eclipse*, `http://www.eclipse.org`.

[61] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur, *Framework for testing multi-threaded Java programs*, Concurrency and Computation: Practice and Experience **15** (2003), no. 3-5, 485–499.

[62] *Ekstazi*, `http://www.ekstazi.org`.

[63] Sebastian Elbaum, David Gable, and Gregg Rothermel, *The impact of software evolution on code coverage information*, International Conference on Software Maintenance, 2001, pp. 170–179.

[64] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel, *Test case prioritization: A family of empirical studies*, Transactions on Software Engineering **28** (2002), no. 2, 159–182.

[65] Sebastian Elbaum, Gregg Rothermel, and John Penix, *Techniques for improving regression testing in continuous integration development environments*, International Symposium on Foundations of Software Engineering, 2014, pp. 235–245.

[66] Sebastian G. Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky, *Selecting a cost-effective test case prioritization technique*, Software Quality Journal **12** (2004), no. 3, 185–210.

[67] Emelie Engström and Per Runeson, *A qualitative survey of regression testing practices*, Product-Focused Software Process Improvement, 2010, pp. 3–16.

[68] Emelie Engström, Per Runeson, and Mats Skoglund, *A systematic review on regression test selection techniques*, Journal of Information and Software Technology **52** (2010), no. 1, 14–30.

[69] Emelie Engström, Mats Skoglund, and Per Runeson, *Empirical evaluations of regression test selection techniques: a systematic review*, International Symposium on Empirical Software Engineering and Measurement, 2008, pp. 22–31.

[70] *Fabricate*, `https://code.google.com/p/fabricate`.

[71] Kurt Fischer, Farzad Raji, and Andrew Chruscicki, *A methodology for retesting modified software*, National Telecommunications Conference, 1981, pp. 1–6.

[72] *Use a gated check-in build process to validate changes*, `http://msdn.microsoft.com/en-us/library/dd787631.aspx`.

[73] *git-merge-base*, `https://www.kernel.org/pub/software/scm/git/docs/git-merge-base.html`.

[74] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov, *Ekstazi: Lightweight test selection*, International Conference on Software Engineering, Demo, 2015, pp. 713–716.

[75] ______, *Practical regression test selection with dynamic file dependencies*, International Symposium on Software Testing and Analysis, 2015, to appear.

[76] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov, *Regression test selection for distributed software histories*, International Conference on Computer Aided Verification, 2014, pp. 293–309.

[77] ______, *Regression test selection for distributed software histories*, Technical report, 2014, `https://www.ideals.illinois.edu/handle/2142/49112`.

[78] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov, *An empirical evaluation and comparison of manual and automated test selection*, Automated Software Engineering, 2014, pp. 361–372.

[79] Michael W. Godfrey and Lijie Zou, *Using origin analysis to detect merging and splitting of source code entities*, Transactions on Software Engineering and Methodology **31** (2005), no. 2, 166–181.

[80] *Google Closure Compiler*, `https://github.com/google/closure-compiler`.

[81] *GraphHopper*, `https://github.com/graphhopper/graphhopper`.

[82] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel, *An empirical study of regression test selection techniques*, Transactions on Software Engineering and Methodology **10** (2001), no. 2, 184–208.

[83] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey, *Test confessions: A study of testing practices for plug-in systems*, International Conference on Software Engineering, 2012, pp. 244–254.

[84] *GS Collections*, `https://github.com/goldmansachs/gs-collections`.

[85] *Guava*, `https://github.com/google/guava`.

[86] Philip J. Guo and Dawson Engler, *Using automatic persistent memoization to facilitate data analysis scripting*, International Symposium on Software Testing and Analysis, 2011, pp. 287–297.

[87] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov, *Reliable testing: Detecting state-polluting tests to prevent test dependency*, International Symposium on Software Testing and Analysis, 2015, to appear.

[88] Ramzi A. Haraty, Nash'at Mansour, and Bassel Daou, *Regression testing of database applications*, Symposium on Applied Computing, 2001, pp. 285–289.

[89] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi, *Regression test selection for Java software*, Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001, pp. 312–326.

[90] Mary Jean Harrold, David S. Rosenblum, Gregg Rothermel, and Elaine J. Weyuker, *Empirical studies of a prediction model for regression test selection*, Transactions on Software Engineering **27** (2001), no. 3, 248–263.

[91] Mary Jean Harrold and Mary Lou Soffa, *An incremental approach to unit testing during maintenance*, International Conference on Software Maintenance, 1988, pp. 362–367.

[92] Jean Hartmann, *Applying selective revalidation techniques at Microsoft*, Pacific Northwest Software Quality Conference, 2007, pp. 255–265.

[93] ———, *30 years of regression testing: Past, present and future*, Pacific Northwest Software Quality Conference, 2012, pp. 119–126.

[94] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy, *The art of testing less without sacrificing quality*, International Conference on Software Engineering, 2015, pp. 483–493.

[95] Jerry L. Hintze and Ray D. Nelson, *Violin plots: A box plot-density trace synergism*, The American Statistician **52** (1998), no. 2, 181–184.

[96] *How does merging work?*, http://cbx33.github.io/gitt/afterhours4-1.html.

[97] Pei Hsia, Xiaolin Li, David Chenho Kung, Chih-Tung Hsu, Liang Li, Yasufumi Toyoshima, and Cris Chen, *A technique for the selective revalidation of OO software*, Journal of Software Maintenance: Research and Practice **9** (1997), no. 4, 217–233.

[98] *IntelliJ*, http://www.jetbrains.com/idea.

[99] *Java, Java everywhere*, Feb 2012, http://sdtimes.com/content/article.aspx?ArticleID=36362.

[100] *Jenkins*, https://github.com/jenkinsci/jenkins.

[101] *Jetty*, https://git.eclipse.org/gitroot/jetty/org.eclipse.jetty.project.git.

[102] *JFreeChart*, https://github.com/apetresc/JFreeChart.

[103] *JGit*, https://git.eclipse.org/r/p/jgit/jgit.git.

[104] *JodaTime*, https://github.com/JodaOrg/joda-time.

[105] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi, *Efficient tracing of cold code via bias-free sampling*, USENIX Annual Technical Conference, 2014, pp. 243–254.

[106] Mijung Kim, Jake Cobb, Mary Jean Harrold, Tahsin Kurc, Alessandro Orso, Joel Saltz, Andrew Post, Kunal Malhotra, and Shamkant B. Navathe, *Efficient regression testing of ontology-driven systems*, International Symposium on Software Testing and Analysis, 2012, pp. 320–330.

[107] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa, *Low overhead program monitoring and profiling*, Workshop on Program Analysis for Software Tools and Engineering, 2005, pp. 28–34.

[108] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima, *Class firewall, test order, and regression testing of object-oriented programs*, Journal of Object-Oriented Programming **8** (1995), no. 2, 51–65.

[109] Hareton K. N. Leung and Lee White, *Insights into regression testing*, International Conference on Software Maintenance, 1989, pp. 60–69.

[110] ______, *A cost model to compare regression test strategies*, International Conference on Software Maintenance, 1991, pp. 201–208.

[111] *LinuxKernel Git repository*, git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[112] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov, *An empirical analysis of flaky tests*, International Symposium on Foundations of Software Engineering, 2014, pp. 643–653.

[113] Alexey G. Malishevsky, Gregg Rothermel, and Sebastian Elbaum, *Modeling the cost-benefits tradeoffs for regression testing techniques*, International Conference on Software Maintenance, 2002, pp. 204–213.

[114] Paul Dan Marinescu, Petr Hosek, and Cristian Cadar, *Covrig: A framework for the analysis of code, test, and coverage evolution in real software*, International Symposium on Software Testing and Analysis, 2014, pp. 93–104.

[115] *Memoize*, `https://github.com/kgaughan/memoize.py`.

[116] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa, *Demand-driven structural testing with dynamic instrumentation*, International Conference on Software Engineering, 2005, pp. 156–165.

[117] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso, *Regression testing in the presence of non-code changes*, International Conference on Software Testing, Verification, and Validation, 2011, pp. 21–30.

[118] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig, *A comparative study of manual and automated refactorings*, European Conference on Object-Oriented Programming, 2013, pp. 552–576.

[119] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson, *Mining fine-grained code changes to detect unknown change patterns*, International Conference on Software Engineering, 2014, pp. 803–813.

[120] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig, *Is it dangerous to use version control histories to study source code evolution?*, European Conference on Object-Oriented Programming, 2012, pp. 79–103.

[121] *NetBeans*, `https://netbeans.org`.

[122] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold, *Leveraging field data for impact analysis and regression testing*, International Symposium on Foundations of Software Engineering, 2003, pp. 128–137.

[123] Alessandro Orso and Gregg Rothermel, *Software testing: A research travelogue (2000–2014)*, Future of Software Engineering, 2014, pp. 117–132.

[124] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold, *Scaling regression testing to large software systems*, International Symposium on Foundations of Software Engineering, 2004, pp. 241–251.

[125] Santiago Perez De Rosso and Daniel Jackson, *What's wrong with Git?: A conceptual design analysis*, International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, 2013, pp. 37–52.

[126] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso, *Understanding myths and realities of test-suite evolution*, International Symposium on Foundations of Software Engineering, 2012, pp. 1–11.

[127] *PIT*, `http://pitest.org`.

[128] *randomizedtesting*, `https://github.com/carrotsearch/randomizedtesting`.

[129] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley, *Chianti: A tool for change impact analysis of Java programs*, Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004, pp. 432–448.

[130] *Retrofit*, `https://github.com/square/retrofit`.

[131] Peter C. Rigby, Earl T. Barr, Christian Bird, Prem Devanbu, and Daniel M. German, *What effect does distributed version control have on OSS project organization?*, International Workshop on Release Engineering, 2013, pp. 29–32.

[132] David S. Rosenblum and Elaine J. Weyuker, *Using coverage information to predict the cost-effectiveness of regression testing strategies*, Transactions on Software Engineering **23** (1997), no. 3, 146–156.

[133] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Brian Davia, *The impact of test suite granularity on the cost-effectiveness of regression testing*, International Conference on Software Engineering, 2002, pp. 130–140.

[134] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Xuemei Qiu, *On test suite composition and cost-effective regression testing*, Transactions on Software Engineering and Methodology **13** (2004), no. 3, 277–331.

[135] Gregg Rothermel and Mary Jean Harrold, *A safe, efficient algorithm for regression test selection*, International Conference on Software Maintenance, 1993, pp. 358–367.

[136] ———, *A framework for evaluating regression test selection techniques*, International Conference on Software Engineering, 1994, pp. 201–210.

[137] ———, *Analyzing regression test selection techniques*, Transactions on Software Engineering **22** (1996), no. 8, 529–551.

[138] ———, *A safe, efficient regression test selection technique*, Transactions on Software Engineering and Methodology **6** (1997), no. 2, 173–210.

[139] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold, *Test case prioritization: An empirical study*, International Conference on Software Maintenance, 1999, pp. 179–188.

[140] Raul Santelices and Mary Jean Harrold, *Efficiently monitoring data-flow test coverage*, Automated Software Engineering, 2007, pp. 343–352.

[141] Mats Skoglund and Per Runeson, *A case study of the class firewall regression test selection technique on a large scale distributed software system*, International Symposium on Empirical Software Engineering, 2005, pp. 74–83.

[142] ______, *Improving class firewall regression test selection by removing the class firewall*, International Journal of Software Engineering and Knowledge Engineering **17** (2007), no. 3, 359–378.

[143] *SLOCCount*, `http://www.dwheeler.com/sloccount`.

[144] Amitabh Srivastava and Jay Thiagarajan, *Effectively prioritizing tests in development environment*, International Symposium on Software Testing and Analysis, 2002, pp. 97–106.

[145] *Streamline testing process with test impact analysis*, August 2013, `http://msdn.microsoft.com/en-us/library/ff576128%28v=vs.100%29.aspx`.

[146] *Testing at the speed and scale of Google*, Jun 2011, `http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`.

[147] *TestNG*, `http://testng.org`.

[148] Nikolai Tillmann and Wolfram Schulte, *Parameterized unit tests*, International Symposium on Foundations of Software Engineering, 2005, pp. 253–262.

[149] *Tools for continuous integration at Google scale*, October 2011, `http://www.youtube.com/watch?v=b52aXZ2yi08`.

[150] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson, *Use, disuse, and misuse of automated refactorings*, International Conference on Software Engineering, 2012, pp. 233–243.

[151] *Visual Studio*, `https://www.visualstudio.com`.

[152] Kristen Walcott-Justice, Jason Mars, and Mary Lou Soffa, *THeME: A system for testing by hardware monitoring events*, International Symposium on Software Testing and Analysis, 2012, pp. 12–22.

[153] Peter Weissgerber and Stephan Diehl, *Identifying refactorings from source-code changes*, Automated Software Engineering, 2006, pp. 231–240.

[154] David Willmor and Suzanne M. Embury, *A safe regression test selection technique for database driven applications*, International Conference on Software Maintenance, 2005, pp. 421–430.

[155] Guoqing Xu and Atanas Rountev, *Regression test selection for AspectJ software*, International Conference on Software Engineering, 2007, pp. 65–74.

[156] Stephen S. Yau and Zenichi Kishimoto, *A method for revalidating modified programs in the maintenance phase*, Signature Conference on Computers, Software, and Applications, 1987, pp. 272–277.

[157] Shin Yoo and Mark Harman, *Regression testing minimization, selection and prioritization: A survey*, Journal of Software Testing, Verification and Reliability **22** (2012), no. 2, 67–120.

[158] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid, *Localizing failure-inducing program edits based on spectrum information*, International Conference on Software Maintenance, 2011, pp. 23–32.

[159] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin, *Empirically revisiting the test independence assumption*, International Symposium on Software Testing and Analysis, 2014, pp. 385–396.

[160] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley, *An initial study of a lightweight process for change identification and regression test selection when source code is not available*, International Symposium on Software Reliability Engineering, 2005, pp. 225–234.

[161] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley, *Applying regression test selection for COTS-based applications*, International Conference on Software Engineering, 2006, pp. 512–522.