EVOLUTION-AWARE RUNTIME VERIFICATION

BY

OWOLABI LEGUNSEN

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, 2019

Urbana, Illinois

Doctoral Committee:

Professor Darko Marinov, Chair
Professor Grigore Roşu, Co-Chair
Professor Tao Xie
Professor Sarfraz Khurshid, The University of Texas at Austin

ABSTRACT

The risk posed by software bugs has increased tremendously as software is now essential to many aspects of our daily lives. Software testing is still the most common method for finding bugs during software development, before software is deployed. However, software testing alone is insufficient for finding bugs, as evidenced by the many devastating bugs that frequently manifest in deployed software. Hence, there is a great need to investigate how to use formal-methods based approaches effectively and efficiently during software testing, to help find more bugs during software development.

This dissertation presents work on finding more bugs during software development by performing runtime verification *during* software testing. Runtime verification can help find bugs by monitoring program executions against formally specified properties. Over the last two decades, great research progress has improved the performance of runtime verification, but mostly focused on deployed software. There was little focus on the bug-finding benefits and scalability challenges of using runtime verification during testing of evolving software. Yet, software testing generates many executions on which properties can be monitored to squeeze more bug-finding value from *existing* tests.

This dissertation presents two lines of work on studying and improving the use of runtime verification for finding more bugs during testing of evolving software. Firstly, this dissertation reports on the first large-scale study of runtime verification during software testing. The study performs runtime verification using 199 properties while running 18K developer written tests and 2.1M automatically generated tests in 200 open-source projects. Results show that runtime verification during software testing finds many bugs from existing tests, but incurs high overhead. In spite of tremendous recent research and algorithmic advances in the runtime verification community on improving the runtime overhead, user experience, and monitored properties, runtime overhead was still as high as 33.9×, many property violations were generated that had to be manually inspected, and 84% of the inspected violations were not bugs due to the ineffectiveness of current properties.

Secondly, this dissertation proposes the idea of, and implements the first set of techniques for, reducing the overhead of runtime verification during software testing by exploiting software evolution. All prior runtime verification research focused on checking a single version of software. The proposed evolution-aware techniques extend runtime verification to support multiple software versions and make runtime verification more usable during software evolution. The key insight behind the evolution-aware techniques is to amortize the over-

head of runtime verification across multiple versions by only monitoring the parts of code that changed between versions. Results show that evolution-aware techniques reduce the accumulated runtime verification overhead by up to 10x and show developers two orders of magnitude fewer violations, without missing new violations. We expect the benefits of evolution-aware runtime verification to still apply after future research yields more effective properties with higher rates of bugs found per violation.

To my family

ACKNOWLEDGMENTS

God made the impossible possible many times in my graduate studies and in my entire life.

Oyinade has been there with me through all the highs and lows, successes and failures, deadlines and doubts. This Ph.D. is as much hers as it is mine.

Ana, Dan, Daniella, Lara, and Luiza made innumerable and unspoken sacrifices that contributed greatly to making this dissertation possible.

I hope to one day understand what my Ph.D. advisors ever saw in me to begin with. Throughout my Ph.D., Darko Marinov and Grigore Roşu constantly and tirelessly believed in me, encouraged me, pushed me, challenged me, helped me, advocated for me, mentored me, and championed my cause. I can never repay them, except to do my best to be as good to my students as they have been to me. Together with my advisors, Sarfraz Khurshid and Tao Xie served on my thesis committee. I am grateful for all the time, feedback, and support that they provided to help shape this dissertation.

Mark Gabel showed me what top-notch software engineering research looks like. This dissertation would not be written if Mark did not leave doors of opportunity open for me. Dr. Lawrence Chung provided my first research experience and my first lessons on writing.

Milos Gligoric's mentorship, collaboration, and friendship have been crucial to my professional success. I gained weight after Milos graduated and left me without a gym buddy.

If I was offered a billion dollars to do another Ph.D., I would propose to share the money with Lamyaa Eloussi, Alex Gyori, Farah Hariri, and August Shi to be my lab mates again. What a fantastic journey since we started in 2013! It has been my honor to call you friends.

My friend, Frederico Araujo, was my study partner when I was new and clueless at the University of Texas at Dallas (UTD). I am inspired by his brilliance, wisdom, and humility. Many thanks also to Fred's wife, Rubia, and her family for all the meals, love, and support.

Professor Oluwaranti and Professor Oluwatope mentored me during my undergraduate days at Obafemi Awolowo University (OAU). They also encouraged me and helped me in several ways during my Ph.D., including by writing recommendation letters for me. Several OAU colleagues overlapped with me at UTD and formed a social circle that I could rely on: Richard Antiabong, Paul Asere, Gbadebo Ayoade, Seun Olanipekun, and Olawale Okusanya. Bayo Adeniyi, Gbenga Adetunji, and Yemi Okewole are the best friends a man can wish for. Michael Fashola and his family took me in upon my arrival in the USA and helped me a lot to acclimatize to my new environment.

When my Ph.D. journey almost ended prematurely and unsuccessfully, the Olabintan and

Fowe families were pillars of support, standing in the gap and offering encouragement and advice. Jimi Olabintan constantly supported me and checked on me throughout my Ph.D.

I learned a lot about research, faculty life, and how to be a good person from Jon Bell. Sasa Misailovic and Tianyin Xu allowed me to look over their shoulders as they began life as Assistant Professors. I thank them for their collaboration, support, advice, and advocacy.

I had a lot of fun working with and learning from all my collaborators: Nader Al Awar, Adip Dsouza, Saikat Dutta, Milica Hadzi-Tanovic, Wajih Ul Hassan, Tom Hill, Michael Hilton, Zixin Huang, Ben Lambeth, Yafeng Lu, Stas Negara, Zhenzhou Sun, Sam Supakkul, Xinyue Xu, Tifany Yung, Lingming Zhang, Yi Zhang, and Chenguang Zhu.

I gained a lot from discussions and help from several people: Deniz Arsan, Felicia Chandra, David Craig, Rahul Gopinath, Wing Lam, John Micco, Daejun Park, Amarin Phaosawasdi, Cosmin Radoi, Andrei Stefanescu, Hanjie Wang, Xin Wei, He Xiao, and Peiyuan Zhao. My interactions with all members of Darko Marinov's research group and Grigore Roşu's Formal Systems Laboratory helped broaden my view and improve my communication.

Elsa Gunter went above and beyond the call of duty to help me understand formal methods concepts and why software engineers should use formal methods. Several professors helped me tremendously when I was on the academic job market, including: Sarita Adve, Vikram Adve, Iftekhar Ahmed, Chandra Chekuri, Romit Roy Choudhury, Danny Dig, Katherine Driggs-Campbell, Chris Fletcher, Sayan Mitra, Alex Orso, Madhusudan Parthasarathy, Denys Poshyvanyk, Atul Prakash, Hari Sundaram, Josep Torrellas, Mahesh Viswanathan, and Tandy Warnow.

Parts of this dissertation were published at the International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE NIER) 2015 [115] (that originally proposed the idea of evolution-aware RV), the International Conference on Automated Software Engineering (ASE) 2016 [118] (Chapter 2), the International Symposium on Foundations of Software Engineering (FSE) 2016 [117] (Chapter 3), the International Conference on Automated Software Engineering Tool Demonstrations Track (ASE Demo) 2017 [119] (Chapter 4), and the International Conference on Software Testing, Verification, and Validation (ICST) 2019 [116] (Chapter 5). I am grateful to all the anonymous reviewers of these papers for their comments which helped to shape my work. I am honored that the ASE 2016 paper on studying runtime verification during testing won the Association for Computing Machinery Special Interest Group on Software Engineering (ACM SIGSOFT) Distinguished Paper Award. I am also humbled to have received the Feng Chen Memorial Award in Software Engineering for the ASE 2016 paper; Feng's seminal work on runtime verification provided a solid foundation for my research.

My research was sponsored by the National Science Foundation (through grants CCF-

1421503, CCF-1439957, CNS-1646305, CNS-1740916), Google, Microsoft, and Qualcomm.

My father, Adebisi, and mother, Adefunke, taught me to love learning from an early age and did whatever they could to make sure I got the best education that they could afford. My big sisters Philomena and Mary helped me a lot to settle into the USA; they still constantly call to check on me. I thank my entire family (we are too numerous to list) for supporting me throughout my Ph.D., and for accepting all my weirdness and nerdiness.

Last but not least, I thank Liam for bringing so much joy to my life. I hope that my journey inspires you to pursue what you love and do your best to excel at it. I love you.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Software is more essential than ever before as we now rely on software in many aspects of daily life—from healthcare to aviation, from manufacturing to banking and even to how parents monitor their babies. As software has grown in importance, the risk posed by software failures has also grown. Many articles appeared in the news media about devastating effects to lives and property caused by software failures [8, 141, 204, 206]. The estimated cost of software failures is also very high: 1.7 trillion USD in financial losses, 3.6 billion people affected, and 268 years in accumulated downtime in 2017 alone [134].

Software testing is still the most common approach for finding software bugs early, during development, before these bugs cause software failures after deployment. In particular, to check the quality of code during software evolution, developers commonly practice *regression testing* [215]. In regression testing, after developers make code changes, they rerun tests to check that code changes do not break previously working functionality. Unfortunately, as evidenced by the continuing spate of failures in deployed software, testing alone has not been sufficient for finding bugs during development. There are several reasons for this insufficiency. Automatically generating tests that find bugs continues to be an active area of research [3, 32,54,60–62,70,152,178,194,199,210]. Manually writing effective tests can be challenging for developers [154]. Also, testing is increasingly computationally expensive [50, 74, 86, 175, 216] which can make developers to not run tests as often as they should [133].

Many formal methods exist that can help find a lot of bugs, but they are often not integrated into developers' everyday software development. Specifically, this dissertation focuses on a lightweight formal method called runtime verification (RV) [12, 26, 27, 38, 48, 83, 84, 89, 99, 132]. RV can help to find bugs by monitoring program executions against formally specified properties. There was little research on whether RV can be used effectively and efficiently during regression testing. To be *effective*, RV must be able to find more bugs from existing tests. To be *efficient*, RV must scale to the rapid evolution of today's software, typically managed via continuous integration [50, 59, 87, 88, 140, 192].

This dissertation (1) studies the effectiveness of RV for finding more bugs during software testing, (2) develops a change-impact analysis that assists in adapting RV to multiple software versions, (3) combines RV with regression test selection (RTS), and (4) proposes novel evolution-aware techniques to make RV more efficient during regression testing. Change-impact analysis [11, 31, 36, 120, 126, 169, 170, 176, 179, 220, 221] is concerned with computing parts of code that can be affected by a code change. RTS [69, 81, 149, 169, 174, 211, 220] is a well studied, evolution-centered technique which aims to reduce regression testing time by

only re-running the tests affected by code changes.

This dissertation proposes to use RV for finding bugs earlier by monitoring program executions during software testing, in contrast to prior RV research which focused on monitoring executions of deployed software. A property is a logical formula over a set of *events*, e.g., method calls or field updates; intuitively, it captures developers' intent on correct API usage [171]. RV takes a program to monitor, a set of properties to check, and some program inputs (e.g., tests). The program is then instrumented based on the properties so that executing the instrumented program generates appropriate events and creates *monitors* to listen to events and check the properties. RV outputs *violation messages* (*violations* for short), each of which reports that the execution violated some property at a code location. If a program's execution does not satisfy a property, manual inspection of the resulting violation could help find a bug in the program. If the properties are perfect, every violation would indicate a bug. However, we found in this dissertation that, although RV helped find many bugs, majority of violations were not true bugs. Thus, much more work needs to be done in the future to come up with better properties.

Combining RV with testing is expected to find more bugs than testing alone, using the same set of tests—the tests that developers already have. Whereas software testing is typically concerned with checking whether code satisfies functional input-output properties, RV allows to check a broader class of safety properties at every step during a program's execution. The potential for using RV during software testing was previously *mentioned* [99, 102, 113, 132], but there was no study of RV during software testing and there were no techniques to make RV more efficient during regression testing of evolving software.

Research on combining RV with software testing and on improving the efficiency of RV during regression testing provides several important benefits. First, combining RV with software testing increases the chance to find many more bugs during software development, before those bugs manifest in deployed software and cause costly failures. Second, studying the effectiveness and efficiency of performing RV during testing of real-world software helps researchers gain important knowledge on how RV performs in practice. Prior to this dissertation, most evaluation of RV was carried out on carefully curated benchmarks that may not be representative of test executions in real-world software. Third, performing RV during regression testing has inspired evolution-aware RV techniques that integrate RV into developers' workflow. The work in this dissertation represents the first steps in bridging the technical gap between RV and regression testing, and in bringing the RV and software testing research communities together, towards making RV practical enough to use during regression testing.

1.1 THESIS STATEMENT

The thesis statement of this dissertation is the following:

> *Evolution-aware Runtime Verification can be effective and efficient for finding many more bugs during regression testing.*

This thesis statement has three aspects:

(1) There is a need to investigate the effectiveness and efficiency of performing RV during testing of real-world software.

(2) It is possible to develop an effective static change-impact analysis as a basis for evolution-aware RV.

(3) It is possible to design and develop evolution-aware RV techniques to better scale RV during regression testing.

In support of this thesis, this dissertation presents work on these three aspects: (1) a study of the effectiveness and efficiency of performing RV during testing, using existing properties from the literature while running tests in many open-source projects; (2) the development and evaluation of a static change-impact analysis, called STARTS, which provides a way to reason about code changes as a basis for evolution-aware RV techniques and also provides a way to perform static regression test selection; and (3) a set of evolution-aware RV techniques that adapt RV to the context of evolving software and amortize the overhead of RV across several program versions.

1.2 CONTRIBUTIONS

The work presented in this dissertation makes the following contributions:

- This dissertation presents the first large-scale evaluation of runtime verification during software testing, with 199 properties and 200 open-source projects. The study shows that RV can help find many bugs from existing tests. However, the study also shows that RV incurs high overhead in (1) machine time to monitor properties and (2) developer time to wait for and inspect all property violations from test executions that do not satisfy the properties. Further, the existing properties were largely ineffective and have way too high rates of false alarms. The study analyzes reasons for bug-finding ineffectiveness of existing properties, in particular, the high rates of false

alarms, and discusses developers' feedback on bugs found from manual inspection of violations. Lastly, the study concludes with a set of recommendations that can help the research community engineer more effective properties and better evaluate these properties. The data from the study is publicly available [189].

- This dissertation presents the development and evaluation of STARTS, a static change-impact analysis and RTS tool which forms a basis for evolution-aware RV. *Static* RTS techniques were proposed over three decades ago [106, 107] but were not extensively evaluated on modern software projects. This dissertation presents the first extensive study of the performance benefits of static RTS techniques and their safety; an RTS technique is *safe* if it selects to run all tests that may be affected by code changes. The study implements two static RTS techniques, one class-level and one method-level, and compares several variants of these techniques. The study compares these static RTS techniques against Ekstazi, a state-of-the-art, class-level, dynamic RTS technique. Experimental results on 985 versions of 22 open-source projects show that the class-level static RTS is comparable to Ekstazi, with similar performance benefits, but is sometimes unsafe. In contrast, the method-level static RTS technique performs rather poorly. Therefore, STARTS, a publicly-available open-source tool [191], was developed to perform class-level static change-impact analysis and RTS.

- This dissertation presents and evaluates the first evolution-aware RV techniques which improve the efficiency and usability of runtime verification in the context of software evolution. Specifically, this dissertation presents three evolution-aware RV techniques that reduce RV overhead across multiple program versions: (1) Regression Property Selection (RPS) re-monitors only a subset of properties, namely those that can be violated in parts of code affected by changes, reducing machine time and developer time overhead of RV, (2) Violation Message Suppression (VMS) simply shows only new violations to reduce developer inspection time after code changes; it does not reduce machine time overhead, and (3) Regression Property Prioritization (RPP) splits RV in two phases: properties more likely to have violations that help find bugs are first monitored in a critical phase to provide faster feedback to the developers; the rest are monitored in a background phase. This dissertation also presents 10 variants of RPS and explores their efficiency/safety tradeoff. Additionally, this dissertation presents the first combination of RV with RTS and shows synergy between them. The evaluation compares these evolution-aware RV techniques with the state-of-the-art, evolution-*unaware* RV (*base* RV, for brevity) when monitoring test executions in 200 versions of 10 open-source projects. The results show that RPS and RPP reduce the average

accumulated base RV overhead from 9.4× to as low as 1.8×, were safe, and VMS shows two orders of magnitude fewer violations than base RV—from 54 violations per version (for base RV) to one violation per 10 versions.

The rest of this chapter describes these contributions in more detail.

1.3 STUDY OF RUNTIME VERIFICATION DURING SOFTWARE TESTING

The quality of runtime verification depends on the quality of the properties. While previous research has produced many properties for the Java API, manually or through automatic mining, there has been no large-scale study of their bug-finding *effectiveness* or of their runtime efficiency when monitored during software testing. In this dissertation, a "property" refers to a behavioral specification, defined by Robillard et al. [171] as *"a way to use an API as asserted by the developer or analyst, and which encodes information about the behavior of a program when an API is used"*. A property violation indicates that some API is used in a way that is not consistent with its usage guideline, but such violation may or may not be a real bug in the code. The quality of properties has generally been taken for granted in the runtime verification research community, where the major research direction over the last decade has been to improve the efficiency and scalability of runtime verification algorithms, techniques, and tools. The properties used in previous research were manually written [2, 27, 98, 132] or automatically mined [19, 37, 43, 63, 105, 112, 122, 123, 143, 144, 155–159, 165, 195, 209, 224]. These properties were monitored to measure their runtime overhead on benchmarks, mostly running one large program on one input. However, for finding bugs by combining runtime verification with software testing, it is critical to study the *effectiveness* of these properties and the efficiency of monitoring them during testing of real-world software.

This dissertation presents the first large-scale study of the bug-finding effectiveness and the efficiency of monitoring the previously proposed properties during testing. A property is considered *effective for bug finding* if it can catch true bugs but does not generate too many false alarms. The dissertation focuses on properties of the API from the standard Java library because such properties can potentially find bugs in many projects across various domains, require no domain knowledge, and the runtime verification tool that we evaluate, JavaMOP [92, 99, 132], works for Java. We evaluated 199 existing manually written and automatically mined properties. These include 182 manually written properties that Lee et al. [113, 132] formalized directly from the Java API documentation and used in previous studies on the efficiency and scalability of runtime verification [115, 132, 163]. The rest were 17 properties that were mined automatically from large traces [156] and used in property

mining studies [157, 159].

All 199 properties used in the study were monitored while running 18,065 developer written and 2,135,081 automatically generated tests in 200 open-source projects. We manually inspected a subset of property violations and submitted pull requests for violations we believed to be bugs to the developers of those projects. Specifically, we manually inspected 652 of 5,263 violations of manually written and 200 of 1,141 violations of automatically mined properties. However, there was a high rate of false alarms among the inspected violations and the runtime overhead was high—4.3× on average and as high as 33.9×. Overall false alarm rates of 82.81% and 97.89%, respectively, were observed for manually written and automatically mined properties. Further, only a small fraction of the properties led to the discovery of bugs—11 of 182 manually written and 3 of 17 automatically mined properties—and even among these, the average false alarm rates were high, 45.51% and 96.69%, respectively. Inspecting property violations and submitting pull requests was also very costly, taking an estimated 1,200 hours of student time.

The study results show that runtime verification can be performed during testing of many open-source projects. Also, existing API properties from prior runtime verification and property mining research can find many bugs that developers are willing to fix. However, the study also revealed two serious problems. First, RV overhead during software testing is still very high, both in terms of machine time to monitor the properties during testing and in developer time to wait for and inspect the resulting property violations. Second, false alarm rates are worrisome and suggest a need for the research community to fundamentally rethink property finding and "property engineering" approaches, to make runtime verification a more effective early-stage bug-finding aid that developers can use. The other contributions of this dissertation target the first problem, namely, reducing the overhead of RV during software testing. The second problem—finding more effective properties—is left as future work.

1.4 STATIC CHANGE-IMPACT ANALYSIS AND REGRESSION TEST SELECTION

All prior RV techniques focused on checking a single program version. Yet, software evolves rapidly as developers add new features, fix bugs, or perform refactorings. Developers commonly perform *regression testing* to check that software evolution does not break existing functionality. Therefore, we proposed the idea of evolution-aware RV to reduce the overhead of RV during regression testing, by only rechecking parts of code affected by changes [115]. Evolution-Aware RV is inspired by RTS—a well-studied, evolution-centered technique.

A main requirement needed to make any evolution-aware RV technique work is a fast change-impact analysis. Existing change-impact analyses are either *static* [31, 106, 114, 176],

approximating the effect of code changes without running the code, or *dynamic* [7, 109, 110, 147, 169], computing the effect of code changes based on program execution. Since we dynamically monitor test executions (with JavaMOP), using a dynamic change-impact analysis would incur additional overhead. Therefore, we implemented a static change-impact analysis in a tool called STARTS and evaluated it in the context of RTS, hence, the name STARTS: STAtic Regression Test Selection.

RTS techniques work in two main steps: (1) use a change-impact analysis technique to compute the parts of code that are affected by a code change, and (2) rerun only the tests that are in the affected parts of code, i.e., affected tests. Intuitively, an affected test depends on code that a change-impact analysis computes as affected.

We investigate change-impact analysis at two program granularity levels in the context of static RTS: ClassSRTS performs change-impact analysis at the class level, and MethSRTS performs change-impact analysis at the method level. We evaluate these two static RTS techniques on 985 versions of 22 open-source Java projects. We consider two variants of ClassSRTS and eight variants of MethSRTS, and we compare both variants with the state-of-the-art dynamic RTS technique Ekstazi [68, 69]. The results show that ClassSRTS has comparable performance as Ekstazi, but ClassSRTS is occasionally unsafe. In contrast, MethSRTS performs rather poorly: it does not provide performance benefits and is more frequently unsafe. The latter result is somewhat surprising as one may expect finer-grain, method-level analysis to be safer and more precise (but potentially slower) than the coarser-grain analysis at the class level. We recommend that researchers continue improving static change-impact analysis and RTS techniques at the coarser granularity, which already shows promising results (at least at the level of classes if not modules or projects [76, 186]). Following our own recommendation, we developed our class-level change-impact analysis and ClassSRTS prototype into STARTS, a publicly-available, open-source tool for static change-impact analysis and static RTS. STARTS has already been used as part of the evaluation in other research [39, 76, 116, 226]. Also, several other theses and at least one doctoral dissertation have been written which use or were inspired by STARTS [1, 78, 103, 130, 214].

1.5 EVOLUTION-AWARE RUNTIME VERIFICATION

As software evolves, rerunning traditional, evolution-unaware RV (*base RV*) incurs unnecessarily high overhead: machine time can be wasted on repeatedly checking unaffected parts of code (because base RV is not evolution-aware) and developers can repeatedly see the same violations (even if they want to handle some violations later, they have no way to suppress those violations). Considering the overhead, developers may start ignoring vi-

olations or remove RV altogether. It is therefore important to develop techniques that can reduce RV overhead—in both machine time and developer time—during software evolution.

This dissertation presents compelling evidence that simply taking software evolution into account can significantly reduce RV overhead across *multiple program versions*. We present three evolution-aware RV techniques to reduce RV overhead during software evolution: *regression property selection (RPS)*, *violation message suppression (VMS)*, and *regression property prioritization (RPP)*. The key idea in RPS, VMS, and RPP is to focus RV (and its users) on changed parts of code and new violations that are generated. RPS can reduce the RV overhead in both machine time and developer time, VMS can reduce the overhead in developer time but not in machine time, and RPP can reduce the time for developers to see results for most critical properties, e.g., those historically more likely to find bugs. Our evolution-aware RV techniques can be used together and they are complementary to techniques that make base RV faster on single program versions.

RPS selects to re-monitor only properties that can be violated in parts of code that are *affected* by changes, i.e., either directly syntactically changed or indirectly affected. These affected parts of code may generate new events due to the changes. Our current implementation of RPS re-monitors only properties whose events can come from affected *classes*, i.e., classes that are in the output of change-impact analysis. We focused on class-level RPS following the results of our evaluation of static change-impact analysis and static RTS [117], as well as other recent evolution-aware techniques which showed greater overall benefits of performing analysis only at the class level than at finer-granularity levels like methods or statements [20, 69, 218].

VMS by itself re-monitors all properties in a new code version, but shows only *new violations* that were not in the old version. VMS collects violations from both versions and provides likely mapping of code locations between new and old versions of changed classes. Using this mapping, VMS then filters out violations of the same property that occurred on likely equivalent locations in both the old and new versions. VMS makes it easier to focus on new violations; developers can decide whether to inspect only new or also old violations.

RPP partitions RV into two phases: it monitors only some properties in the critical phase—so called because it is on the developer's critical path from the moment of submitting code changes to the moment of getting the results—and monitors the remaining properties in the background phase. RPP reduces time that it takes for a developer to get feedback on critical properties but still monitors all properties. Developers select critical properties as they want, e.g., those that helped find bugs in the past or those for heavily-used APIs. In our evaluation, critical properties become those that were previously violated.

We define safety and precision for evolution-aware RV techniques (Section 5.2.1): an

evolution-aware RV technique is *safe* if it does not miss a new violation and *precise* if it shows only new violations. We develop two strong RPS variants that are safe under certain assumptions. We also develop 10 weak RPS variants that can trade some safety for more efficiency, i.e., reduced overhead. RPS variants differ in what properties they select and where they instrument the selected properties.

We compared RPS, VMS, and RPP with base RV using 161 properties on 200 versions of 10 open-source projects (20 versions per project). The 161 properties are those that remain from a set of 182 manually written properties, after removing 21 properties that we identified in our previous study as being broken [118]. The results show that our evolution-aware RV techniques substantially reduce the runtime overhead and number of violations shown, compared to base RV. We compute the runtime overhead and the number of violations showed per version, then average across versions of a project and then across all projects.

Base RV has average runtime overhead of 9.4×, showing 54 violations per version. The two strong RPS variants have runtime overhead of 7.5× and 7.9×, showing 37 and 42 violations, respectively. The 10 weak RPS variants have runtime overhead of 2.5×–7.5×, showing 21–37 violations. Surprisingly, *all* weak RPS variants were safe in our experiments although they can be unsafe in theory. Our manual inspection shows why: all new violations happened due to changes whose effects were in the classes computed as affected by all weak RPS variants. VMS has negligible extra runtime overhead and reduces the number of violations shown by two orders of magnitude relative to base RV. VMS shows, on average, 0.1 new violation per version, while base RV shows 54 violations per version. RPP's critical phase overhead is 1.8× (when combined with RPS) and our analysis of RPP showed that about 76% of base RV overhead goes into monitoring unviolated properties.

1.6 DISSERTATION ORGANIZATION

The rest of this dissertation is organized as follows.

Chapter 2: Runtime Verification during Software Testing

This chapter presents our study of RV during software testing; RV helped find many bugs but incurred high overhead. These results partly motivated the work on reducing the accumulated overhead of RV during software evolution.

Chapter 3: Static Change-Impact Analysis and Regression Test Selection

This chapter presents the implementation of static change-impact analysis as a

basis for evolution-aware RV. Evaluation is done at different levels of granularity and compared with a dynamic approach in the context of RTS.

Chapter 4: STARTS: Static Change-Impact Analysis and RTS Tool

This chapter presents the design and implementation of STARTS, our open-source tool for change-impact analysis and RTS, based on the results from Chapter 3. STARTS is central to the evolution-aware RV techniques in Chapter 5.

Chapter 5: Evolution-Aware Runtime Verification

This chapter presents the first set of evolution-aware RV techniques, which amortize the overhead of performing RV during regression testing by focusing RV and its users on parts of the code affected by the changes.

Chapter 6: Related Work

This chapter presents an overview of other research related to the work presented in this dissertation.

Chapter 7: Conclusions and Future Work

This chapter concludes the dissertation and highlights some avenues for future work to extend the results presented in this dissertation.

CHAPTER 2: RUNTIME VERIFICATION DURING SOFTWARE TESTING

This chapter presents the first large-scale, in-depth study of performing RV during software testing of open-source software. Specifically, we evaluate bug-finding effectiveness and the efficiency of monitoring previously proposed properties during the execution of developer written and automatically generated tests. The results of this study partially motivated the work presented in chapters 3–5. The rest of this chapter is organized as follows. Section 2.1 provides a background on RV. Section 2.2 describes the experimental setup for our study. Section 2.3 presents quantitative results, while Section 2.4 discusses qualitative results based on analysis of developer responses to pull requests that we submitted for bugs found during our study. Section 2.5 provides some discussion about our study. Finally, Section 2.5.1 contains our recommendations, based on our study results, for improving the research on coming up with more effective properties for finding bugs when performing RV during testing.

2.1 BACKGROUND

In RV, the execution of a software system is dynamically checked against formal properties [27, 29, 38, 48, 89, 132, 138, 139]. At a high level, the program being monitored is instrumented to capture, as *events*, method calls and field updates related to the properties being checked. Then, at runtime, the instrumented program creates listener objects, called *monitors*, which check that the events conform to the properties and report *violations* when execution does not satisfy some property. In this dissertation, a "property" refers to a behavioral specification, defined by Robillard et al. [171] as *"a way to use an API as asserted by the developer or analyst, and which encodes information about the behavior of a program when an API is used"*. A property violation indicates that an API is used inconsistently with its usage guideline; such violation may or may not be a true bug in the code.

2.1.1 Overview of our Study

In this dissertation a property is *effective for bug finding* if it can catch true bugs but does not generate too many false alarms. We evaluate 199 existing manually written and automatically mined properties: 182 manually written properties that were formalized directly from the Java API documentation [113] and used in previous studies on the efficiency and scalability of runtime verification [115, 132, 163]. We also use 17 properties that were mined automatically from large traces [156] and were used in property mining studies [157, 159].

```
1 Collections_SynchronizedCollection(Collection c, Iterator i) {
2  Collection c;
3  creation event sync after() returning(Collection c):
4   call(* Collections.synchronizedCollection(Collection)) || ... /* more calls */) { this.c = c; }
5  event syncMk after(Collection c) returning(Iterator i) :
6   call(* Collection+.iterator()) && target(c) && condition(Thread.holdsLock(c)) {}
7  event asyncMk after(Collection c) returning(Iterator i) :
8   call(* Collection+.iterator()) && target(c) && condition(!Thread.holdsLock(c)) {}
9  event access before(Iterator i) :
10  call(* Iterator.*(..)) && target(i) && condition(!Thread.holdsLock(this.c)) {}
11 ere: (sync asyncMk) | (sync syncMk access)
12 @match { RVMLogging.out.println(Level.CRITICAL, __DEFAULT_MESSAGE); ... /* more printing */ }
13 }
```

Figure 2.1: Example property, Collections_SynchronizedCollection (CSC), with its events and specification

2.1.2 Runtime Verification in JavaMOP

We briefly describe RV of properties in JavaMOP [38, 92, 99, 132, 139]. `Collection_SynchronizedCollection` (`CSC`), shown in Figure 5.1a, is one of the properties in our study. `CSC` was earlier proposed by Bodden et al. [28] (they called it `ASyncIteration`) to check for cases where a synchronized `Collection`'s `Iterator` is accessed from some non-synchronized code. Figure 5.1a shows the three parts of a JavaMOP property: lines 3–10 define the *events* relevant to the property, line 11 is the formal *specification* to monitor over the events, and line 12 shows user-defined *handler* code that JavaMOP invokes when the monitored program reaches a certain state, i.e., when the property is violated.

Each property is parameterized by the types of objects whose instances may generate the events. Specifically, `CSC` is parameterized (line 1) by `Collection c` and `Iterator i`, which means that one monitor object will be created at runtime for every pair of related `c` and `i`. The `creation` keyword indicates that a monitor will be created after the `sync` event occurs (i.e., when one of the `synchronized*` methods on line 4 is invoked on a `Collection`). The monitor subsequently listens for the events `syncMk` (line 5), `asyncMk` (line 7), and `access` (line 9). The `syncMk` events occur after `iterator()` is invoked on a `Collection` instance, `c`, to create an `Iterator`, `i`, and the thread did synchronize on `c` (lines 5–6). The `asyncMk` events occur after `iterator()` is invoked on `c`, but the thread did not synchronize on `c` (lines 7–8). Finally, the `access` events occur before any invocation of `Iterator` methods on `i` from any thread that did not synchronize on `c` (lines 9–10).

If the monitored program ever reaches a state where the extended regular expression (`ere`) property on line 11 is matched, then the handler code on line 12 is invoked. The `ere` matches when non-synchronized code creates an `Iterator` from a synchronized `Collection` (`sync asyncMk`) or when accessing a synchronized `Collection`'s `Iterator` from non-synchronized code (`sync syncMk access`). In our experiments, we used the default handler in JavaMOP:

```
1   im = Collections.synchronizedList(...);
2 + synchronized(im) {
3   for (IInvokedMethod iim : im) {
4     ITestNGMethod tm = iim.getTestMethod();
5     ... }
6 + }
```

Figure 2.2: Buggy code in `TestNG`

```
Specification Collections_SynchronizedCollection has been violated on line org.testng.reporters.SuiteHTMLReporter.
 generateMethodsChronologically(SuiteHTMLReporter.java:365). Documentation for this property can be found at
 https://runtimeverification.com/monitor/annotated-java/__properties/html/java/util/
 Collections_SynchronizedCollection.html
A synchronized collection was accessed in a thread−unsafe manner.
```

Figure 2.3: A sample violation

print a violation containing the property name, the program line number where the property violation occurred, a URL for the property, and an explanation.

As an example, consider the buggy code in Figure 2.2, which is simplified from one of six bugs that we found in `TestNG`, a widely used unit-testing framework. The lines not starting with "+" (1 and 3–5) represent part of the original code that iterates over the synchronized `Collection im`. Note that the `for` loop is not synchronized, leading to a violation of the `CSC` property. The violation that JavaMOP reports is shown in Figure 5.3; our inspection starting from this reported line of code led us to find the bug. The developers accepted our pull request that added the synchronization code, in the lines starting with "+" (2 and 6).

2.2 EXPERIMENTAL SETUP

We describe the open-source projects used in our study, the manually written and automatically mined properties that were monitored while running tests in the projects, and how we automatically generated tests using Randoop [150, 152, 164]. We also explain our procedure for running JavaMOP on the projects and for inspecting the resulting violations.

2.2.1 Experimental Subjects

We selected the projects for our study from GitHub, starting from a list of the most popular Java projects. From these, we selected 200 projects that (1) used Maven (for ease of automation), (2) had at least one test (so we can monitor test runs), (3) had all tests pass without monitoring, and (4) had all tests pass when monitoring with JavaMOP. Requirements (3) and (4) are important to have a fair measurement of runtime overhead of JavaMOP—if tests were to fail between the two runs, with and without monitoring, they may fail at different points in the execution, leading to rather different time measurements. Furthermore, tests

Table 2.1: Some statistics of 200 projects used in our study

PID	Project	SHA	LOC	ManTests	AutoTests
P1	Altoros.YCSB	bfcfe23a	7290	1	–
P2	LogBlock.LogBlock-2	40548aad	875	1	–
P3	edanuff.CassandraCompositeType	6d09cceb	1234	1	5427
P4	jriecken.gae-java-mini-profiler	80f3a59e	908	8	92058
P5	mqtt	f4384253	11478	18	–
P6	plista.kornakapi	178061c3	3088	2	21594
P7	threerings.playn	c969160c	38388	139	–
P8	tbuktu.ntru	8126929e	7715	70	–
P9	OpenGamma.ElSql	db6c6d07	2581	160	11034
P10	sematext.ActionGenerator	10f4a3e6	1864	7	–
P11	vivin.GenericTree	15c59c99	677	49	7787
P12	hoverruan.weiboclient4j	6ca0c73f	8748	34	1229
P13	joda-time	cc35fb2e	85000	4157	12123
P14	IvanTrendafilov.Confucius	2c302878	1203	84	23196
P15	mikebrock.jboss-websockets	fd03a4ef	1736	1	6668
P16	b3log.b3log-latke	afb48c40	24399	76	–
P17	Thomas-S-B.visualee	410a80f0	3574	76	8164
P18	asterisk-java	b07617fe	39498	220	33632
P19	Cue.lucene-interval-fields	8f8bff6d	736	9	13162
P20	JSqlParser	001d665d	10517	341	14837
P21	Ovea.jetty-session-redis	afb2b25b	6358	7	15414
P22	bcel	24014e5e	35827	87	–
P23	zookeeper-utils	a2b80474	455	4	633
P24	bucchi.OAuth2.0ProviderForJava	db5e1d06	2654	47	–
P25	htrace	c32ec0b1	2521	11	–
P26	ptgoetz.storm-jms	d152d72f	1085	2	–
P27	UrbanCode.terraform	d67ac40c	12108	4	3069
P28	pignlproc	1a609980	2296	19	53693
P29	jmxtrans.embedded-jmxtrans	4f1ce2cc	5806	56	–
P30	apache.gora	bb09d891	24185	56	–
F69	69 projects with 100% FAR	various	349029	3834	561031
N101	101 projects without violations	various	520472	8484	1250330
		TOTAL	1214305	18065	2135081
		AVG	6071.52	90.33	17500.66
		MIN	24	1	1
		MAX	93260	4157	219404

could fail due to problems in the project or due to integration of JavaMOP. For example, we observed some failures of time-sensitive tests that have some timeouts resulting from the time or memory overhead of JavaMOP. We also observed test failures that happened because JavaMOP instrumentation interacted unexpectedly with some other instrumentation frameworks, e.g., test-mocking frameworks. We already reported some of these issues to the JavaMOP project on GitHub [189].

Table 2.1 lists some basic statistics about the 200 projects used in our study. PID either starts with "P" to provide the short ID of a project in which we found some real bug, or summarizes multiple projects with similar characteristics—"F69" summarizes 69 projects in which all inspected violations were false alarms and "N101" summarizes 101 projects in which no violations were generated for the properties that we inspected. Project is the

project name, SHA is the project version we used, LOC is the number of Java lines in the project, ManTests is the number of manually written tests, and AutoTests is the number of automatically generated tests. "–" marks that we did not have Randoop test methods, which happened for 49 projects with multiple Maven modules, 16 projects where generated tests did not compile, and 13 projects where Randoop did not generate any test method within the time limit. For F69 and N101, ManTests and AutoTests show the sums for all respective projects. The rows TOTAL, AVG, MIN, and MAX are the sum, average, minimum, and maximum across all projects in each column.

2.2.2 Properties Used in this Study

All Java API properties that we used in our study were obtained from the literature, 182 manually written properties [113, 132] and 17 automatically mined properties [159, 160]. We next describe our rationale and procedure for selecting each set of properties.

Manually Written Properties: We used 182 manually written JavaMOP properties [115, 132], which are publicly available [161]. The properties were originally written by Lee et al. [113], who read Javadoc comments in four widely-used packages (`java.lang`, `java.net`, `java.io`, and `java.util`) and formalized sentences describing "must", "should" or "is better to" conditions. The properties are formalized using finite-state machines (FSM), extended regular expressions (ERE), linear temporal logic (LTL), and context-free grammars. JavaMOP can monitor properties in any formalism for which a suitable logic plugin exists.

To illustrate manual formalization of properties, consider again the `CSC` property [42] from Section 2.1. It was formalized as an ERE from text in `Collections.synchronizedCollection()` method's Javadoc: *"It is imperative that the user manually synchronize on the returned collection when iterating over it ... Failure to follow this advice may result in non-deterministic behavior"* [93]. Section 2.1 explained `CSC` in detail, line-by-line. As mentioned there, `CSC` was used earlier [28]; by analyzing Javadoc comments, Lee et al. [113] ended up with some properties that others had formalized before. Monitoring `CSC` in our experiments revealed bugs in several widely used projects, including `TestNG`, `ActiveMQ`, and `XStream`. However, our experiments also revealed a number of issues and opportunities for improving the manually written properties, discussed in Section 2.4.2.

Automatically Mined Properties: To compare the effectiveness of manually written properties and automatically mined properties, we monitored 17 of the 223 properties automatically mined by Pradel et al. [156, 158–160]. Before settling on these properties, we searched for mined properties for Java by performing a mini-survey of the property mining literature to search for properties and to identify how property mining was evaluated.

Table 2.2: Mini-Survey. Ref: references; Subjects: kind of subjects; OSS: open-source projects; Sel-Classes: selected classes; #Sub: number of subjects; FAR[%]: false alarm rate reported; #Bugs: number of bugs found; Rep?: bugs reported to developers?

Ref	Subjects	#Sub	FAR[%]	#Bugs	Rep?
[155]	DaCapo+OSS	12	n/a	n/a	n/a
[165]	n/a	n/a	n/a	n/a	n/a
[105]	n/a	8	n/a	n/a	n/a
[143]	DaCapo	7	43.00	20	no
[43]	OSS+JDK	7	n/a	n/a	n/a
[224]	OSS	5	73.90	100	yes
[209]	OSS	8	n/a	1	no
[157]	DaCapo	10	0.00	54	no
[122, 123]	Sel-Classes	3	n/a	n/a	n/a
[207]	OSS	6	58.00	9	yes
[37]	OSS	4	n/a	n/a	n/a
[144]	OSS	3559	n/a	n/a	n/a
[63]	DaCapo	11	70.00	11	no
[19]	DaCapo	1	n/a	n/a	n/a
[111]	OSS	7	5.00	265	no
[159]	DaCapo	12	49.00	26	no
[195]	Sel-Classes	15	n/a	n/a	n/a

Paper Search: We searched for property mining papers on DBLP [44] using this query:
`specification|propert|contract|invariant|precondition mining|monitor|enforce|infer|mine venue:ICSE|venue:ASE|venue:RV|venue:PLDI|venue:POPL|venue:ISSTA|venue:ieee_trans_software_eng_tse_|venue:sigsoft_fse|venue:autom_softw_eng_ase_|venue:esec_sigsoft_fse|venue:tacas|venue:icsm|venue:icsme|venue:sas|venue:sac|venue:paste|venue:icfem|venue:issre|venue:compsac|venue:formats|venue:sttt|venue:ecoop|venue:fase|venue:oopsla_companion|venue:kdd|venue:vmcai|venue:seke|venue:cav|venue:oopsla|venue:electr_notes_theor_comput_sci_entcs_`
We obtained 163 potentially related papers, of which we considered only the 100 papers published in 2009–2015.

Paper Filtering: We split these 100 papers in half and two of the authors of our paper [118] read abstracts from each half independently to find relevant papers that mined Java API properties that we could use. We omitted related papers, e.g., a survey [171], which did not report finding new properties. The result was 26 papers that we then read in more detail to answer these questions: (1) in what formalism are the mined properties (and can they be monitored with JavaMOP)? (2) how many properties did they mine? (3) did they find any bugs? (4) do they report false alarms from evaluating the bug-finding effectiveness of the properties? (5) what is the reported false alarm rate, if any?

Email to Authors: After filtering, we settled on 17 papers and emailed authors who are not at our institution to ask for their mined properties. We received responses from authors of 7 papers, with 5 providing their properties. Of these 5, the properties from Pradel et al. [159] had the largest number that we could easily use—their properties were provided in the DOT format, which was straightforward to automatically translate to finite-state machines in the JavaMOP syntax.

Prior Evaluations: Table 2.2 lists the 17 papers whose authors we emailed. Although 7 papers report finding bugs while evaluating mined properties, only 2 papers report confirming the bugs with the developers. Further, evaluations were mostly performed on DaCapo, the benchmark initially curated to evaluate performance and not bug-finding effectiveness, and on a small number of open-source projects, with the exception of Nguyen et al. [144] who used thousands of projects but only to apply statistical techniques to mine properties and not to evaluate their bug-finding effectiveness. Finally, among the 7 papers that reported false alarm rates, the rates varied widely, from 0.0% to 73.9%. Our experiments are therefore complementary to those in the papers we surveyed on property mining. In fact, we find even higher false alarms rates. As for the manually written properties, our experiments also revealed issues and opportunities for improvement in the automatically mined properties, as discussed in Section 2.5.

2.2.3 Runtime Verification with JavaMOP

Using JavaMOP to monitor test runs is quite simple: integrate JavaMOP in the project and invoke `mvn test`. JavaMOP integration in Maven-based projects is described online [91]. First, the JavaMOP compiler generates a Java agent [145] from the properties to be monitored, enabling dynamic instrumentation of code running in the Java Virtual Machine. Next, the Maven build configuration file, `pom.xml`, is modified to make the Maven Surefire plugin (which runs the tests) aware of the JavaMOP agent. Subsequent invocations of `mvn test` attach the JavaMOP agent to the test-running process for monitoring the runs against all the properties simultaneously. We fully automated JavaMOP agent creation, changing `pom.xml`, monitoring each project, and post-processing results. This allowed us to scale our experiments to 199 properties and 200 projects.

In each set of experiments, we ran the tests in each project twice. First, we ran without integrating JavaMOP to measure the base test-running time and as a check that the tests in the projects pass by themselves. We then integrated JavaMOP and reran the tests to measure test-running time with JavaMOP and to record violations. We configured JavaMOP to log all output to a file. We excluded from monitoring standard Java libraries (that are

less likely to have bugs) and some third-party libraries, such as Maven Surefire (to reduce overhead) and test-mocking frameworks (which we found to have unexpected interactions with JavaMOP, as mentioned in Section 2.2.1). All JavaMOP experiments to monitor the execution of manually written tests were run on a 64-bit computer with Intel® Core™ i7-3770K CPU @ 3.50GHz processor and 32GB of RAM running Ubuntu 14.04.4 LTS and Java 7 or 8 (as required by the project).

2.2.4 Automatically Generating Tests

To evaluate whether the type of tests impacts the bug-finding effectiveness of the properties, we used Randoop [150, 152, 164] to automatically generate additional tests. We generated tests and monitored them on a Core™ i7-4700MQ 2.40GHz Quad-Core processor PC with 8GB of RAM, running Ubuntu 15.04, Java 7 or 8 (as required by the project), and Randoop heap usage limited to 4GB. We ran Randoop on all 151 single-module Maven projects (out of total 200 single- and multi-module projects), which were easier to automate than multi-module Maven projects. We limited test-generation time to 1 mins and 5 mins. After generating tests, we had a separate run to monitor the generated tests (using JavaMOP) against the same set of manually written properties. The number of *new* violations, i.e., those which were not already reported while monitoring manually written tests, showed little difference between the tests automatically generated in 1 mins and 5 mins. Therefore, we decided to use the tests generated in 5 mins and did not increase the time limit for Randoop any further. Other researchers who used Randoop also found tests generated in different intervals to behave similarly [151, 178, 196].

2.2.5 Inspecting Violations

We describe our procedure for selecting and inspecting violations that JavaMOP reported while monitoring test runs. We refer to the source-code line number at which JavaMOP reports a property violation as the violation *site*. JavaMOP reports a violation every time a property is violated at runtime, so it can report many violations of the same property at the same site (e.g., if the site is in a loop or invoked from multiple tests). We refer to all violations that are reported by JavaMOP during test execution as *dynamic violations (DV)* and we refer to unique violations—those that happen in the same project, for the same property, and at the same site—as *static violations (SV)*.

We manually inspected some static violations from both manually written and automatically generated properties. For manually written properties, we inspected all violations

from 42 properties and ignored all violations from 21 properties. For automatically mined properties, we sampled to inspect 200 out of 1141 violations of the 17 automatically mined properties that we monitored. To sample 200 violations, we used stratified sampling [40]: we divided all violations into strata based on the property and from each stratum randomly selected a number of violations, in proportion to the ratio of the stratum's size to the total number of violations. We excluded 21 manually written properties from inspection and did not monitor 206 automatically mined properties because of issues with these properties, discussed in Section 2.4.2.

Our inspection goal was to find as many bugs as possible while increasing the chance that the developers accept the resulting pull requests. Therefore, multiple authors of our paper [118] inspected most violations. For manually written tests and manually written properties, first, two reviewers independently inspected each violation and classified it as:

TrueBug. A potential bug to be confirmed by reporting to the developers or by checking if it was already fixed;

FalseAlarm. The violation does not indicate a bug in the code but effectively a bug/imprecision in the property; or

HardToInspect. The violation is hard to classify as a TrueBug or a FalseAlarm, because source code is missing or is particularly hard to reason about.

Next, the independent reviewers met to discuss and agree on the classifications they had independently assigned and to resolve cases in which one reviewer had classified a violation as a TrueBug but the other had given another classification. Cases where they still could not agree were classified as TrueBug if any one of the reviewers had classified as a TrueBug. A third reviewer then met with the two initial reviewers to confirm all violations that were classified as TrueBugs. For automatically mined properties, we followed a similar procedure: two reviewers inspected each violation reported from monitoring automatically mined properties while running manually written tests. For automatically generated tests, only one reviewer inspected each violation because we had built enough experience from inspecting the violations from manually written tests.

For each violation that we classified as a TrueBug, we submitted a bug report and/or a fix (pull request) to the developers of the respective project to check whether they agree that a code change can be beneficial. As discussed in Section 2.5, inspecting violations and submitting pull requests to developers is challenging. For inspections alone, each of the two initial reviewers spent between 4 mins and 54 mins per violation. Summing up all the time

to meet for resolving disagreements, to prepare pull requests, to iterate over them internally, to communicate with developers, and to record and process the status of each pull request, we estimate that it took over 1200 hours just for this process of inspecting and creating pull requests. The next section discusses the inspection results and other experimental results.

We carefully prepared pull requests, trying to obtain an "upper bound" on the effectiveness of the properties. That is, some violations that we classified as TrueBugs may have been ignored by developers running a tool on their own or in the absence of our carefully prepared pull requests. We did not simply submit bug reports indicating the violation of a property in a codebase; we were concerned that developers may not understand the property or care to change the code. Instead, we submitted pull requests that included a proposed code change.

2.3 RESULTS

We aim to evaluate the effectiveness of existing properties for finding bugs when monitoring test runs in open-source projects. We discuss here some quantitative aspects of the results.

2.3.1 Research Questions

We investigated the following research questions (RQs):

RQ1 What is the runtime overhead of monitoring?

RQ2 How many bugs are found from violations?

RQ3 What are the false alarm rates among violations?

2.3.2 RQ1: Runtime Overhead of Monitoring

Table 2.3 shows the runtime overhead (Overhead[%]) from monitoring 42 manually written properties. We measured overhead only for manually written (and not automatically generated) tests, because they pass in all 200 projects (while some automatically generated tests fail, making it hard to reliably measure overhead). Runtime overhead is computed as $(mop - base)/base * 100\%$, where *mop* is the time to run tests *with* monitoring and *base* is time to run the tests *without* monitoring. As in previous JavaMOP studies, we observed some negative runtime overheads, e.g., in P5. These can be due to noise in the time measurements or due to the instrumentation changing the garbage-collection behavior of the program, causing it to run faster [98, 100, 139].

Table 2.3: Dynamic (DV) and static (SV) violations, and overhead for 42 manually written properties. ManTests: manually written tests; AutoTests: automatically generated tests; PID, TOTAL, AVG, MIN, MAX, "–": as in Table 2.1

PID	ManTests			AutoTests	
	DV	SV	Overhead[%]	DV	SV
P1	13	4	187.93	–	–
P2	1	1	50.96	–	–
P3	20	2	110.72	0	0
P4	0	0	157.72	20	1
P5	412	2	-28.37	–	–
P6	24	2	155.36	0	0
P7	1	1	201.39	–	–
P8	27	7	27.88	–	–
P9	384	9	239.98	0	0
P10	961	3	128.17	–	–
P11	26	5	128.86	7	1
P12	0	0	248.97	558	16
P13	236	95	245.26	0	0
P14	74	1	123.09	75242	9
P15	0	0	2.30	287	3
P16	167	13	72.25	–	–
P17	37	13	126.38	18	1
P18	2	1	104.53	6717	6
P19	746	5	284.76	12520	3
P20	27977	1	105.98	1493	3
P21	21	4	324.51	7241	4
P22	181430	4	338.72	–	–
P23	1038	16	67.46	0	0
P24	88	5	228.58	–	–
P25	31	10	69.88	–	–
P26	7	5	51.50	–	–
P27	0	0	84.23	1322	8
P28	414	13	14.57	0	0
P29	29	13	4.30	–	–
P30	467	29	616.59	–	–
F69	85120	269	13297.49	96111	64
N101	0	0	8106.04	0	0
TOTAL	299753	533	25877.95	201536	119
AVG	1498.77	2.67	129.39	1651.93	0.98
MIN	0	0	-28.37	0	0
MAX	181430	95	1036.57	75242	16

The average runtime overhead was 129.39% when monitoring only the 42 inspected properties (as shown in the table) and 330.14% when monitoring all 182 manually written properties (elided for lack of space). Therefore, the overhead of simultaneously monitoring all properties is under 4.3× on average. We believe this runtime overhead is acceptable during development time (not production time), considering the number of bugs we found and the fact that the tests in these projects run relatively fast—the average additional time incurred by JavaMOP was 4.08s for 42 properties and 12.48s for 182 properties. The relatively small

average overhead reflects the tremendous progress made in the research community over the last decade to make runtime verification more efficient.

Table 2.3 also shows the number of dynamic (DV) and static (SV) violations from monitoring 42 manually written properties on both manually written tests (ManTests) for all 200 projects and automatically generated tests (AutoTests) for 122 projects (of the 200 projects, 151 were single-module Maven projects, but the tests generated by Randoop did not compile in 16 projects, and Randoop did not generate any test in 5 mins for 13 projects). Even when DV is relatively high, the overhead remains reasonable.

2.3.3 RQ2: Bugs Found

We found a total of 114 SV that were TrueBugs, 110 for manually written properties and 4 for automatically mined properties. Recall that we map dynamic to static violations based on the project being monitored, the property being violated, and the violation site. When multiple projects use the same library (even if not the exact same version), then multiple static violations can actually map to the same bug. Our 114 TrueBugs map to 97 unique bugs. Because most projects evolved since we started our experiments (with then latest versions of the projects), 2 of the unique bugs we found were already fixed in the current latest versions. For the remaining 95 bugs, we submitted pull requests, with 74 already accepted and only 3 rejected; the remaining 18 are still pending.

2.3.4 RQ3: False Alarm Rates

A key metric to evaluate the effectiveness of properties is the false alarm rate (FAR), i.e., the ratio $FA/(TB + FA)$, where FA and TB are the number of FalseAlarms and TrueBugs among inspected violations. For manually written properties, we inspected 652 violations—533 from manually written tests and 119 from automatically generated tests. Table 2.4 shows, for each project in which we inspected violations, the project ID (PID), number of inspected static violations (SV), number of violations in each classification (HTI, TB, and FA), and false alarm rate (FAR[%]). All 69 projects in F69 have 100% FAR (no TrueBugs) and had slightly more violations than all those with TrueBugs. 19 of 30 projects with some TrueBug had greater than 50% FAR. The TOTAL row shows the overall FAR: for manually written properties, it is 82.81% (110 TrueBugs and 530 FalseAlarms). For automatically mined properties, we inspected 200 violations. We elide the breakdown per project, but the overall FAR for automatically mined properties is 97.89% (4 TrueBugs and 186 FalseAlarms).

We further analyzed FAR along several dimensions, trying to identify where it may be

Table 2.4: Per-project inspection summary for 42 manually written properties. SV: static violations; HTI: hard to inspect; TB: true bugs; FA: false alarms; FAR[%]: false alarm rate

PID	SV	HTI	TB	FA	FAR[%]
P1	4	0	4	0	0.00
P2	1	0	1	0	0.00
P3	2	0	2	0	0.00
P4	1	0	1	0	0.00
P5	2	0	2	0	0.00
P6	2	0	2	0	0.00
P7	1	0	1	0	0.00
P8	7	0	6	1	14.29
P9	9	0	6	3	33.33
P10	3	0	2	1	33.33
P11	6	0	3	3	50.00
P12	16	0	7	9	56.25
P13	95	0	40	55	57.89
P14	10	0	4	6	60.00
P15	3	0	1	2	66.67
P16	13	0	4	9	69.23
P17	14	0	4	10	71.43
P18	7	0	2	5	71.43
P19	8	0	2	6	75.00
P20	4	0	1	3	75.00
P21	8	0	2	6	75.00
P22	4	0	1	3	75.00
P23	16	0	4	12	75.00
P24	5	0	1	4	80.00
P25	10	0	2	8	80.00
P26	5	0	1	4	80.00
P27	8	0	1	7	87.50
P28	13	1	1	11	91.67
P29	13	0	1	12	92.31
P30	29	1	1	27	96.43
F69	333	10	0	323	100.00
TOTAL	652	12	110	530	82.81

lower. Table 2.5 (top part) shows the FAR breakdown for manually written properties. Violations in third-party libraries had 86.55% FAR, while violations in the project code had 80.82% FAR. Violations in single- vs. multi-module Maven projects had 81.87% vs. 86.23% FAR, and violations for manually written tests vs. automatically generated tests had 82.51% vs. 84.21% FAR. The similar FARs across all these dimensions suggests that the FARs are mostly due to inherent (in)effectiveness of the properties and less due to specific code-related factors. An interesting finding is that violations in libraries are somewhat more likely to be false alarms, as one would expect that libraries are indeed better tested and have fewer bugs than the project code.

Table 2.5 (bottom part) shows the breakdown for automatically mined properties. Com-

Table 2.5: Split of inspection results along various dimensions. Column headers are same as in Table 2.4

Type of properties	SV	HTI	TB	FA	FAR[%]
Manually written	652	12	110	530	82.81
Libraries	232	9	30	193	86.55
Project code	420	3	80	337	80.82
Single-module	513	11	91	411	81.87
Multi-module	139	1	19	119	86.23
ManTests	533	7	92	434	82.51
AutoTests	119	5	18	96	84.21
Automatically mined	200	10	4	186	97.89
Libraries	122	10	0	112	100.00
Project code	78	0	4	74	94.87
Single-module	148	9	3	136	97.84
Multi-module	52	1	1	50	98.04

Table 2.6: Per-property inspection summary. Column headers are same as in Table 2.4

Property	SV	HTI	TB	FA	FAR[%]
URLDecoder_DecodeUTF8	1	0	1	0	0.00
Collections_SynchronizedCollection	22	0	19	3	13.64
Collections_SynchronizedMap	5	0	4	1	20.00
Byte_BadParsingArgs	3	0	2	1	33.33
Long_BadParsingArgs	22	0	14	8	36.36
InetSocketAddress_Port	2	0	1	1	50.00
ByteArrayOutputStream_FlushBeforeRetrieve	123	0	55	68	55.28
StringTokenizer_HasMoreElements	11	0	4	7	63.64
Math_ContendedRandom	14	0	5	9	64.29
Short_BadParsingArgs	3	0	1	2	66.67
Iterator_HasNext	157	3	4	150	97.40
31 Properties with 100% FAR	289	9	0	280	100.00
TOTAL	652	12	110	530	82.81

pared to manually written properties, the FAR values are higher along all dimensions. The overall FAR was 97.89% (186 of 190 non-HTI violations). Compared within different dimensions, the FAR values were similar, e.g., 100.00% for violations in libraries vs. somewhat lower 94.87% for violations in the project code, showing a consistent relationship with violations of manually written properties. In brief, all these FARs appear rather high.

Table 2.6 shows the FAR values for the 42 manually written properties that we inspected (we did not inspect violations of 21 properties, as explained in Section 2.4.2). First, only 11 properties (i.e., 26.19% of 42 inspected properties and 6.04% of all 182 properties) helped find a TrueBug and could have provided some value to developers of some project(s). Second, 119 properties were never violated, so they only increased the runtime overhead. These properties may get violated if monitored on other projects. Third, all but one of the properties that we inspected caused at least one FalseAlarm and the only property without false alarms, `URLDecoder_DecodeUTF8`, was violated only once. Interestingly, the property that

was violated the most and is the least effective at bug finding, `Iterator_HasNext` with 97.40% FAR, is the *de facto* example property in research papers on property mining and RV. Section 2.4.2 discusses why this property and others generate so many FalseAlarms.

For automatically mined properties, only 3 (i.e., 17.65% of the 17) led to at least one TrueBug in the 200 inspected violations—`FSM162`, `FSM33`, and `FSM373`[1], with FARs of 87.50%, 90.00% and 98.06%, respectively. `FSM373` is very similar to the manually written `Iterator_HasNext` property and has similar FAR as well. Based on the very high FARs among violations of both manually written and automatically mined properties, we conclude that the existing properties are rather ineffective for finding bugs, because they raise too many false alarms.

2.4 ANALYSIS OF RESULTS

We discuss some bugs we found, some issues with the properties (and opportunities to improve them), and some developers' responses to our pull requests (bug reports and fixes).

2.4.1 Analysis of Bugs Found

We describe some of our pull requests that the developers accepted and all three pull requests that the developers rejected so far.

Accepted Pull Requests: The project with the largest number of accepted pull requests in our study was `joda-time`, *"the* de facto *standard date and time library for Java prior to Java SE 8"* [101]. The `joda-time` developers accepted all our 40 pull requests, 37 of which based on the violations of the manually written property `ByteArrayOutputStream_FlushBeforeRetrieve` (`BAOS`). `BAOS` catches cases where an underlying `ByteArrayOutputStream` is not closed or flushed before retrieving the contents of the enclosing stream. The fix in our pull requests was simply to invoke `flush()` before `toByteArray()`, `toString()`, or `write*()` on a `ByteArrayOutputStream`. In all projects, 49 out of 55 `BAOS` pull requests that we submitted were accepted, 1 was rejected, and the others are pending.

Another big set of bugs was found from the violations of `CSC` (discussed in sections 2.1 and 2.2.2) and a closely related property, `Collections_SynchronizedMap`. These properties are violated if the `Iterator` of a synchronized `Collection` is accessed from code that is not synchronized. Our fix was to put the calling code in a synchronized block. Our pull requests for these properties were mostly accepted, or were already fixed between the start of our

[1]These properties are publicly available [160].

experiments and when we wanted to report them in widely used applications—`Spring-Beans`, `TestNG`, and `XStream`. We also have a pending pull request in `ActiveMQ`.

All the 18 bugs that we found while monitoring automatically generated tests were related to missing checks for invalid input. 17 were of the form `Type_BadParsingArgs`, where `Type` is `Long`, `Short`, or `Byte`. These properties check that calls to the respective `Type.parseType(String s, int r)` methods do not have `s` empty or `null`. 12 pull requests were accepted, 1 has been rejected, and 4 are pending. The remaining (and still pending) invalid-input-related pull request was for a violation of `InetSocketAddress_Port` property which checks that the `int` port number used to create new `java.net.InetSocketAddress` objects is between 0 and 65535, inclusive.

Finally, we found 4 bugs from monitoring the properties that Pradel et al. [159] mined automatically. Of these, 3 were duplicates of bugs found from monitoring manually written properties, so we did not report them again. The additional bug (with a pending pull request) was from a violation of `FSM33`, where `removeFirst()` was invoked on a `java.util.LinkedList` object without first checking that it was not empty.

Rejected Pull Requests: Three of our pull requests were rejected, mostly because we had limited domain knowledge. In `XStream`, we submitted a pull request for a `Collections_SynchronizedMap` violation, but the developer rejected it and responded: *"...there's no need to synchronize it... As explicitly stated in the documentation, XStream is not thread-safe during setup... this is documented behavior."* In `JSqlParser`, we reported a missing check for the validity of `s` in `Long.parseLong(String s, int i)`. The developer responded: *"...The parser itself ensures that only long values are passed to LongValue. So do you have a problematic SQL, that produces a NumberFormatException?"* Indeed, the violation was from monitoring an automatically generated test, but since the violation is in a public class, it could lead to unhandled exceptions in applications that depend on `JSqlParser` but which do not thoroughly sanitize their own input `SQL` queries. In `threerings.playn`, we submitted a fix for a `BAOS` violation and the developer responded: *"JsonAppendableWriter automatically flushes the target stream when `done()` is called, as is documented in the Javadoc for done. So an additional flush is unnecessary."* Indeed, `BAOS` did not detect the flush because the property is buggy. The violation occurred in a method which casts a `java.io.OutputStream` to `java.io.Flushable` before invoking `flush()`. However, `BAOS` was written to only track calls of `flush()` on `java.io.OutputStream` and its subtypes, whereas `Flushable` is a supertype of `OutputStream`. JavaMOP, therefore, correctly finds a violation of the property, but the property is incorrect. We submitted a bug report for `BAOS` to the JavaMOP repository and confirmed that it did not affect any other `BAOS`-related pull request that we sent.

2.4.2 Issues with Monitored Properties

We next discuss why we did not monitor some properties or inspect some violations, and give examples to show why the properties reported a lot of false alarms.

Ignored Manually Written Properties: We inspected all (652) static violations (SV) from 42 manually written properties. 21 other manually written properties had violations, but we did not inspect them: (1) 8 `*StaticFactory` properties may, at best, find performance bugs not functional bugs (459 SV); (2) 2 `*_Obsolete` properties get violated for every call to `Dictionary()` or `Enumeration()`, which were written as "suggestion" properties that should not lead to bugs (518 SV); (3) 4 `*_StandardConstructors` properties were marked as potentially reporting false alarms (430 SV); (4) 2 `Enum_*` properties were buggy and get violated on every invocation of `Enum` methods (874 SV); (5) 1 `Serializable_UID` property gets violated when a `Serializable` class does not declare a `serialVersionUID`, which can be trivially checked statically (2348 SV); and (6) 4 more properties were ignored because they did not report violation sites (93 SV). We reported 16 of these property issues, together with 7 bugs that we found in other properties during our inspections— 23 bug reports total—to JavaMOP developers.

Ignored Automatically Mined Properties: Although we originally obtained 223 mined properties from Pradel et al., we monitored only 17, because a brief manual inspection of properties found that 206 had one or more of the following issues: (1) the property (FSM) was very large, sometimes having tens of transitions and/or states, making it hard to understand and to inspect its violations; (2) the property relates only methods in the `javax.swing.*` or `java.awt.*` libraries; (3) the property imposes unnecessary temporal order on methods of multiple unrelated object types; and (4) the property imposes unnecessary temporal order on unrelated methods of the same object type. We did not report or attempt to improve the automatically mined properties. Pradel et al. [159] acknowledge that some of these properties are of low quality and develop a system to prune some violations of mined properties. However, it would be better to additionally evaluate the property mining techniques on larger, more diverse projects and confirm detected (potential) bugs with developers.

Analysis of False Alarms: The monitored properties reported many false alarms mainly because the properties (1) did not encode all correctness conditions, or encoded wrong conditions, and thus need to be improved; or (2) captured harmless misuse of APIs which would rarely or never lead to actual bugs.

For example, consider the `Iterator_HasNext` property which states that each invocation of `next()` on a `java.util.Iterator` object must be preceded by an invocation of `hasNext()` that returns `true` on the same object. `Iterator_HasNext` violations led us to discover 4 ac-

```
1 ArrayList<Integer> list = new ArrayList<>();
2 list.add(1);
3 Iterator<Integer> it = list.iterator();
4 if (it.hasNext()){ int a = it.next();}
5 if (list.size() > 0){
6  int b = list.iterator().next();
7 }
8 if (!list.isEmpty()){
9  int c = list.iterator().next();
10 }
11 HashMap<String, Integer> map = new HashMap<>();
12 map.put("one", 1);
13 if (map.containsKey("one")){
14  int d = map.values().iterator().next();
15 }
16 int e = list.iterator().next();
17 int f = map.values().iterator().next();
```

Figure 2.4: False alarms from `Iterator_HasNext` property

```
1 Map<String, String> map = new HashMap<>();
2 map.put("1", "1"); map.put("2", "2");
3 for(String key : map.keySet()) {
4  String value = map.get(key);
5  map.put(key, value + "x");
6  //map.put(key + "x", value + "x");
7 }
```

Figure 2.5: False alarms from `Map_UnsafeIterator` property

cepted bugs in the `Thomas-S-B.visualee` project and other researchers had previously used `Iterator_HasNext` to find some real bugs in `AspectJ` (bug IDs #218167 and #218171 [207]). However, `Iterator_HasNext` also reports a huge number of false alarms—150 of 154 non-HardToInspect violations were false alarms—with FAR of 97.40%. Figure 2.4 illustrates several valid invocations of `Iterator.next()`—lines 4, 6, 9, 14, 16, and 17—with no bugs in the shown code. However, `Iterator_HasNext` will be violated for all those invocations except the one on line 4. The example `next()` invocations in Figure 2.4 illustrate only a few of the valid uses of the `next()` method that were violations of the `Iterator_HasNext` property during our experiments. To make the `Iterator_HasNext` property more precise, one would need to ensure that it encodes more valid ways of checking that an `Iterator` has enough elements before invoking `next()`, taking into consideration various possible `Collection` types.

`Map_UnsafeIterator` is another property with false alarms; it checks whether code is modifying a `java.util.Map` instance while iterating over it. All 9 `Map_UnsafeIterator` violations that we inspected were false alarms. To illustrate, consider the code snippet in Figure 2.5. On each iteration, line 5 modifies the values in the `Map`—a valid operation. Nevertheless, `Map_UnsafeIterator` is violated, because it is too restrictive and reports a violation for any modification to the `Map`. If line 5 is replaced with the commented-out statement on line 6, the standard Java library would throw a `ConcurrentModificationException`. We therefore asked other JavaMOP developers (not involved in this project) why anyone would want to

monitor this property. The response reflects one challenge in coming up with effective properties: *"Invoking the put method on a map object may or may not change its key set... there is a trade-off between accuracy and simplicity... it is up to the user; one can put more effort into writing more fine-grained specs... so that there will be fewer false alarms reported; or write a simple spec easily and [then] manually eliminate the false alarms."*

Roughly 20% of all the false alarms among manually written properties were from two `Closeable_*` properties, with 113 violations between them. Both had 100% FAR. One of them catches calls of `close()` on subtypes of `java.io.OutputStream` for which `close()` is a no op. The other catches situations where calling `close()` on an `OutputStream` object that is already closed has no effect. Although both of these properties can help find developers' likely misunderstanding of the API, we classified them as FalseAlarms because they are harmless in the current version of the code. It is debatable whether we should have classified these as "code smells" as done in some prior work [63,143,159] and whether these were serious enough to submit to the developers. We could not easily change the code to avoid these problems and it is highly unlikely developers would have accepted our changes.

Automatically mined properties have similar reasons for false alarms as manually written properties. For example, `FSM373` is similar to `Iterator_HasNext`, so its false alarms were similar as well. However, one additional cause of false alarms among violations of `FSM373` was that it did not permit to call `hasNext()` multiple times successively (a self transition is missing from a state in the FSM). `FSM162` also contains transitions that are similar to `Iterator_HasNext`, but also adds in a single transition on the `Iterator.remove()` method such that the property is violated if `remove()` is called multiple times successively.

2.4.3 Developers' Responses

We discuss some example responses and comments that developers made regarding our pull requests, which gives a valuable insight into developers' perception.

Developers Asked for More: After we submitted a pull request for a `BAOS` violation, the `apache.gora` developers asked us to help check other portions of their code: *"...Are there any other instances of this behavior throughout the codebase? ...I just undertook a quick scan of the codebase for ByteArrayOutputStream, I found the following instances. Can you please check these out as well?"* Even after we fixed these other instances that they pointed out, the developers asked whether we would be interested to help with similar problems in their other codebase. In another project, `hoverruan.weiboclient4j`, we sent a pull request that fixed one of seven `Long_BadParsingArgs` violations and simply reported the other six. The

developers fixed the remaining six within a day of accepting our pull request.

Developers Viewed Pull Requests Liberally: The `joda-time` developers accepted one of our `BAOS` pull requests although they found it unnecessary: *"While I'm not convinced it is necessary, this will cause no harm."* We got similar comments for two pending pull requests. In `Apache Zookeeper`, for the `BAOS` property, the developer wrote *"Makes sense. I don't see why we shouldn't do what you suggest (add the flush). You see why it's a no-op currently though, right? (and why we haven't seen issues with this code)"*. In `TestNG`, the developer tagged one of our synchronization-related pull requests as a `perf/enhancement` and said, *"I'm not sure if it is relevant here: the lists of results should be already computed...no one is supposed to add something new at the report phase."*

Developers Accepted Better Exception Messages: For pull requests pertaining to missing checks for invalid inputs, developers responded well to the better error messages that we provided. In `IvanTrendafilov.Confucius`, the developer responded *"Looks good, I'll be happy to add that more helpful error message to the lib. Yes, please also add this check for `parseShort` and `parseByte`..."*. Similarly, in `jriecken.gae-java-mini-profiler`, the developer commented on our suggested error message *"Not sure that this is much better than the previous behavior - the exception message is a little more helpful, but it still throws a NumberFormatException"* and requested changes to our pull request before accepting it.

2.5 DISCUSSION

Our work differs from previous evaluations of properties in the runtime verification and property mining literature in three major ways. First, previous runtime verification studies mostly focused on the *efficiency* of monitoring, but we focus on the *effectiveness* question: "How good are the properties?" Second, most previous evaluations were conducted on the DaCapo benchmarks [25] (with at most 14 projects) or with a smaller number of open-source projects; in contrast, we use 200 open-source projects. Our results thus provide fresh insights to researchers in both runtime verification and property mining communities, because our evaluation is based on a substantially larger set of more diverse projects. We believe that evaluating properties on (current) open-source projects instead of (old) benchmarks can be more representative for assessing the effectiveness of properties from developers' point of view and should be strongly considered in future evaluations of properties. Third, in many previous studies, researchers assumed that any property violations were bugs, or decided themselves what was a bug or not, but we submit bug reports and fixes (i.e., *pull requests*) to let the developers be the judges of the bugs we discovered by inspecting property violations.

Inspecting property violations and submitting pull requests to developers took an estimated 1200 hours and was challenging for three reasons. First, understanding the root cause of a violation is non-trivial. Although JavaMOP reports the line number for each property violation, reasoning about a change that could correct the violation often requires deeper understanding of the code (and we were not developers on any of the 200 open-source projects); moreover, some of the violations were in third-party libraries, so we needed to comprehend parts of those libraries as well. Second, it is challenging to decide what constitutes an actual bug that should be submitted to the developers. At one extreme, we could only submit violations that can lead to program crashes. At the other extreme, we could simply submit every violation to the developers and see what they say, but this could unnecessarily burden the developers (who may then blacklist us or start to "desk-reject" our pull requests if they feel those are mostly useless to them). Even between these two extremes, it is debatable how to classify so-called "code smells" [63, 143, 159] which may indicate API misunderstanding by developers but are harmless in the current version of the code, e.g., calling `close()` on an `OutputStream` instance for which `close()` is a no op. Third, preparing a pull request in a way that developers would find useful requires substantial effort (another reason to not even attempt to submit every violation) and sometimes involved multiple internal iterations before submission. For these reasons, we reported to the developers those cases where at least one of the authors of our paper [118] believed that a violation indicated some problem in the current version of the code.

2.5.1 Recommendations for Finding Better Properties in the Future

Based on the experience from this study, we give several suggestions to help the property mining and runtime verification research communities with *property engineering*, i.e., writing/discovering and evaluating more effective properties.

(1) Increased Focus on Bug-Finding Effectiveness: More focus should be on the bug-finding effectiveness of properties, which is more important to developers than the performance of monitoring. For example, the most widely used `Iterator_HasNext` property was highly ineffective for finding bugs, matching the finding by Thummalapenta and Xie [198].

(2) Better Property Categorization: It is crucial to find good ways to designate the severity levels of properties. All properties are not equal in their bug-finding effectiveness. Some properties, when violated, indicate a bug with a very high probability. Other properties indicate issues that may be bugs in some projects but not in others. Finally, some properties are less severe, indicating poor coding practices and may not lead to detecting true bugs.

(3) Complementing Benchmarks: Continued use of benchmarks like DaCapo is good for

comparison with older results and evaluating performance of new techniques, but benchmarks should be complemented with evaluations on a larger number of open-source projects, to assess the techniques and properties in more realistic scenarios.

(4) Confirming Detected Bugs with Developers: Evaluating on recent project versions and reporting detected bugs to developers of open-source projects should be encouraged more. Admittedly, the process is challenging and time consuming, requiring domain knowledge and communication with developers. We publicly released a list of all our pull requests, to serve as a starting point for collecting true bugs: [189]. We found interesting results from submitted pull requests, e.g., even "buggy" properties like `BAOS` led to accepted pull requests

(5) Automated Filtering of Properties and False Alarms: It is necessary to better automatically filter out likely false alarms to improve ineffective properties. We found that properties with too many violations were almost always ineffective. Pradel et al. [159] defined heuristics-based automated techniques for filtering out violations when statically checking mined properties. Gabel and Su [64], and Nguyen and Khoo [143] proposed techniques for checking that mined properties are true properties. More work in this direction is needed, especially because manual inspection, which we did in this dissertation, is rather tedious.

(6) Open Property Repositories: It would be beneficial to have community-driven property repositories and standardized ways of representing properties to facilitate property sharing—we could have evaluated more properties if it were easier to find and use them. We started such a repository using all the properties monitored in this dissertation [161]; we plan to continue adding more properties to this repository, and invite the research community to contribute their properties there as well to facilitate research on engineering better properties.

2.6 THREATS TO VALIDITY

External: The results of our study may not generalize beyond the projects, tests, or properties that we evaluated. To mitigate this threat, we used a larger number of open-source projects than had been evaluated in previous runtime verification and property mining studies. Further, the 200 projects that we used were quite diverse in size, number of tests, and GitHub activity. Concerning the bug-finding effectiveness of properties, we used the largest sets of manually written and automatically mined properties that we could find with our mini-survey of the property mining and runtime verification literature, and which could easily work with JavaMOP. JavaMOP is representative of the performance of runtime verification tools and allows to simultaneously monitor properties written in different formalisms, making it well suited for our large-scale evaluation of existing properties. Our study is focused on Java and the results may differ for other programming languages.

Internal: We wrote scripts to automate the monitoring of tests against the properties. Our scripts that run the tests, measure overhead, and post-process results were reviewed by at least two authors of our paper [118]. During inspection and classification of violations into TrueBug, FalseAlarm, and HardToInspect, we initially had two reviewers inspect independently to prevent them from influencing each other. Some violations that we labeled as FalseAlarms may be TrueBugs. For violations that we labeled as TrueBugs, we submitted 95 pull requests and developers decide whether to accept (74 so far) or reject (3 so far).

2.7 SUMMARY

This chapter motivated the need for techniques that can reduce the overhead of performing RV during software testing. The large-scale study of monitoring properties during testing showed that runtime overhead is high and developers have to wait for and manually inspect the many violations that RV generates. However, although software changes very frequently and developers perform regression testing after every code change, there were no prior techniques to optimize RV across multiple program versions. Therefore, we proposed to make RV evolution aware [115] to reduce the overhead of performing RV during regression testing. The rest of this dissertation presents contributions on evolution-aware RV.

CHAPTER 3: STATIC CHANGE-IMPACT ANALYSIS AND RTS

Evolution-aware RV reduces the costs of performing RV during regression testing by focusing RV and its users on parts of code affected by changes. Therefore, change-impact analysis [11, 31, 36, 120, 126, 169, 170, 176, 179, 220, 221], which is concerned with computing parts of code that are affected by code changes, is a central component of evolution-aware RV. This chapter describes static change-impact analyses that we implemented and evaluated in the context of regression test selection (RTS). Evaluating the static change-impact analyses in the context of RTS is beneficial to evolution-aware RV: we obtained an RTS tool that enabled us to compare and combine evolution-aware RV with RTS—a well-known, evolution-centered regression testing technique (Section 5.4.3). Note also that change-impact analysis has been traditionally often evaluated in the context of RTS [107, 114, 169, 176, 221].

We chose static change-impact analysis because RV is already a dynamic analysis. We expect a dynamic change-impact analysis to incur more overhead than a static change-impact analysis when combining RV with testing. Since change-impact analysis can be performed at different levels of program granularity, we compare static RTS based on method- and class-level static change-impact analyses. To evaluate the quality of static change-impact analysis, we compare static RTS based on static change-impact analysis with Ekstazi, the state-of-the-art dynamic class-level RTS approach [68], which was previously shown to outperform a method-level dynamic RTS approach [69]. The results show that static RTS based on class-level change-impact analysis performs significantly better than static RTS based on method-level change-impact analysis. Class-level static RTS also performs similarly as Ekstazi—the static change-impact analysis was very fast but led to static class-level RTS selecting more tests than Ekstazi, such that both RTS techniques have about the same end-to-end time.

3.1 BACKGROUND ON CHANGE-IMPACT ANALYSIS

Performing change-impact analysis to reason about how changed code can affect the behavior of other (unchanged) parts of code has long been studied for various software analysis and maintenance tasks. Cai and Santelices [36], Lehnert [120], and Li et al. [126] provide detailed overviews of the change-impact analysis literature. In this dissertation, we categorize existing change-impact analysis techniques based on when they are applied, whether they are static or dynamic, and the program granularity at which analysis is done. We highlight here some existing work in each of these categories, and provide a rationale for the choice of change-impact analysis that we made.

Change-impact analysis can be performed before or after making code changes. In this dissertation, we are concerned with performing change-impact analysis *after* code changes are made. A different and important line of research exists on predictive change-impact analysis that is performed *before* making code changes [34–36, 65, 66, 104, 127, 177]. Change-impact analysis can be performed statically [31, 106, 114, 176] or dynamically [7, 109, 110, 147, 169]. We investigate static change-impact analysis as a basis for evolution-aware RV because we expect dynamic change-impact analysis to be more costly. Further, the instrumentation needed for dynamic change-impact analysis, however lightweight, may interfere with the instrumentation that the RV tool, JavaMOP, performs. Finally, change-impact analysis has been applied at various program granularity levels and program representations, including statement level [197], method level [169,176], class level [106,107], program slice [23,148], program dependence graph [18], control-flow graph (CFG) [114], and whole-program path [110].

In the absence of existing empirical results on comparing the performance and quality of all these granularity levels, we compared RTS based on class- and method-level change-impact analysis for two reasons. First, recent work on dynamic RTS [67,69] showed that performing RTS at class level provided a better end-to-end time than RTS at the method-level [220], but no such comparison was previously performed for static RTS. Secondly, due to the growing scale of modern software systems, other researchers have proposed to use coarser granularity levels (i.e., methods or classes) rather than finer granularity levels (e.g., statements or CFG edges), which are more expensive to collect [46].

3.2 BACKGROUND ON REGRESSION TEST SELECTION

Modern software projects evolve rapidly as developers add new features, fix bugs, or perform refactorings. To ensure that software evolution does not break existing functionality, developers commonly perform regression testing. However, frequent re-running of full regression test suites can be extremely time consuming. Some test suites require weeks to run [175], but waiting even a few minutes for test results can be detrimental to developers' workflow. In addition to reducing developers' productivity, slow regression testing can consume a lot of computing resources. For example, engineers at Google observed a quadratic increase in their total test-running times [50, 74, 216], showing that regression testing is challenging, even for a company with a lot of computing resources. As a result, a large body of research has been dedicated to reducing the costs of regression testing, using approaches such as regression test selection [69, 81, 149, 169, 174, 211, 220], regression test-suite reduction [80, 180, 185, 222, 225], regression test-case prioritization [47, 79, 175, 217, 219], and test parallelization [22]. Yoo and Harman provide a thorough survey of regression testing approaches [215].

Regression test selection (RTS) is the most widely used approach to speeding up regression testing [50]. RTS aims to reduce regression testing efforts by only re-running the tests affected by code changes. An RTS technique is *safe* if it selects all tests whose behavior may be affected by code changes; not running any of those tests may cause developers to miss regressions. Prior research on RTS can be broadly split into *dynamic* and *static* techniques.

A typical dynamic RTS technique requires two types of information: (1) changes between two code versions, and (2) test dependencies dynamically computed while running the tests on the old code version. Given these inputs, the technique analyzes how the code changes interact with the dependencies to determine a subset of tests that may execute (and thus get affected by) the code changes. Dynamic and safe RTS has been drawing attention in the literature since at least 1993 [173, 215], with some newer techniques such as DejaVOO [149], FaultTracer [220], and Ekstazi [69]. Different dynamic RTS techniques differ in safety, precision and analysis overhead [226]. Safe techniques always select all tests affected by code changes but could also select non-affected tests. Precise techniques select only affected tests; they do not select any non-affected tests. Techniques that collect finer-granularity dependencies may be more precise, selecting fewer tests to be run, but can incur higher analysis overhead; in contrast, techniques that collect coarser-granularity test dependencies may be less precise but can have lower analysis overhead.

The state-of-the-art dynamic RTS technique, Ekstazi [67–69], tracks changes and dependencies at the granularity level of files; for Java code, these files include bytecode classes. Ekstazi computes (1) classes which changed between versions, and (2) classes that each test class required while running on the old code version. Ekstazi selects to run on the new code version only tests that depend on at least one of the changed classes. Prior experiments with Ekstazi showed that using coarse-granularity test dependencies (at the class level) can substantially save the end-to-end testing time (that includes time to analyze changes, run selected tests, and update dependencies) [69]. Due to this, several open-source projects (e.g., Apache Camel [4], Apache Commons Math [5], and Apache CXF [6]) have already incorporated Ekstazi into their build systems [69].

Despite the recent advances in dynamic RTS, its reliance on dependencies collected dynamically could limit its application in practice, making it important and timely to reconsider static RTS. First, when performing RTS, dynamic test dependencies for the old version may not always be available, e.g., on the first application of RTS to the project (the project may have earlier versions). Second, dynamic test dependencies for large projects may be time-consuming to collect. Third, for real-time systems, dynamic RTS may not be applicable, because code instrumentation for obtaining dependencies may cause timeouts or interrupt normal test run. Finally, for programs with non-determinism (e.g., due to randomness or

concurrency), dependencies collected dynamically may not cover all possible traces, leading to dynamic RTS being unsafe.

In contrast to dynamic RTS, which collects test dependencies dynamically, static RTS [11, 31,107,176] uses static program analysis to infer an over-approximation of the test dependencies to enable safe test selection. However, although static RTS techniques for object-oriented languages have been proposed over three decades ago [31,107], to our knowledge, these techniques have not been studied extensively on modern, real-world projects. In particular, it is not clear *a priori* which granularity level would be better for *static* RTS.

To investigate the safety, precision, and overhead of static RTS, we implemented one class-level static RTS technique and one method-level static RTS technique. The class-level static RTS technique (*ClassSRTS*) uses our implementation of the *class firewall* [107, 125, 149,187,188] for change-impact analysis; it finds class-level dependencies by reasoning about inheritance and reference relationships in a class dependency graph. ClassSRTS selects to re-run test classes that transitively depends on a changed class in the dependency graph. The method-level static RTS technique (*MethSRTS*) uses a call-graph based change-impact analysis [11, 73, 200]; it constructs a call graph with all test *methods* as entry points and selects to re-run test *classes* that can transitively reach a changed class through a traversal of the call graph. The ClassSRTS and MethSRTS implementations discussed in this chapter are based on the ASM bytecode manipulation and analysis framework [10] and the T.J. Watson Libraries for Analysis (WALA) [205], respectively. Chapter 4 discusses an improved implementation of ClassSRTS in a tool called STARTS.

We evaluated these two static RTS techniques on 985 versions of 22 open-source Java projects. We considered two variants of ClassSRTS and eight variants of MethSRTS, and we compared them against Ekstazi. The results show that ClassSRTS has comparable performance as Ekstazi, but ClassSRTS is occasionally unsafe. In contrast, MethSRTS performs rather poorly: it does not provide performance benefits and is more frequently unsafe. The latter result was somewhat surprising as one may expect finer-grain analysis at the method level to be safer and more precise (but potentially slower) than the coarser-grain analysis at the class level (sections 3.3.3 and 3.6 give some explanation of this surprising result).

Concerning safety, our experiments show that reflection was the reason why ClassSRTS missed tests that Ekstazi selects in a small number of cases (Section 3.6). Therefore, we have begun an orthogonal but complementary line of research to evaluate whether we could make ClassSRTS safe with respect reflection [183], which we discuss Chapter 6. In conclusion, we recommend that researchers continue improving static RTS techniques at the coarser granularity, which already shows promising results (at least at the level of classes if not modules or projects).

3.3 CHANGE-IMPACT ANALYSIS FOR REGRESSION TEST SELECTION

We introduce the two static change-impact analysis techniques that we implemented at different granularity levels and the static RTS techniques that they support: ClassSRTS uses class-level change-impact analysis [107] (Section 3.3.1) and MethSRTS uses method-level change-impact analysis [11, 73, 170, 176, 200] (Section 3.3.2). Although they are based on change-impact analyses at different granularity levels, we implemented the RTS techniques to report selected test *classes* to aid comparison. In the rest of this chapter, when we refer to a *test*, we mean a test class. MethSRTS could, in principle, report only selected test *methods*. Recent surveys on regression testing [215] and change-impact analysis [120, 126] provide more details about static and dynamic RTS.

3.3.1 Change-Impact Analysis for Class-Level Static RTS (ClassSRTS)

Leung et al. [125] first introduced the notion of *firewall* to assist testers in focusing on code modules that may be affected by program changes. Kung et al. [107] further introduced *class firewall* to account for the characteristics of object-oriented languages, e.g., inheritance. Given a set of changed classes, a class firewall computes the set of classes that may be affected by the changes, conceptually building a "firewall" around the changed classes. The original class firewall technique was proposed for the object-relation graph in C++ [107], and Orso et al. [149] generalized it to the *intertype relation graph* (IRG) to additionally consider interfaces in Java. Subsequently, we use *types* to denote classes and interfaces. To the best of our knowledge, using the IRG and the class firewall is the only proposed technique to perform class-level static RTS in Java.

An IRG represents the use and inheritance relations between types in a program, as defined by Orso et al. [149]:

Definition 3.1 (intertype relation graph)**.** *An intertype relation graph, IRG, of a given program is a triple* $\langle N, E_I, E_U \rangle$ *where:*

- N *is the set of nodes representing all types in the program;*
- $E_I \subseteq N \times N$ *is the set of inheritance edges; there exists an edge* $\langle n_1, n_2 \rangle \in E_I$ *if type* n_1 *inherits from* n_2*, and a class implementing an interface is in the inheritance relation;*
- $E_U \subseteq N \times N$ *is the set of use edges; there exists an edge* $\langle n_1, n_2 \rangle \in E_U$ *if type* n_1 *directly references* n_2*, and aggregations and associations are in the use relations.*

Based on this definition of IRG, the class firewall is defined as the types that can (transitively) reach some changed type through use or inheritance edges:

Definition 3.2 (Class Firewall)**.** *The class firewall corresponding to a given set of changed types $\tau \subseteq N$ is computed over the IRG $\langle N, E_I, E_U \rangle$ using the transitive closure of the dependence relation $D = (E_I \cup E_U)^{-1}$; $firewall(\tau) = \tau \circ D^*$, where $^{-1}$ denotes the inverse relation, * denotes the reflexive and transitive closure, and $\circ$ is the relational product.*

ClassSRTS takes as input the two program versions and the regression test suite T that consists of the tests for the new version. The output is the subset of tests $T_s \subseteq T$ that may be affected by the changes. ClassSRTS first builds an IRG, computes the changed types between two program versions, and adds the transitive closure of each changed type to the class firewall. Finally, ClassSRTS returns all tests in the class firewall as the selected test set T_s. Note that ClassSRTS need not include supertypes of the changed types (but must include all subtypes) in the transitive closure because a test cannot be affected statically by the changes even if the test reaches supertype(s) of the changed types unless the test also reaches a changed type or (one of) its subtypes.

3.3.2 Change-Impact Analysis for Method-Level Static RTS (MethSRTS)

A program call graph (CG) represents invocation relationships among program methods [200]. Intuitively, starting from each root method (e.g., the `main` method), the call-graph construction finds all methods that can be (transitively) invoked from the root method. A call graph is defined as follows:

Definition 3.3 (Call Graph)**.** *A call graph CG of a given program is a pair $\langle N, E \rangle$, where:*

- *N is the set of all methods in the program under analysis;*
- *$E \subseteq N \times N$ are the method invocation edges.*

MethSRTS takes as input the two program versions and the regression test suite. The output is T_s, the subset of tests that may be affected by the changes. A test is affected if any of its test methods is affected. (Our experiments show that MethSRTS is rather slow, not because it selects to run test classes instead of test methods, but because the analysis itself is slow.) Further, changes are computed at the *class* level rather than the *method* level because recent work [69] demonstrated that class-level changes do not require complex modeling of changes due to dynamic dispatch (e.g., using *lookup* changes [170,176]), and can be extremely fast to compute. MethSRTS first builds a call graph for the old version using as root methods all public methods (e.g., including the "`@Before`" and "`@After`" methods) in the test suite. Then, MethSRTS iterates over each test to check if any of its methods may be affected by the changed types τ, i.e., if it can (transitively) reach methods in τ. Finally, MethSRTS returns the selected set of tests.

```
1 //library code
2 class L {
3   void m1() {}
4 }
5
6 //source code
7 class C1 extends L {
8   void m1(){
9     C2.m3()
10  }
11  void m2(){}
12 }
13
14 class C2 {
15  static void m3(){}
16 }
```

```
1 //test code
2 class T1 {
3   void t1() {
4     L l = new L();
5     l.m1();
6   }
7 }
8 class T2 {
9   void t2() {
10    L l = new C1();
11    l.m1();
12  }
13 class T3 {
14  void t3() {
15    C1 c = new C1();
16    c.m2();
17  }
18 }
```

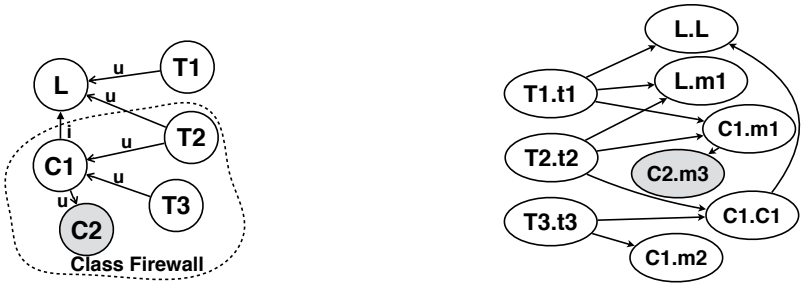
Figure 3.1: Example code

3.3.3 Example

Although MethSRTS performs method-level change-impact analysis, a finer level of granularity than ClassSRTS, it does not necessarily deliver more precise RTS. To illustrate both techniques, consider the example in Figure 3.1. Class `L` is in a library, while classes `C1` and `C2` are in the code under test. `T1`, `T2`, and `T3` are three tests. Suppose `C2` is modified (in gray). An dynamic RTS technique (e.g., Ekstazi), will select to re-run only `T2`, the only test that executes the modified `C2`. We discuss the RTS results for ClassSRTS and MethSRTS.

ClassSRTS: Figure 3.2(a) shows the IRG for the example. Edges labeled "`u`" and "`i`" are use and inheritance edges, respectively. The class firewall (enclosed in the dashed area) consists of all classes that can potentially reach the modified class, `C2`. Tests `T2` and `T3` are in the class firewall and thus selected. Note that `T3` is selected due to the imprecision of the class-level analysis: although `T3` uses `C1`, it does not invoke any method of `C2`.

MethSRTS: Figure 3.2(b) shows the call graph for the example. The method `C2.m3` in the modified class `C2` is marked in gray. From the call graph, tests `T1` and `T2` can reach `C2.m3` and are thus selected. Although MethSRTS can be more precise than ClassSRTS in determining that `T3` cannot invoke `C2.m3`, MethSRTS incurs another imprecision—even advanced call-graph analyses cannot always precisely determine receiver object types. For example, `T1` invokes `m1` with the static receiver type `L`; a naive call-graph analysis (e.g., CHA) treats all subclasses of `L` as potential runtime receiver types. So, MethSRTS may imprecisely select `T1` that could invoke `C1.m1` because `C1` extends `L`. ClassSRTS does not have this issue; a runtime object type is referenced by the test that instantiates it.



(a) IRG (b) Call graph

Figure 3.2: Example IRG and call graph

3.3.4 Analysis Scope

Modern software projects use external libraries extensively which can slow down static analysis in general and MethSRTS in particular. For example, in one version pair of the project `invokebinder` (p1, Table 3.5), we found that using RTA call-graph analysis takes 370.8% of RetestAll time when third-party libraries are excluded from the analysis, but it becomes 81304.2% of RetestAll time when libraries are included—orders of magnitude difference! Therefore, we consider the impacts of excluding third-party libraries for ClassSRTS and MethSRTS as follows.

Theorem 3.1. *Excluding third-party libraries* cannot *introduce a new type of safety issue for ClassSRTS.*

Proof. Excluding third-party libraries removes all types in the library from the IRG. If the exclusion induced a safety issue, there must be a path from a test `T` to a changed type `C` that contains a library type `L`; otherwise, the exclusion will not impact the selection results. This implies a path from `T` to `L` and a path from `L` to `C`. However, third-party libraries are built before the code under test. Therefore `L` cannot statically inherit from or use `C`[1]. Therefore, the path from `L` to `C` does not exist. Contradiction. □

Theorem 3.2. *Excluding third-party libraries* can *introduce a new type of safety issue for MethSRTS.*

Proof. As shown in Figure 3.2, by including library code MethSRTS can select the truly affected test `T2`. However, if the library is excluded, Line 11 in Figure 3.2(b) will be ignored by the call-graph analysis because the static receiver type is `L`. Then, `T2` only reaches `C1.C1()` that does not invoke any methods from the code under test. Thus, `T2` does not reach any

[1]Note that even when there are callbacks from the library code, the static reference to the receiver type of the callback is usually referenced by some class from the code under test to pass to the library code; also note that the safety issues of static RTS caused by reflection (shown in Section 3.6) exist even with library code analysis.

method in the modified class `C2` and would not be selected. Therefore, excluding third-party libraries can introduce a new type of safety issue for MethSRTS. □

As a result, we exclude libraries for ClassSRTS (making the analysis run faster while selecting the same set of tests), but we evaluate MethSRTS both with and without library exclusion to investigate the cost/safety tradeoff.

3.4 IMPLEMENTATION

We describe the implementation details of our ClassSRTS and MethSRTS techniques. Ekstazi, the dynamic RTS tool used in our evaluation, is described elsewhere [68, 69].

Change computation: Finding syntactically changed source files can be done easily using the `diff` utility or version-control systems, but syntactic changes (e.g., simple reformatting) may not translate to bytecode changes [69]. Therefore, we compute changes at the bytecode level, leveraging the comparison utility from Ekstazi. More specifically, given two program versions, the comparison first detects the bytecode files that differ between the versions, and then invokes the Ekstazi API to compute *smart checksums* [69] (by removing debug-related information) of those bytecode files to further filter out the files where only debug-related information changed. Using the Ekstazi change computation also enables a fairer evaluation and comparison of tools.

Graph construction: For ClassSRTS, we used the ASM bytecode manipulation framework (version 5.0) [10] to construct the IRG. Our tool uses ASM to parse the bytecode of each (changed) classfile, traversing all the fields, methods, signatures, and annotations to collect all types that are referenced/used by the type in the classfile. It also collects all types that the type in the classfile extends/implements. Importantly, it incrementally updates the IRG computed from a prior version by analyzing only the classfiles that changed.

For MethSRTS, we used the call-graph analyses from the IBM WALA framework [205]. We evaluated four widely used call-graph anlayses: CHA (Class Hierarchy Analysis), RTA (Rapid Type Analysis), 0-CFA (Control-Flow Analysis), and 0-1-CFA, in the ascending order of precision [73, 200]. The analyses effectively differ in how they approximate the runtime types of receiver objects, e.g., CHA does not approximate the runtime types at all, while 0-CFA uses one set of types to approximate the runtime types. In general, a more precise call-graph analysis may incur a higher overhead. Furthermore, WALA also allows excluding the library code from the analysis to speed it up. Therefore, we studied these four analyses both with and without library exclusion to investigate the cost/safety tradeoffs for MethSRTS. We used *0-CFA with library exclusion* as the default MethSRTS variant

because (1) it is recommended by the WALA tutorial [205] for general call-graph analysis applications, and (2) our results show it to perform well among all the eight variants for the specific application of call-graph analysis to RTS. Both tools construct the appropriate graph (IRG or call graph) on the *old* program version and serialize the constructed graph to the disk for the new version.

Graph traversal: Given an appropriate graph (IRG or call graph) and a set of changed nodes, each tool needs to find the tests that can reach the changed nodes. Our tools always traverse the graph representing the old version. For ClassSRTS, we evaluate two modes, offline and online, that traverse the (old) graph at different points. The *offline* mode computes transitive closure of the entire graph in advance (before the new version and the changes are known) and produces a mapping from test to dependencies; the time to compute the closure *is not* counted in the end-to-end time. The selection then simply checks what test has some changes among its dependencies. The *online* mode computes only nodes transitively reachable from the changes once it knows what those changes are; the time to compute reachability *is* counted in the end-to-end time. Both modes also incrementally update the old graph to produce a new graph for the next version; the time to perform the update *is not* counted for offline, while it *is* counted for online. For MethSRTS, all eight variants use the online mode. We did not try the offline mode for MethSRTS because (1) MethSRTS performs poorly in terms of safety and precision; it is not worth further cost analysis, and (2) MethSRTS selects many more tests than Ekstazi/ClassSRTS so its offline mode can be predicted to be inferior to both others. To implement graph traversals, we use JGraphT [97].

3.5 EVALUATION AND RESULTS

We present our experimental setup and the results of evaluating static RTS techniques in terms of number of selected tests, time overhead, precision, and safety. We evaluate two variants of ClassSRTS and eight variants of MethSRTS. We compare these variants with two baselines: *RetestAll* (which just runs all tests) and Ekstazi. Finally, we present some examples of the safety and precision issues of static RTS.

3.5.1 Experimental Setup

To evaluate static RTS, we use 22 open-source projects, listed in Table 3.1. We chose these projects among single-module Maven projects with JUnit 4 tests from (1) the original Ekstazi paper [69], (2) one of our previous studies [115], and (3) popular GitHub Java projects with longer-running tests (which we deliberately chose because they are more likely to benefit

Table 3.1: Projects used in study

ID	Project Name	SHA	kLOC	Revs	Tests	T[s]
p1	invokebinder	8611721	2.0	66	2.2	1.7
p2	logback-encoder	4fe0f4a	3.2	43	18.7	3.4
p3	compile-testing	8d5229e	3.0	30	7.6	3.7
p4	commons-cli	3ba638a	5.9	50	23.0	3.8
p5	commons-dbutils	1429538	5.4	33	23.2	4.1
p6	commons-fileupload	1460343	4.3	54	12.0	4.8
p7	commons-validator	bcb1ec4	11.9	19	61.0	4.8
p8	asterisk-java	08dda72	34.5	59	38.1	6.1
p9	commons-codec	50a1d17	17.0	63	47.5	6.5
p10	commons-compress	ec07514	32.5	12	89.4	9.4
p11	commons-email	1607174	6.5	23	17.0	12.3
p12	commons-collections	1543740	54.3	66	149.6	19.9
p13	commons-lang	bcb33ec	69.0	61	133.8	21.6
p14	commons-imaging	b1fdec9	37.1	87	58.9	28.9
p15	commons-dbcp	1587107	18.7	31	27.2	68.9
p16	b.HikariCP	19e0c5d	9.4	49	21.0	80.2
p17	commons-io	1686461	27.7	49	93.9	91.4
p18	addthis.stream-lib	4dc3705	8.3	5	24.0	104.8
p19	commons-math	79c4719	185.4	57	450.2	109.3
p20	OpenTripPlanner	aa21c92	79.3	20	135.8	277.9
p21	commons-pool2	1622091	12.8	51	19.5	294.6
p22	jankotek.mapdb	eac22b7	67.9	57	144.2	515.9
	Average		32.3	44.8	80.2	75.8

from RTS). Table 3.1 shows basic statistics for the projects: SHA is the initial version of the project on which our experiments started, kLOC is the number of thousands of lines of code in the project, Revs is the number of version pairs used in our evaluation, Tests is the number of tests in the project, and T[s] is the time in seconds to run all the tests in each project. kLOC, Tests, and T[s] are averages across all versions that we used.

We used Ekstazi off-the-shelf to compare with static RTS techniques. To select the versions, we followed the methodology from the original Ekstazi paper [69]. For each project, we started with the 100 versions immediately preceding SHA and then chose the subset of those 100 versions (1) that compiled, (2) for which `mvn test` ran successfully, and (3) for which Ekstazi ran successfully with `mvn ekstazi:ekstazi`. Starting from the oldest version in our list, we ran each RTS technique on the successive pairs of versions, simulating what an end user would have experienced when using an RTS tool. We measure the number of tests selected by each technique, the time taken to run the selected tests, and the time taken by RTS techniques to perform the selection. The end-to-end time includes the time to collect dependencies, analyze what tests to run, and to actually run the selected tests for the online variants of Ekstazi and the static RTS techniques. For all the offline variants, the time

to collect dependencies is not included. We ran all experiments on a 3.40 GHz Intel Xeon E3-1240 V2 machine with 16GB of RAM, running Ubuntu Linux 14.04.4 LTS and Oracle Java 64-Bit Server version 1.8.0_91.

3.5.2 Research Questions

Our study aims to answer these research questions:

- **RQ1:** How do static RTS techniques compare among themselves and with RetestAll and dynamic RTS in terms of the number of tests selected to run?
- **RQ2:** How do static RTS techniques compare among themselves and with RetestAll and dynamic RTS in terms of runtime?
- **RQ3:** How do static RTS techniques compare with a safe class-level dynamic RTS in terms of precision and safety?
- **RQ4:** How do different variants of the MethSRTS influence the cost/safety trade-offs?

3.5.3 RQ1: Tests Selected

Figure 3.3 shows the percentage of tests selected by static and dynamic RTS relative to RetestAll (the black line in the middle of each boxplot is the mean). On average, Ekstazi, ClassSRTS, and MethSRTS select to run 20.6%, 29.4%, and 43.8% of all tests, respectively. As expected, ClassSRTS and MethSRTS both tend to select a higher percentage of tests than Ekstazi. We further discuss exactly how precise and safe ClassSRTS and MethSRTS are with respect to Ekstazi in Section 3.5.5. Surprisingly, the coarser-granularity ClassSRTS technique selects *fewer* tests than the finer-granularity MethSRTS technique. One reason was discussed in Section 3.3.3, and more concrete cases are analyzed in Section 3.6. Overall, although inferior to the state-of-the-art dynamic RTS in terms of the number of tests selected, static RTS still selects only a fraction of all tests.

3.5.4 RQ2: Time Overhead

While the number of selected tests is an important internal metric in RTS, the time taken for testing is the relevant external metric because a developer using RTS perceives it based on this time. We measure time from the point of view of a developer who commits/pushes some code changes and then waits for test results before proceeding with other tasks. Specifically, we follow the original Ekstazi experiments [69] and compare static and dynamic RTS in terms of the *end-to-end time* that includes time to (1) analyze what tests should be run,

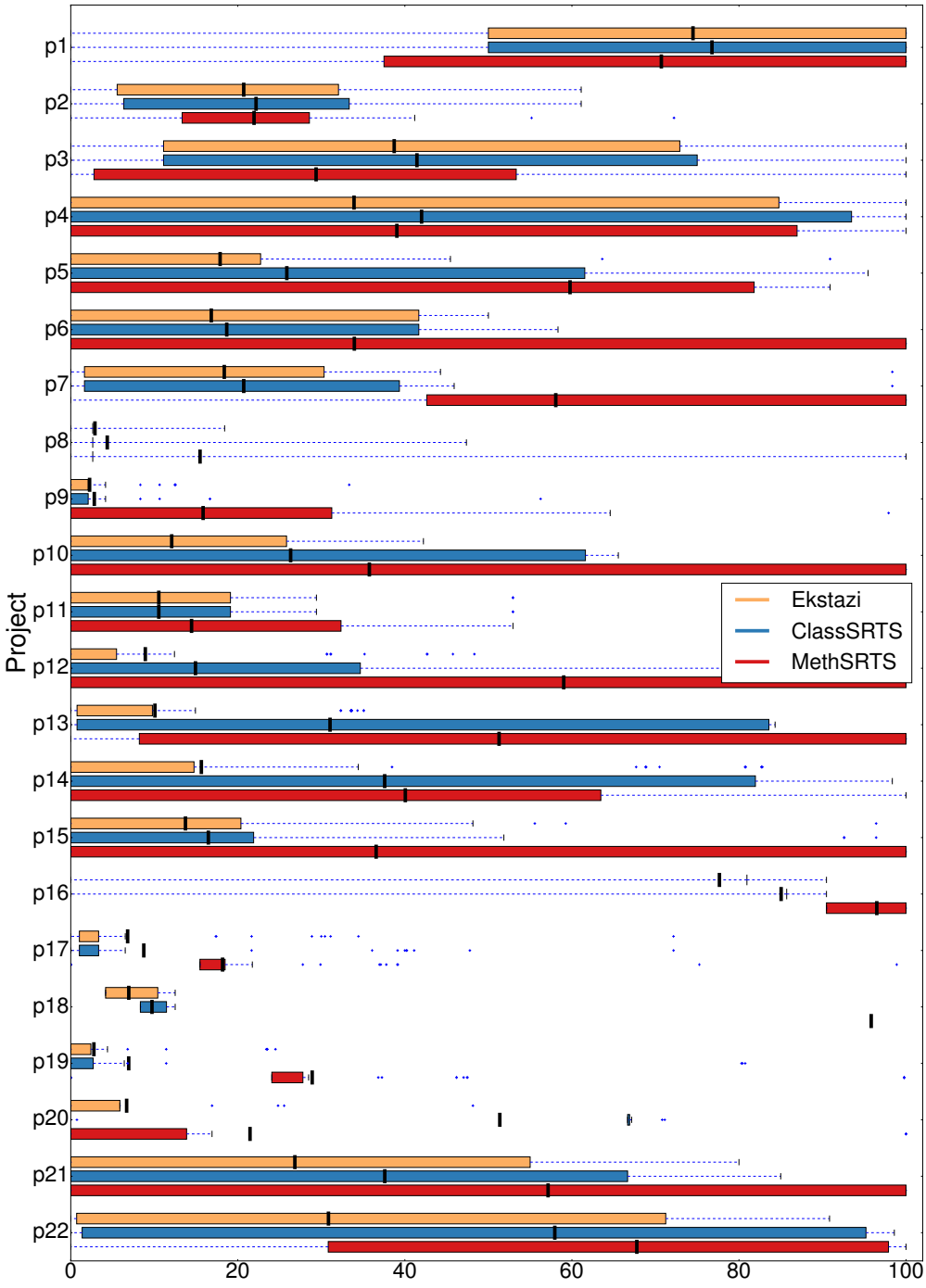


Figure 3.3: Percentage of tests selected

(2) run those tests, and (3) collect dependencies for future RTS[2]. For any RTS technique to be beneficial, this end-to-end time must be less than RetestAll time.

Table 3.2 summarizes the end-to-end times for static and dynamic RTS relative to RetestAll. Columns Ekstazi (offline), Ekstazi (online), ClassSRTS (offline), ClassSRTS

[2]Note that the dependency collection time is not counted in the end-to-end time of the offline mode.

Table 3.2: Summary of end-to-end testing time relative to RetestAll

	Ekstazi (offline)			Ekstazi (online)			ClassSRTS (offline)			ClassSRTS (online)			MethSRTS		
Project	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg
p1	92.4	132.7	115.5	88.1	147.3	126.2	72.4	113.8	102.9	75.7	130.9	114.8	215.1	520.8	390.5
p2	82.9	122.3	105.4	85.4	146.2	121.6	64.4	114.0	90.9	65.8	127.3	99.5	190.7	270.9	220.6
p3	74.9	122.6	103.0	76.4	139.0	113.9	65.0	109.6	91.8	64.3	124.5	97.8	130.7	284.4	227.7
p4	92.4	119.4	106.7	97.0	131.2	114.3	83.0	117.7	99.3	83.0	125.0	102.2	167.1	250.3	202.5
p5	81.2	112.8	101.4	87.2	129.1	106.2	73.7	116.7	94.5	80.7	116.9	97.7	156.9	282.9	231.7
p6	73.6	124.0	93.4	70.2	148.3	98.3	61.3	114.6	85.7	62.0	125.3	87.6	123.0	230.1	175.9
p7	71.5	107.0	88.9	75.3	114.4	95.0	62.2	97.9	77.5	61.2	106.2	82.9	131.9	15688.0	9305.2
p8	31.6	109.6	44.4	31.7	117.2	49.0	25.5	104.5	42.9	24.8	330.2	54.1	85.2	458.6	352.8
p9	41.7	93.2	56.4	45.4	96.0	58.2	35.6	80.7	49.6	36.0	91.9	51.3	92.6	258.3	166.9
p10	43.6	91.3	54.1	46.1	108.9	57.9	34.7	105.0	64.2	35.6	110.2	65.3	98.2	253.1	165.8
p11	29.5	99.4	49.9	28.3	103.2	50.1	24.4	105.9	44.8	23.5	105.6	47.1	51.7	155.6	86.5
p12	28.4	101.8	44.0	26.6	109.8	45.5	25.7	102.4	51.3	25.5	252.4	55.8	61.3	4675.5	2670.4
p13	29.4	74.7	44.7	30.6	83.0	47.6	25.2	99.2	55.6	25.9	103.5	57.1	58.7	444.2	316.8
p14	20.4	104.2	46.1	23.1	111.9	48.9	19.0	107.2	54.5	19.3	107.4	55.7	39.3	136.2	81.7
p15	6.5	104.7	26.4	6.7	107.5	27.0	5.6	104.9	26.0	5.6	103.4	26.1	12.4	123.8	57.3
p16	35.1	99.8	94.5	10.9	105.3	97.0	2.8	98.1	68.4	3.0	98.8	66.9	7.8	120.5	80.0
p17	4.1	96.6	19.9	4.1	109.6	21.3	3.4	94.9	22.0	3.4	96.4	22.3	9.7	113.4	46.2
p18	31.6	45.1	35.3	31.9	46.1	36.1	29.9	43.6	34.7	30.4	44.5	34.6	103.6	109.5	107.4
p19	10.9	69.4	17.8	11.3	96.7	20.6	9.6	97.5	19.4	9.7	98.0	19.9	21.9	187.1	112.1
p20	47.8	90.9	67.1	47.9	100.7	71.0	47.0	99.3	85.9	47.1	100.5	86.2	85.8	149.7	112.5
p21	1.1	101.5	52.4	1.1	101.8	52.5	0.9	100.7	59.8	0.8	100.8	59.7	2.2	102.6	68.2
p22	0.9	102.4	41.1	0.9	104.3	42.0	0.8	87.5	52.7	0.7	89.3	53.2	1.6	177.6	117.4
Average	42.3	101.2	**64.0**	42.1	111.7	**68.2**	35.1	100.7	**62.5**	35.6	122.2	**65.3**	84.0	1136.0	**695.3**

(online), and MethSRTS represent the time for Ekstazi in offline and online modes, ClassSRTS in offline and online modes, and MethSRTS, respectively. The difference between the offline and online modes of ClassSRTS is explained in Section 3.4. For Ekstazi, the online mode collects test dependencies during test runs on the new version, while the offline mode collects them in a separate phase (hence, the results of test runs can be obtained faster, but the overall machine time used is higher) [69]. For each technique, we show the minimum, maximum, and average time relative to RetestAll. The last row shows the average for each column.

The results show that ClassSRTS and Ekstazi both provide benefits over RetestAll (with the average end-to-end time across all projects being 62.5% to 68.2% of the RetestAll time). Comparing ClassSRTS and Ekstazi, we find them to be fairly similar in the respective modes. Based on the average time, ClassSRTS slightly outperforms Ekstazi (62.5% to 64.0% in the fastest, offline modes), but we do note that Ekstazi is implemented as a plugin for the Maven build system [68], while our implementation for ClassSRTS is not yet available as a Maven plugin. Therefore, Ekstazi analysis involves some overhead from Maven. (Both ClassSRTS and Ekstazi actually run tests through Maven.) In brief, we find ClassSRTS and Ekstazi to be equally good based on these experiments. In the future, we plan to perform a deeper comparison, especially to determine how much the analysis performed by ClassSRTS to select tests is faster than the dynamic analysis that Ekstazi uses, because Ekstazi tends to select fewer tests to run than ClassSRTS, as seen in Section 3.5.3. For both ClassSRTS

Table 3.3: Safety and precision violations of static RTS compared to Ekstazi

	Safety Violation %								Precision Violation %							
	ClassSRTS				MethSRTS				ClassSRTS				MethSRTS			
Project	revs	min	max	avg	revs	min	max	avg	revs	min	max	avg	revs	min	max	avg
p1	0.0	n/a	n/a	n/a	13.6	33.3	100.0	48.1	4.5	33.3	66.7	50.0	6.1	33.3	66.7	45.8
p2	0.0	n/a	n/a	n/a	76.7	11.1	100.0	40.6	9.3	6.7	75.0	42.3	67.4	15.4	80.0	57.5
p3	0.0	n/a	n/a	n/a	43.3	11.1	100.0	41.9	20.0	20.0	33.3	27.5	0.0	n/a	n/a	n/a
p4	0.0	n/a	n/a	n/a	4.0	16.7	16.7	16.7	16.0	5.9	90.9	61.7	42.0	4.3	80.0	36.4
p5	0.0	n/a	n/a	n/a	30.3	3.2	10.0	6.8	30.3	4.8	100.0	51.1	66.7	33.3	100.0	78.6
p6	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	22.2	14.3	16.7	16.3	29.6	50.0	58.3	57.8
p7	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	36.8	3.6	25.0	16.0	79.0	1.6	96.3	75.6
p8	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	16.9	42.9	100.0	72.9	13.6	81.6	97.4	92.8
p9	0.0	n/a	n/a	n/a	1.6	6.7	6.7	6.7	6.3	25.0	40.7	28.9	44.4	48.4	93.3	87.5
p10	0.0	n/a	n/a	n/a	8.3	50.0	50.0	50.0	41.7	28.3	64.8	53.9	50.0	71.9	100.0	78.9
p11	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	26.1	20.0	71.4	42.3
p12	0.0	n/a	n/a	n/a	6.1	0.8	100.0	25.9	39.4	0.8	98.0	50.0	63.6	7.0	99.3	85.3
p13	0.0	n/a	n/a	n/a	16.4	5.9	15.4	11.3	49.2	6.7	96.3	54.9	83.6	55.6	97.0	80.2
p14	0.0	n/a	n/a	n/a	5.8	2.8	46.1	20.6	39.1	14.0	98.0	62.4	72.4	4.4	97.3	68.3
p15	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	16.1	7.1	40.0	26.2	41.9	3.6	91.7	65.8
p16	2.0	100.0	100.0	100.0	0.0	n/a	n/a	n/a	77.5	5.6	100.0	13.1	93.9	9.5	100.0	20.9
p17	0.0	n/a	n/a	n/a	12.2	2.7	24.2	8.4	14.3	20.0	56.8	33.3	75.5	4.1	95.0	76.0
p18	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	80.0	50.0	50.0	50.0	100.0	87.0	95.7	93.9
p19	1.8	50.0	50.0	50.0	5.3	2.3	7.0	3.9	17.5	35.5	100.0	61.2	79.0	50.2	100.0	92.6
p20	0.0	n/a	n/a	n/a	5.0	55.8	55.8	55.8	75.0	63.0	100.0	91.0	25.0	20.9	94.1	70.6
p21	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	51.0	15.4	92.3	31.9	68.6	15.4	88.9	55.0
p22	0.0	n/a	n/a	n/a	5.3	0.7	13.5	6.5	63.2	7.9	100.0	50.1	96.5	7.9	100.0	65.7
Average	0.2	6.8	6.8	**6.8**	10.6	9.2	29.3	**15.6**	33.0	18.7	70.2	**42.9**	55.7	28.4	86.5	**64.9**

and Ekstazi, comparing the offline and online modes shows that they do not differ much, which means that the end-to-end time is, on average, dominated by the time taken to run the selected tests, rather than by the times to either analyze what tests to run in the case of ClassSRTS or to compute test dependencies in the case of Ekstazi. Finally, we observe that MethSRTS is substantially slower, not only much worse than the other RTS techniques but even worse than RetestAll. Overall, we find that static and dynamic RTS at the class-level have comparable performance, while MethSRTS is effectively useless.

We also point out that RTS techniques provide more benefits for projects with longer-running tests. Counting the number of projects in which some mode on average performs worse than RetestAll (i.e., the AVG is over 100%), we see that ClassSRTS performs worse in two projects, while Ekstazi performs worse in five projects. However, most of those projects have shorter-running tests, with all five of these projects having tests that run in less than 5 seconds. For such projects, one can simply use RetestAll and not attempt any RTS. In fact, we could have even removed from our evaluation such projects with short-running tests, but we preferred to keep them to highlight that RTS is not appropriate in all cases. An important point is that RTS techniques should be not only as fast as possible but also as safe as possible—any RTS technique can be simply made faster by not selecting to run some tests, but then it risks missing regressions.

3.5.5 RQ3: Safety and Precision

We compare the safety and precision of ClassSRTS and MethSRTS with respect to Ekstazi, because Ekstazi is a fairly safe and precise dynamic RTS technique [69]. Recall that a safe RTS technique selects to run *all* tests that could change their behavior due to the code changes between two versions and a precise RTS technique selects to run *only* the tests that could change their behavior.

Table 3.3 shows a summary of safety and precision violations, computed as follows. Let E be the set of tests selected by Ekstazi and T be the set of tests selected by another technique on some version. Safety, respectively precision, violations are computed[3] as $|E \setminus T|/|E \cup T|$, respectively $|T \setminus E|/|E \cup T|$, to measure how much a technique is less safe, respectively precise, than Ekstazi; lower percentages are better. For each of the four combinations (of two types of violations and two static RTS techniques), we tabulate four metrics: revs is the percentage of all versions in which the technique was unsafe/imprecise (i.e., the percentage was not 0), and MIN, MAX, and AVG are the minimum, maximum, and average, respectively, percentages of tests missed (for safety) or selected extra (for precision). Intuitively, AVG captures how bad the safety/precision violations are when they happen in a project. We use "n/a" when there were no safety/precision violations for the project. The last row shows the average for each column; we treat "n/a" as 0 when computing the overall averages.

Table 3.3 shows several interesting results. One surprising result for us is that ClassSRTS was rarely unsafe. Only 2 projects had safety violations, averaging 0.2% across all versions of all projects evaluated. (Some example safety violations are discussed in Section 3.6.) Moreover, we found that MethSRTS is both less safe and less precise than ClassSRTS, i.e., method-level static RTS is much less effective than class-level static RTS. On average, across all 22 projects, the percentages of versions in which ClassSRTS incurs safety violations and precision violations are 0.2% and 33.0%, respectively. For MethSRTS, these percentages increase to 10.6% and 55.7%, respectively (10.4 and 22.7 percentage points more unsafe and imprecise, respectively, than ClassSRTS). ClassSRTS is also more effective than MethSRTS in terms of number of projects with safety violations: 2 projects for ClassSRTS vs. 14 projects for MethSRTS. (The number of projects with a precision violation is the same, 21, for both techniques.) Comparing the MIN, MAX, and AVG values shows the same trend, i.e., when there is a safety or precision violation, MethSRTS is more unsafe or imprecise.

[3]We consider the union of tests selected by both Ekstazi and the technique to avoid division by zero in cases where Ekstazi does not select any test but a static RTS technique selects some tests.

Table 3.4: Impacts of different call-graph analyses on MethSRTS

	Tests Selected %				Safety Violation %				Precision Violation %				Time %			
Project	CHA	RTA	0CFA	01CFA	CHA	RTA	0CFA	01CFA	CHA	RTA	0CFA	01CFA	CHA	RTA	0CFA	01CFA
p1	74.8	74.0	70.7	70.7	48.1	48.1	48.1	48.1	45.0	44.4	45.8	45.8	777.8	493.7	390.5	387.2
p2	27.8	21.9	21.9	21.9	40.5	40.6	40.6	40.6	61.0	57.5	57.5	57.5	515.2	280.6	220.6	222.1
p3	39.7	32.7	29.4	29.4	75.0	47.5	41.9	41.9	19.7	30.0	n/a	n/a	571.2	571.5	227.7	237.6
p4	44.7	40.8	39.0	38.7	16.7	16.7	16.7	15.0	39.8	35.8	36.4	36.1	441.8	257.5	202.5	203.3
p5	68.9	68.8	59.8	59.8	n/a	n/a	6.8	6.8	75.0	75.1	78.6	78.6	484.9	346.4	231.7	247.6
p6	35.8	34.0	34.0	34.0	n/a	n/a	n/a	n/a	60.3	57.8	57.8	57.8	263.5	193.3	175.9	176.0
p7	52.4	50.6	58.1	58.1	n/a	n/a	n/a	n/a	72.2	71.2	75.6	75.6	661.0	659.8	9305.2	121213.1
p8	25.4	23.7	15.5	10.3	n/a	n/a	n/a	71.4	94.8	94.5	92.8	93.0	470.5	403.8	352.8	427.5
p9	21.6	21.6	15.8	15.8	n/a	n/a	6.7	5.0	90.0	90.0	87.5	87.5	435.9	231.5	166.9	167.1
p10	35.8	35.8	35.8	35.8	50.0	50.0	50.0	50.0	78.9	78.9	78.9	78.9	378.5	237.0	165.8	169.6
p11	15.3	15.3	15.1	14.6	n/a	n/a	n/a	11.1	42.9	42.9	42.3	42.3	143.2	100.6	86.5	86.3
p12	61.0	61.0	59.0	59.0	50.8	50.8	25.9	25.9	85.4	85.4	85.3	85.3	496.5	395.1	2670.4	416.1
p13	61.2	61.2	51.3	51.1	13.9	13.9	11.3	11.6	82.5	82.5	80.2	79.8	880.1	713.6	316.8	235.6
p14	40.9	40.9	40.0	40.0	20.6	20.6	20.6	20.6	68.7	68.7	68.3	68.3	109.5	88.1	81.7	82.4
p15	36.6	36.6	36.6	36.6	n/a	n/a	n/a	n/a	65.8	65.8	65.8	65.8	69.1	61.1	57.3	59.0
p16	98.1	100.0	96.5	93.0	94.1	n/a	n/a	12.5	24.4	24.3	20.9	23.4	121.8	125.6	80.0	79.4
p17	37.7	37.2	18.2	18.2	n/a	n/a	8.4	8.4	86.1	85.5	76.0	76.0	113.5	85.8	46.2	47.3
p18	95.8	95.8	95.8	95.8	n/a	n/a	n/a	n/a	93.9	93.9	93.9	93.9	134.1	131.0	107.4	106.3
p19	36.0	35.8	28.9	28.9	n/a	n/a	3.9	3.9	94.4	94.4	92.6	92.5	356.3	267.2	112.1	107.7
p20	31.4	31.4	17.5	17.5	n/a	n/a	55.8	58.1	89.2	89.2	70.6	70.6	158.2	194.1	112.5	128.6
p21	61.4	61.4	57.1	57.1	n/a	n/a	n/a	n/a	58.6	58.6	55.0	55.0	73.3	70.7	68.2	67.8
p22	71.7	71.4	67.8	67.8	7.1	6.5	6.5	6.5	66.0	65.8	65.7	65.7	118.1	112.0	117.4	107.1
Average	48.8	47.8	43.8	43.4	18.9	13.4	15.6	19.9	67.9	67.8	64.9	65.0	353.4	273.6	695.3	5680.7

3.5.6 RQ4: MethSRTS Variants

Impacts of call-graph analyses: Table 3.4 summarizes the results comparing different call-graph analyses for MethSRTS. For each analysis, columns 2–5 present the average percentage of tests selected for all versions in each project, while columns 6–9/10–13/14–17 present the safety/precision/overhead information. From the table, we observe three things. First, in general, more precise call-graph analyses tend to select fewer tests. For example, on average, CHA selects 48.8% of tests, while 0-1-CFA selects 43.4% of tests. This is because some changed nodes may be reachable from tests in imprecise call graphs but not reachable in precise call graphs. Second, there is no clear trend for safety issues because the precision of call-graph analyses does not directly impact the soundness of the constructed call graphs; instead, the safety issues are usually due to reflection and library exclusion. Third, although 0-1-CFA has a much higher time overhead, it has similar precision with 0-CFA while being the most unsafe of all the four variants. 0-1-CFA is very expensive because it spends a lot of time to approximate possible runtime types of the receiver object for each method invocation [200]. Comparing 0-CFA and 0-1-CFA, 0-CFA has much fewer safety issues and lower overhead; comparing CHA and RTA, RTA has much fewer safety issues and lower overhead. Therefore, 0-CFA (also suggested by WALA developers) and RTA seem to be the most suitable for MethSRTS.

Impacts of library exclusion: To study the impacts of analyzing library code, for each project, we ran WALA with and without library exclusion on only *one* randomly selected version pair. We do not run for all versions, because analysis without library exclusion is quite slow. We set a timeout of 3 hours for each version pair. The results are shown in

Table 3.5: Impacts of library exclusion on MethSRTS (only one version pair, not all pairs, per project)

	Tests Selected %				Safety Violation %				Precision Violation %				Time %			
	exclusion		no exclusion		exclusion		no exclusion		exclusion		no exclusion		exclusion		no exclusion	
Project	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA
p1	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	450.0	370.8	5550.0	81304.2
p2	0.0	0.0	100.0	100.0	n/a	n/a	n/a	n/a	n/a	n/a	100.0	100.0	478.1	237.5	5475.0	97087.5
p3	33.3	0.0	66.7	66.7	50.0	100.0	n/a	n/a	n/a	n/a	n/a	n/a	678.1	609.4	5271.9	73028.1
p4	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	4.3	4.3	4.3	4.3	627.0	316.2	6573.0	71027.0
p5	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	81.8	81.8	81.8	81.8	474.5	333.3	4452.9	53649.0
p6	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	50.0	50.0	50.0	50.0	402.1	261.7	3542.6	47459.6
p7	44.3	44.3	44.3	44.3	n/a	n/a	n/a	n/a	96.3	96.3	96.3	96.3	600.0	585.5	4960.0	65352.7
p8	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	97.4	97.4	97.4	97.4	566.1	496.6	3819.1	54051.5
p9	97.8	97.8	97.8	97.8	n/a	n/a	n/a	n/a	88.9	88.9	88.9	88.9	501.9	226.0	4018.3	34651.0
p10	1.2	1.2	1.2	1.2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	437.1	243.8	3788.6	34364.8
p11	41.2	41.2	41.2	41.2	n/a	n/a	n/a	n/a	42.9	42.9	42.9	42.9	204.6	122.9	1681.7	20593.1
p12	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	64.8	64.8	64.8	64.8	678.4	553.8	3670.8	32296.5
p13	98.5	98.5	98.5	DNF	n/a	n/a	n/a	DNF	48.1	48.1	48.1	DNF	958.1	752.1	6385.5	DNF
p14	13.5	13.5	13.5	13.5	n/a	n/a	n/a	n/a	71.4	71.4	71.4	71.4	80.8	61.5	595.3	7551.1
p15	96.3	96.3	96.3	96.3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	157.4	136.1	507.0	5935.8
p17	97.7	97.7	97.7	97.7	n/a	n/a	n/a	n/a	67.4	67.4	67.4	67.4	181.8	151.7	758.8	6142.2
p19	100.0	100.0	100.0	DNF	n/a	n/a	n/a	DNF	94.2	94.2	94.2	DNF	527.3	422.4	2527.9	DNF
p21	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	44.4	44.4	44.4	44.4	108.1	104.0	181.7	1106.3
p22	58.0	58.0	58.0	58.0	n/a	n/a	n/a	n/a	50.0	50.0	50.0	50.0	140.8	130.3	586.5	4919.1
Average	72.7	71.0	79.8	77.5	2.6	5.3	n/a	n/a	47.5	47.5	52.7	50.6	434.3	321.9	3386.7	36343.1

Table 3.5, where "DNF" means that the run timed out. (Note that the corresponding times in tables 3.5 and 3.4 do not match because Table 3.5 is for only one version pair for each project.) 0-CFA and 0-1-CFA timed out for *all* projects without library exclusion, so we do not show them. Additionally, the analyses failed due to memory constraints for three projects (p16, p18, and p20); we do not show these rows. From the results in Table 3.5, we observe two things. First, without library exclusion, the more expensive and precise RTA analysis does not pay off—RTA selects the same number of tests as the less expensive CHA in all cases, for projects where neither analyses timed out. Second, the analysis overhead relative to RetestAll is much higher without library exclusion than with library exclusion. For `commons-math` (p19), CHA overhead is 2527.9% without library exclusion but 527.3% with library exclusion because without library exclusion, 33,770 more classes need to be analyzed for this project. The analysis time for RTA is even higher without library exclusion. In p19, the RTA times out. RTA has such a high overhead because it spends significant time to compute the sets for approximating potential runtime types of the receiver object for every method invocation [200]. Overall, these results demonstrate that call-graph analyses without library exclusion are not practical for static RTS due to the high cost and precision issues. In fact, method-level static RTS unfortunately appears impractical in all configurations.

3.6 QUALITATIVE ANALYSIS

We discuss safety and precision violations; cases where static RTS does not select some test(s) that Ekstazi selects or selects some test(s) that Ekstazi does not select. An unsafe RTS

technique is bad in a non-obvious way—it can be deceivingly fast but risk missing regressions. Our analysis of safety violations identifies some cases where static RTS techniques were unsafe in our experiments. In contrast, an imprecise RTS technique is bad in an obvious way; the imprecision is reflected in the end-to-end time. These cases show current limitations of static analysis for RTS and can provide insight on improving static RTS in the future.

3.6.1 Safety Violations of Static RTS

Safety violations due to reflection.: In `commons-math`, between versions `2773215` and `c246b37`, ClassSRTS and MethSRTS miss to select nine tests that Ekstazi selects. The relevant change is to the abstract class `AbstractFieldIntegrator`, extended by several `*FieldIntegrator` classes (e.g., `ClassicalRungeKuttaFieldIntegrator`) that are definitely affected by the change. All techniques do select multiple tests, such as `ClassicalRungeKuttaFieldIntegratorTest`. The nine additional tests that only Ekstazi selects are in `*FieldStepInterpolatorTest` classes (e.g., `ClassicalRungeKuttaFieldStepInterpolatorTest`) that extend `AbstractRungeKuttaFieldStepInterpolatorTest`. As shown in Figure 3.4, `ClassicalRungeKuttaFieldStepInterpolatorTest` invokes the method `doInterpolationAtBounds` in `AbstractRungeKuttaFieldStepInterpolatorTest` that invokes `setUpInterpolator`, which in turn invokes `createButcherArrayProvider` in the same class. The latter method (red dashed box) gets `*FieldStepInterpolator` instances, replaces `"StepInterpolator"` with `"Integrator"`, and uses reflection to create the new `*FieldIntegrator` instance. ClassSRTS and MethSRTS are unsafe because they do not detect the potential use edge (red dashed arrow) due to reflection, and do not select related tests. In contrast, Ekstazi tracks the precise class dependencies, even in the presence of reflection, and detects that the `*FieldStepInterpolatorTest` instances depend on the corresponding `*FieldIntegrator` instances.

Safety violation due to library exclusion: In `compile-testing`, between versions `8d5229e` and `40c141b`, both Ekstazi and ClassSRTS select the same two tests, but MethSRTS with library exclusion is unsafe and selects no test. The relevant code change is to the class `Compilation`. As shown in Figure 3.5, test `JavaSourcesSubjectFactoryTest` can reach method `compile` in the changed class `Compilation` (marked in gray). The underlined statement in `JavaSourcesSubjectFactoryTest` invokes methods on `ASSERT`, a class in the third-party library `truth`. When third-party libraries are excluded from call-graph analysis, MethSRTS does not analyze the underlined statement and fails to select the test.

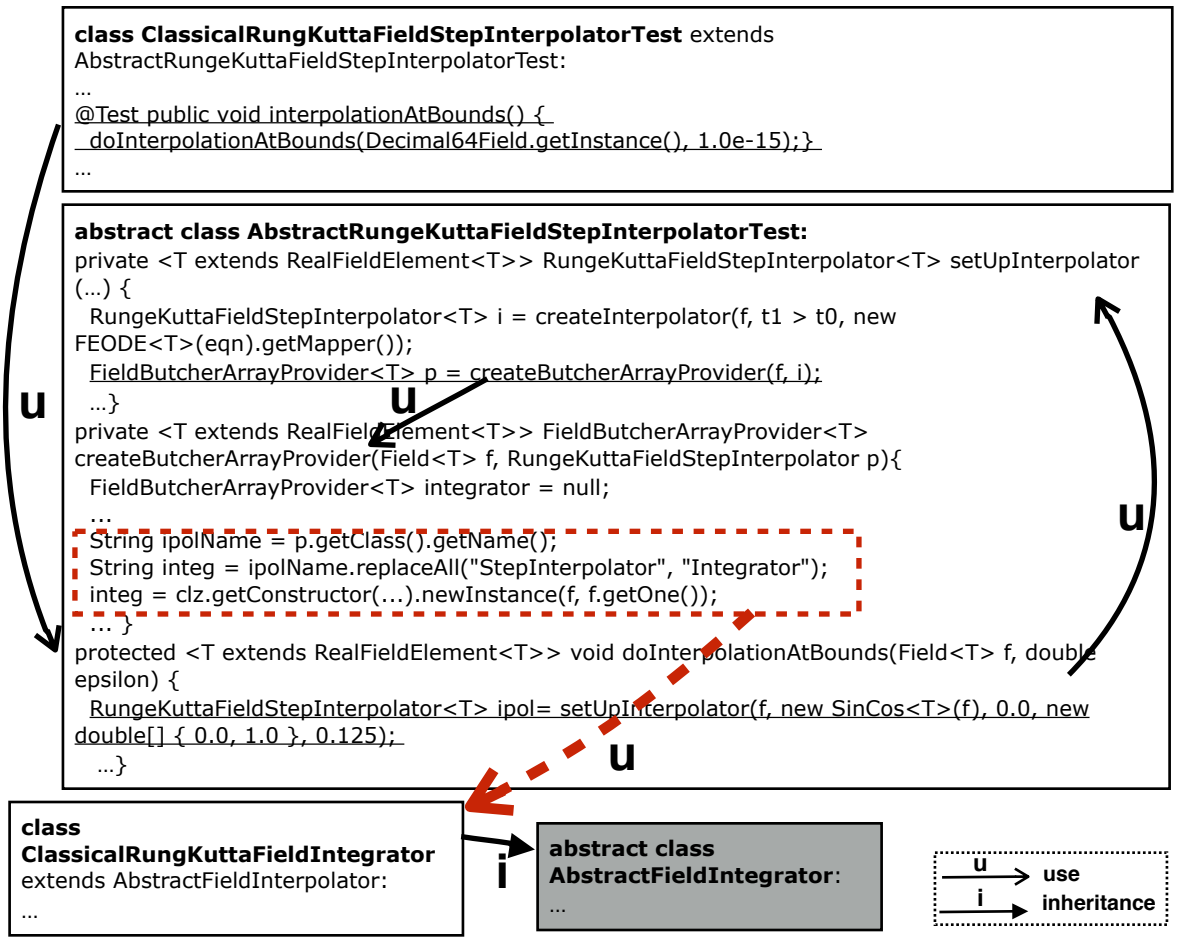


Figure 3.4: ClassSRTS safety violation due to reflection

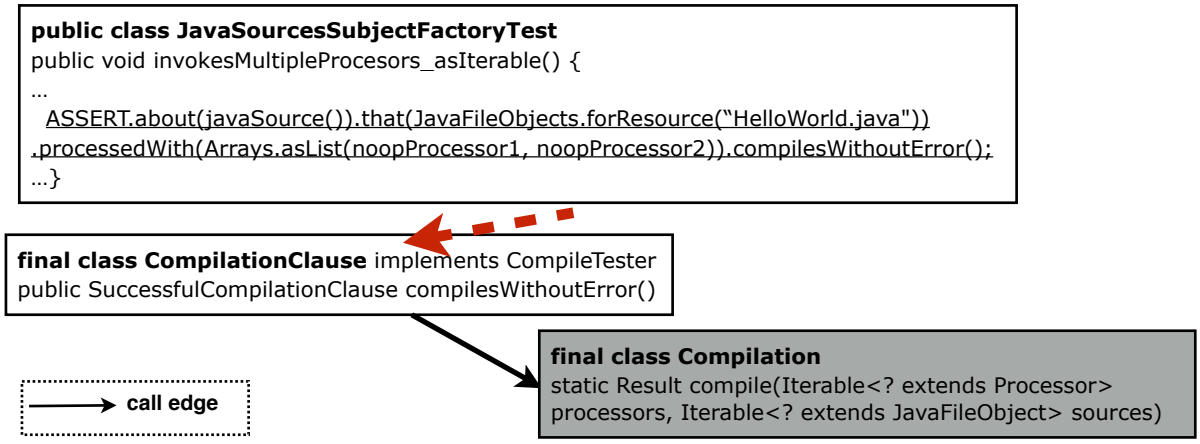


Figure 3.5: Safety violation due to third-party library exclusion (MethSRTS)

3.6.2 Precision Violations of Static RTS

Imprecision due to class-level analysis: In `asterisk-java`, between versions `166f2 93` and `d6bfce1`, Ekstazi selects four tests, while ClassSRTS selects four additional tests

due to the imprecision of analyzing at the class level. The setup method of `AsteriskAgentImplTest` calls `new AsteriskServerImpl()`. The method `onManagerEvent` in the class `AsteriskServerImpl` contains checks for the type of `event`, e.g., `if (event instanceof AgentCalledEvent)`. Therefore, ClassSRTS finds a transitive static dependency of `AsteriskAgentImplTest` on `AgentCalledEvent`, which changed between the mentioned versions. However, the actual run of `AsteriskAgentImplTest` never invokes `onManagerEvent` in `AsteriskServerImpl` (the relevant conditional is never executed), and Ekstazi correctly does not track this dependency.

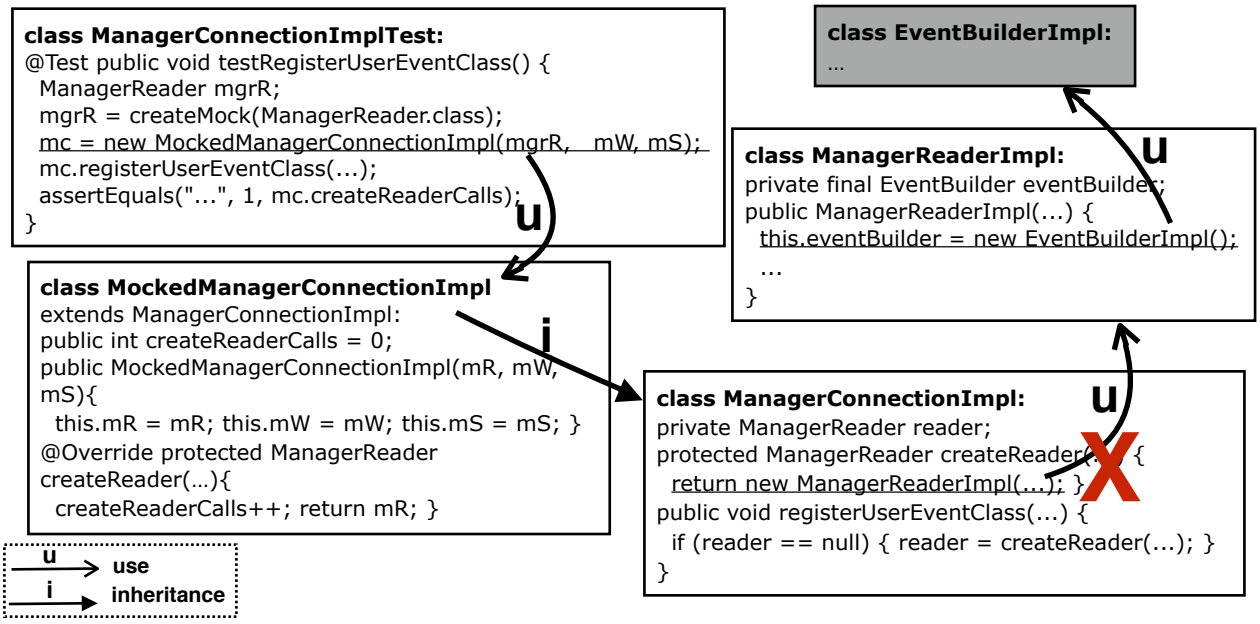


Figure 3.6: ClassSRTS precision violation due to dynamic dispatch

Imprecision due to dynamic dispatch: For the same `asterisk-java` versions, Figure 3.6 shows simplified code for a case where ClassSRTS finds a false transitive dependency of the test `ManagerConnectionImplTest` on the changed class `EventBuilderImpl` (marked in gray) due to dynamic dispatch. `ManagerConnectionImplTest` uses `MockedManagerConnectionImpl` that extends `ManagerConnectionImpl` and overrides `createReader`. `ManagerConnectionImpl` uses class `ManagerReaderImpl` which in turn uses the changed class `EventBuilderImpl`. Therefore, ClassSRTS finds that `ManagerConnectionImplTest` depends on the changed class. However, during the test run, when `ManagerConnectionImplTest` invokes `ManagerConnectionImpl.registerUserEventClass` to check how many times the `createReader` is called, the overriding method `createReader` in `MockedManagerConnectionImpl` is executed instead of `createReader` in the parent class. The use edge from `ManagerConnectionImpl.createReader` to `ManagerReaderImpl` is never executed and should not be considered (the red cross), i.e., the edge exists statically but not dynamically as `ManagerC`

onnectionImpl.createReader is never executed. In brief, ClassSRTS has this imprecision due to dynamic dispatch.

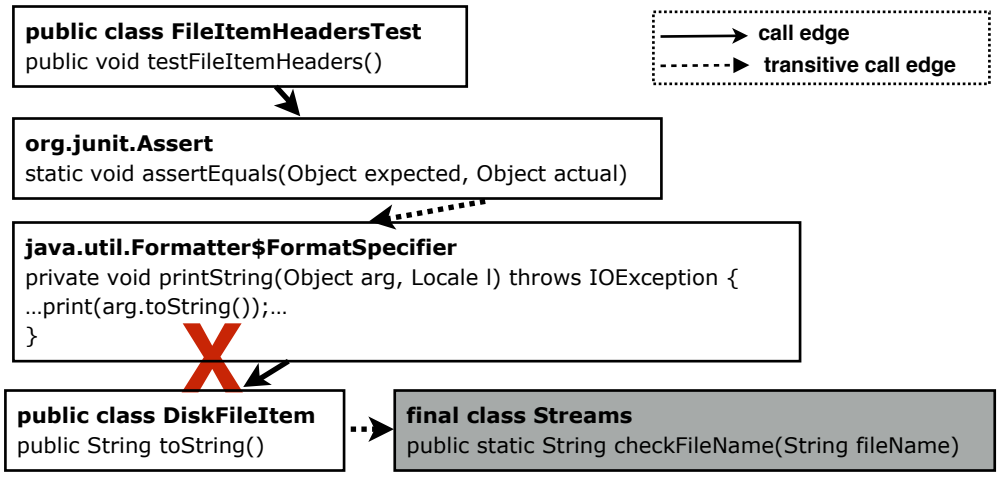


Figure 3.7: MethSRTS precision violation due to dynamic dispatch

Compared with ClassSRTS, MethSRTS is even less precise in identifying potential targets for dynamic dispatch. For example, between versions `1460430` and `1475836` of `commons-fileupload`, while Ekstazi and ClassSRTS select 6 and 7 tests, respectively, MethSRTS selects all 12 tests. Figure 3.7 shows a simplified call graph for `FileItemHeadersTest` which invokes JUnit's `assertEquals`, which, in turn, transitively invokes `printString` in library class `FormatSpecifier`. To resolve `toString` in `FormatSpecifier`, even the most advanced 0-1-CFA cannot precisely determine the runtime type of the `Object` receiver. So, all classes overriding `Object.toString` are potential targets including `DiskFileItem.toString` which transitively invokes a method in the changed class `Streams` (in gray). This example, as the one in Section 3.3.3, shows ClassSRTS is more precise because any possible runtime object type has to be referenced by a test to instantiate it.

3.7 THREATS TO VALIDITY

Internal: Our static RTS prototypes and scripts for running experiments may contain bugs. To mitigate risks, we use well-known libraries, e.g., ASM and WALA. We also wrote unit tests to check basic functionality, and we implemented some sanity checks for numbers generated from scripts, e.g., we check that the offline and online variants of both ClassSRTS and Ekstazi select the same tests.

External: The projects in our study may not be representative, so our results may not generalize. To address this, we used 985 versions from 22 open-source projects varying in size, application domain, number of tests, and test-suite running time. However, all our

projects are single-module Maven projects. The results could differ for bigger, multi-module Maven projects, but we expect that these results could show RTS to be even better as we generally find RTS to be more effective for projects with longer-running tests.

Construct: We chose Ekstazi as the ground truth against which to evaluate static RTS techniques. Ekstazi is a state-of-the-art, publicly available tool for dynamic RTS, but may still not represent the ground truth for all RTS.

3.8 SUMMARY

This chapter presented our implementation of several variants of two change-impact analyses and their evaluation in the context of RTS. ClassSRTS, based on class-level change-impact analysis, outperforms MethSRTS, based on method-level change-impact analysis, and ClassSRTS performs similarly as dynamic RTS. The next chapter describes STARTS, our tool for ClassSRTS, whose efficient change-impact analysis is a central component of evolution-aware RV techniques (Chapter 5). We also use STARTS for comparing and combining evolution-aware RV techniques with RTS.

CHAPTER 4: STARTS: CHANGE-IMPACT ANALYSIS AND RTS TOOL

We present STARTS, a tool for static change-impact analysis and *STA*tic *R*egression *T*est *S*election. In Chapter 3, we used a prototype tool to evaluate static change-impact analysis at different granularity levels in the context of RTS. The results show that (1) static RTS which uses a class-level change-impact analysis significantly outperforms static RTS which uses a method-level change-impact analysis, and (2) static RTS which uses a class-level static change-impact analysis performs comparably with the state-of-the-art dynamic RTS tool, Ekstazi [68, 69]. The results are encouraging, showing that static class-level change-impact analysis can be a core component of evolution-aware RV techniques (as we show Chapter 5). The results also show that static RTS is practical and worthy of further research.

The STARTS tool that we present in this chapter is a robust, publicly-available tool for performing static change-impact analysis and static RTS. STARTS constructs a class-level dependency graph relating all types (including classes, interfaces, and enums) in an application and computes a transitive closure for each type to find what are its dependencies. In STARTS, static RTS is a special case of change-impact analysis where the only types whose transitive closure are computed are the test classes, and the result of the transitive closure computation are the dependencies of each test class. STARTS determines the types that changed by computing the checksum of each type's corresponding compiled classfile (`.class` file) and comparing the computed checksum with the one that was computed in the prior version. In the change-impact analysis mode, STARTS returns all types with a changed dependency as impacted by the changes. In the RTS mode, STARTS selects to run impacted tests, which are tests whose transitive dependencies include a changed type.

Summary of Changes to the Initial Prototype: We made several improvements to our initial prototype [117] to make STARTS more robust and usable on real, large software projects. We added support for multi-module Maven projects and made several optimizations to make STARTS faster, including parsing only constant pools instead of entire class-files, saving dependencies as a *type-to-dependencies* map instead of a *dependency-to-types* map, using graph library with faster transitive closure computation (yasgl [213], instead of JGraphT [97]), and incrementally caching dependencies.

Evaluation and Publicly-Available Artifacts: We evaluated STARTS on 32 Maven-based projects from GitHub. We find that STARTS selects on average 35.2% of all tests, leading to an end-to-end runtime (consisting of the time to select what tests to run plus time to run those tests) that is 81.0% of RetestAll time to run all tests. STARTS scales well, and for 11 projects with longer-running tests that take over one minute to run, STARTS

selects on average 40.5% of all tests, leading to an end-to-end runtime that is only 68.2% of RetestAll time. STARTS source code is publicly available on GitHub at `https://github.com/TestingResearchIllinois/starts` and binary code is released on Maven Central. A video demo of STARTS can be found at `https://youtu.be/PCNtk8jphrM`.

4.1 USAGE

In this section, we describe how developers can integrate and use STARTS in their projects, as well as how to download and install from source. STARTS is a Maven plugin [137] and can be easily integrated with any Maven-based Java project.

Integrating STARTS. The easiest way to integrate STARTS with a project is to add the latest version of the STARTS plugin from Maven Central to the project's `pom.xml` file:

```
<plugin>
  <groupId>edu.illinois</groupId>
  <artifactId>starts-maven-plugin</artifactId>
  <version>${latest_STARTS_version}</version>
</plugin>
```

Detailed instructions, with example, for integrating STARTS into a Maven project can be found in the `README.md` file at the root of the STARTS repository [191].

Installing STARTS from source. The STARTS source code GitHub [191] can be installed by running the following command from the cloned STARTS directory:

```
$ mvn install
```

Using STARTS. Developers can use STARTS to perform several tasks: (1) finding types that changed semantically at the bytecode level (2) performing change-impact analysis (i.e., finding types that are impacted by the changes), (3) finding tests that are impacted by the changes without running those tests, and (4) finding and running tests that are impacted by the changes. To achieve these tasks, developers can invoke several STARTS Maven goals:

```
$ mvn starts:help # list all goals
$ mvn starts:diff # find types that changed semantically
$ mvn starts:impacted # find types impacted by changes (change-impact analysis mode)
$ mvn starts:select # find (but not run) impacted tests
$ mvn starts:starts # find and run impacted tests (RTS mode)
$ mvn starts:clean # delete STARTS artifacts
```

The first goal, `starts:help`, lists all the Maven goals in STARTS and what they can be used for. The other five goals are related to change-impact analysis and RTS. `starts:diff` displays all the `Java` types (including classes, interfaces, and enums) that changed since the last time STARTS was run. `starts:impacted` runs STARTS in change-impact analysis mode; it displays all types (not just *test* classes) that are impacted by the changes. `starts:select` displays, but does not run, the test classes that are impacted by the changes since the last time STARTS was run, where the display can be to screen or to file—allowing developers more flexibility in their test selection process, to first select impacted tests and then run those tests later. `starts:starts` runs the impacted tests; it performs the functions of the previous goals (except `starts:help`), plus execution of the impacted tests. Finally, `starts:clean` removes all artifacts that STARTS stored from a previous run (in a `.starts` directory), resetting STARTS so that in the next run, all types are considered changed (and all tests are selected to be run, if using `starts:starts`).

STARTS provides several options that give some flexibility to the user. The most important option is whether or not to update the STARTS artifacts after invoking a goal. As described in Section 4.2.2, STARTS keeps track of the checksums of all types from the previous run, storing them to disk and using them in the new run to find impacted types/tests. All goals for change-impact analysis and RTS in STARTS provide an `update*Checksums` option, which, when `true`, updates the stored checksums after a run. This option is set to `true` by default for the `starts:starts` goal, but is `false` by default for all other goals.

4.2 TECHNIQUE AND IMPLEMENTATION

We describe the technique implemented in STARTS and the STARTS Maven plugin.

4.2.1 Technique

STARTS performs static change-impact analysis and static RTS at the class level—impacted types (respectively, tests) are selected at the class (not method) level and the dependencies of these types (respectively, tests) are also computed at the class/type level. Our recent work [117], discussed in Chapter 3, showed that static RTS based on class-level change-impact analysis outperformed static RTS based on method-level change-impact analysis and was comparable with the state-of-the-art class-level dynamic RTS technique Ekstazi [69]. Thus, we implemented STARTS to perform class-level change-impact analysis and static RTS, based on the idea of a *class firewall* [107, 125, 149], which encloses impacted types that need to be retested and tests that need to be rerun because they may behave

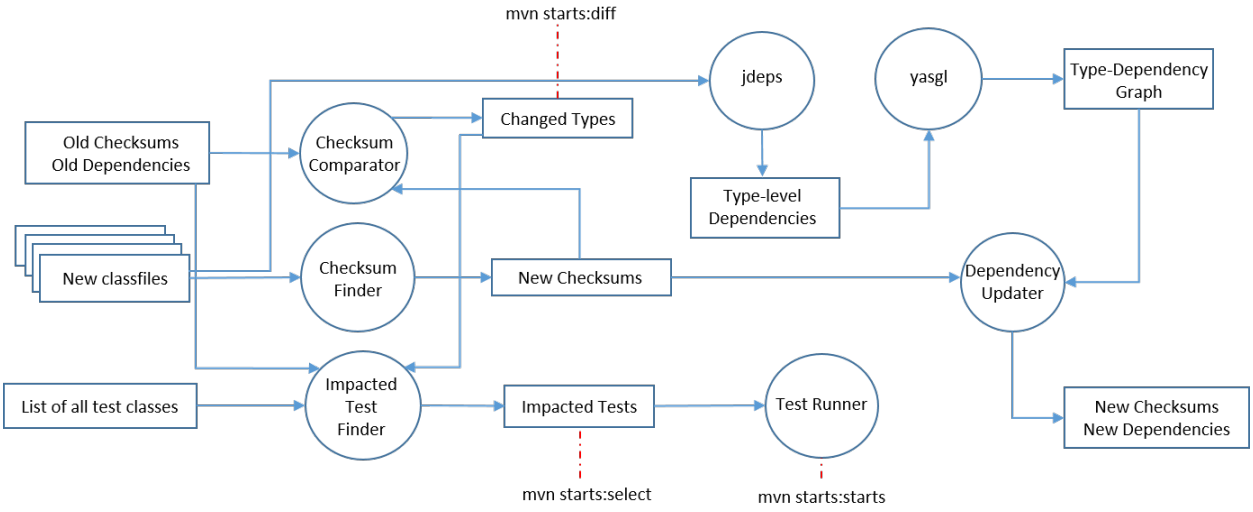


Figure 4.1: STARTS Architecture

differently after code changes. The class firewall is computed on an *intertype relationship graph (IRG)* where each node is a type (i.e., class, interface, enum, etc) in the application and there is a directed edge from one type τ to another type τ' if τ has a direct use or inheritance dependency on τ'. Test class nodes are also included in the IRG. If τ_c is a type that changed, then τ is impacted by the change to τ_c iff $\langle \tau, \tau_c \rangle \in E^*$, where E is the set of all edges in the IRG, and * denotes the reflexive and transitive closure. The class firewall is the set of all types that can transitively reach any of the types that changed in the IRG, and can therefore be defined as $firewall(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^*$, where $\mathcal{T}_c$ is the set of all types that changed, $^{-1}$ denotes the inverse relation, and $\circ$ denotes relation composition. Given (1) classfiles for all types (obtained from compiling a new version) in an application and (2) checksums of classfiles from a prior version, STARTS can output the set of changed types ($\mathcal{T}_c$), the class firewall ($firewall(\mathcal{T}_c)$), and T_i, the set of impacted tests. T_i is computed as the set difference between the set of all tests in the new version and the set of tests that are not in $firewall(\mathcal{T}_c)$ (which is computed from the old version). We compute T_i this way to include any newly-added tests while still using the IRG computed on the old version.

4.2.2 Implementation

Figure 4.1 shows the STARTS architecture, containing components to: (1) find dependencies among types in the application, (2) construct the IRG, (3) find changed types between two versions of the application, (4) store checksums of all types from the current version, (5) select the tests impacted by the changed types, and (6) run impacted tests.

Finding Dependencies Among Types. STARTS needs to compute the dependencies

among all types in the application. The prototype in our prior work [117], discussed in Section 3.4, used ASM to parse *all* the bytecode in the compiled classfile of a given type in order to compute its dependencies. However, parsing entire classfiles just to find dependencies is rather slow because it requires to recursively visit each type's fields, methods, signatures, and annotations to collect all the types that are referenced. STARTS improves on computing dependencies among types by only reading the constant pool in each classfile to determine all types that the type in the classfile may depend on. We use the recent Oracle *jdeps* tool [95], now part of the standard Java library, to read the constant pools. After the new version of an application has been compiled to produce classfiles, STARTS makes a single jdeps invocation (via the jdeps API) to parse all classfiles in the application at once, and then processes the jdeps output in memory to find the dependencies for each type.

Constructing the Dependency Graph. The IRG contains an edge from one type to each of its dependencies. We use a custom graph library called yasgl [213] to construct graphs and to find tests that can transitively reach some changed type. We add each type as a node in a yasgl graph and add dependencies computed by jdeps as edges between nodes in the graph. With a yasgl graph, STARTS computes the transitive closure of each test class to find all types that each test depends on. Our initial prototype [117] used JGraphT [97], but yasgl is faster for computing the transitive closure. For example, yasgl takes 1.4 sec to compute the transitive closure for a graph with 41,960 nodes and 509,946 edges (coming from a single module of a project with 110 test classes). JGraphT takes 2.7 sec to compute the same transitive closure, a difference that accumulates when considering all the modules in the project. Note that the yasgl IRG that STARTS uses does not distinguish between use edges and inheritance edges, as done in our initial prototype and in prior work [149].

Finding Changed Types. STARTS finds the types that changed since the last time it was run. STARTS uses the same checksum function from Ekstazi [68, 69] to compute a checksum that ignores debug-related information for each classfile and stores that checksum to a file. STARTS tracks changes in classfiles because the corresponding source file can be different yet result in the same classfile that is actually executed, so tracking classfiles is more precise. Also, STARTS uses checksums for checking whether a classfile is modified instead of seemingly faster methods like timestamps, which can be unreliable (e.g., Maven's incremental build system is broken [135] and often recompiles every type on each run, so one cannot rely on the timestamps of the classfiles). Once compilation is complete in the new version, STARTS computes the checksums of all compiled classfiles and compares against the stored checksums computed from the previous version for each file. If the old and new checksums differ, STARTS considers that type to have changed. If the type had no previously computed checksum (i.e., a new type was added), its checksum is stored for

future runs. Finally, if a type for which STARTS previously computed a checksum cannot be found in the new version (i.e., an old type was deleted), then that type is no longer stored in the checksum file for future runs. If there is no checksum file on disk (e.g., on the very first run, or after running `mvn starts:clean`), STARTS considers all types as changed.

Computing and Storing Checksums. In our initial prototype [117], as well as in Ekstazi, the transitive closure of each test class in the graph was stored as a mapping from each test class to its dependencies, i.e., a *test-to-types* mapping. Further, there was one dependency file per test. Once a tool computed the set of types that changed, it then checked the dependency file of each test to see if the test depends on any of the changed types. However, we observed that STARTS discovers many more test dependencies than Ekstazi, due to inherent imprecision of static analysis, and that many tests shared a lot of these dependencies. As a result, we reversed the dependency storage format in STARTS to reduce the amount of repetitive checking of test dependencies by storing a *type-to-tests* mapping. STARTS stores in a single file a mapping from each type in the application to the set of tests that depend on that type. This file is stored in a directory called `.starts` under the root directory of the application. More precisely, if the application is a multi-module Maven-based project, STARTS creates multiple `.starts` directories, each with its own type-to-tests-mapping file, under each module, and the types may span across modules if that is where the dependencies lead. Updating the checksums that are stored on disk after invoking a STARTS goal on a new version can be turned on or off, as described in Section 4.1.

The type-to-tests storage format that STARTS uses, together with processing only one file on disk, greatly improves the performance of selecting impacted tests. For example, in one project, STARTS takes 22.9 sec to check if any of the dependencies changed when using the type-to-tests, single-file format, but the same check takes 79.8 sec with Ekstazi's test-to-types, multiple-files format. One possible modification of the test-to-types format could be to first read all the files and then reverse the mapping (in memory) to be from type to tests before comparing checksums. However, this modification would still incur the cost of reading potentially many files from disk and it would put the mapping-reversal process on the critical path from when testing is initiated until developers obtain test results—mapping reversal in STARTS can happen in a separate offline phase that is not on the critical path.

Selecting Impacted Tests. STARTS uses the type-to-tests dependency mapping from the previous version and the set of all changed types to find the tests that are **not** impacted by changes. STARTS then computes the impacted tests as the difference between the set of all tests in the current version and the set of non-impacted tests. Thus, newly-added tests are always in the set of impacted tests. Dependency graph construction on the new version is *not* required to find impacted tests (allowing quicker computation of impacted tests).

Rather, STARTS reads the type-to-tests dependency file which was computed based on the dependency graph constructed in the previous version. The fact that STARTS requires only compile-time information to find impacted tests can allow a clean separation of phases: an *analysis phase (*a*)* finds changes and impacted tests, an *execution phase (*e*)* runs the impacted tests, and a *graph computation phase (*g*)* builds the dependency graph and uses it to create a type-to-tests mapping for the next version[1]. This separation can enable the choice to run STARTS in an "online mode" (the a, e, and g phases are run together) or an "offline mode" (the a and e phases can run separately from or in parallel with the g phase). We did not yet implement goals to toggle the online/offline modes, but report times for offline mode as the time for online mode minus the time for the g phase. `starts:select` displays the impacted tests but does not run them.

Running Impacted Tests. STARTS computes the set of selected tests to run as previously described: it excludes non-impacted tests from the set of all tests in the application. Specifically, STARTS dynamically adds the non-impacted tests to the set of tests that Surefire plugin is already configured to not run. As a result, when STARTS invokes the Maven Surefire plugin to run the tests, Surefire will run only the tests that are impacted by the changes. The goal `starts:starts` will perform all the previous steps to find changed types, select impacted tests, and run those selected tests.

4.2.3 Important STARTS Options

STARTS provides a number of other options, in addition to turning on/off the checksum file updates (Section 4.1).

Caching jdeps output. One consideration in the design of STARTS is how to handle the output of running jdeps on third-party libraries (JARs). Many projects do not frequently change their library versions and using jdeps to parse the library code on each version would needlessly repeat work. STARTS therefore provides options to (1) use a preprocessed cache, (2) incrementally build the cache on each version, and (3) parse the third-party libraries on each version. The default is to incrementally build the cache on each version. When STARTS encounters a JAR in the application's classpath, it first checks whether a corresponding jdeps output file exists in the `jdeps-cache` directory, which is found in each module of the application. If there is one, STARTS reads it; otherwise, STARTS runs jdeps on the JAR, uses the jdeps output for its current processing, and stores the jdeps output for that JAR in a file in the `jdeps-cache` directory. The names of the files in the `jdeps-cache` directory

[1]The g phase in STARTS is analogous to the coverage-collection (c) phase in Ekstazi and other dynamic RTS techniques where separation of c and e phases is harder to achieve in practice.

match the Group/ArtifactId/Version convention of naming Maven-based projects [136], and have a `.graph` extension.

If there is a cache (possibly computed elsewhere or even from different applications), STARTS can be configured to specify the location of this cache. The following command shows using an RTS-related goal with a preprocessed cache, where `${GRAPH_CACHE}` is the directory containing the preprocessed jdeps output for each third-party library and the jdeps output files are organized as described for the default option:

```
$ mvn starts:starts -DgCache=${GRAPH_CACHE}
```

If no cache is input or the cache is empty, STARTS runs jdeps on all libraries per version.

File Formats for Checksums and Dependencies. STARTS supports two formats for storing the checksums of all types in the application and the tests that transitively depend on them: the new type-to-tests (`ZLC`) format and the old test-to-types (`CLZ`) format (proposed in Ekstazi). Section 4.2.2 describes these formats and their tradeoffs. `ZLC` is the default file format that STARTS uses. To run STARTS using the `CLZ` file format:

```
$ mvn starts:starts -DdepFormat=CLZ
```

Controlling STARTS Artifact Storage. Configuring different logging levels can control the amount of information that STARTS stores between runs, where the logging levels are the same as in the `java.util.logging` API. When running at the default logging `Level.INFO`, STARTS stores only the checksum and dependency file, `.starts/deps.zlc`, between runs. At `Level.FINEST`, STARTS will store all its files: the lists of all/impacted/non-impacted tests, the dependencies that jdeps computed, the classpath that STARTS used, the yasgl graph that STARTS constructed internally, and the set of changed types. Running at logging level `Level.FINER` will store only `.starts/deps.zlc`, the set of impacted tests, and the set of all tests. To run STARTS while storing all its files:

```
$ mvn starts:starts -DstartsLogging=FINEST
```

4.3 EVALUATION

We ran all experiments on a 3.40 GHz Intel Xeon E3-1240 V2 machine with 16GB of RAM, running Ubuntu Linux 16.04.3 LTS and Oracle Java 64-Bit Server version 1.8.0_144. We evaluated STARTS on 32 Maven projects. These projects include 21 single-module Maven projects we used in our previous study [117] and 11 multi-module Maven projects that we did not evaluate before, showing that STARTS can be integrated into larger Maven projects. We ran STARTS on each project over a number of versions and measured the number of

Table 4.1: Statistics about selected tests and end-to-end time of STARTS compared to RetestAll

Project	SHAs [#]	ALL	Selected [#]	Selected [%]	RTA[s]	Offline Time [%]	Online Time [%]	Breakdown			
								a	e	g	Comp.
headius/invokebinder	68	2.1	1.6	76.0	3.3	110.7	134.8	0.0	20.0	20.0	60.0
google/compile-testing	32	7.3	3.1	44.1	4.4	118.3	137.6	0.0	18.3	13.3	68.3
apache/commons-cli	52	23.0	10.2	44.1	4.8	109.4	126.1	0.0	10.3	10.3	79.3
logstash/logstash-logback-encoder	45	18.2	3.7	23.5	5.6	112.8	129.8	0.0	13.7	12.3	74.0
apache/commons-dbutils	15	24.6	8.2	33.0	5.6	109.1	121.6	0.0	13.0	13.0	73.9
apache/commons-validator	22	61.0	13.8	22.6	6.6	93.5	107.2	0.0	17.1	11.4	71.4
apache/commons-fileupload	8	12.0	3.8	31.2	6.8	98.2	102.1	0.0	12.9	5.7	81.4
apache/commons-codec	65	47.4	2.1	4.5	9.3	69.5	73.9	0.0	10.1	7.2	82.6
srt/asterisk-java	47	38.1	2.3	6.0	9.6	70.2	79.4	0.0	18.4	10.5	71.1
apache/commons-functor	20	164.0	23.2	14.1	10.7	91.7	96.3	0.0	8.7	5.8	85.6
apache/commons-compress	12	89.4	23.8	26.6	13.3	79.8	83.5	0.0	25.9	4.5	69.6
apache/commons-email	10	18.0	5.4	30.0	16.2	68.0	71.9	0.0	39.7	6.0	54.3
square/retrofit	13	32.2	10.3	32.2	21.1	79.7	86.8	5.9	43.0	9.7	41.4
apache/commons-lang	63	133.7	42.8	32.0	24.8	73.3	76.8	0.0	35.3	4.7	60.0
apache/commons-collections	12	164.0	7.2	4.4	25.3	58.3	58.9	0.0	10.0	1.3	88.7
AdoptOpenJDK/jitwatch	23	26.0	10.6	40.6	26.4	58.4	60.9	0.0	62.5	4.4	33.1
graphhopper/graphhopper	8	106.8	70.1	65.7	29.8	90.8	97.3	0.0	45.2	6.9	47.9
apache/commons-imaging	89	58.8	21.5	37.9	29.5	65.2	67.5	0.0	51.5	3.5	45.0
cloudera/oryx	17	58.0	17.3	29.8	37.6	85.0	91.3	6.0	40.1	6.6	47.3
robovm/robovm	11	32.0	9.1	28.4	39.5	107.5	111.6	1.4	5.7	3.6	89.3
ninjaframework/ninja	6	102.0	55.0	53.9	40.5	93.6	120.3	7.2	42.0	22.5	28.3
Average(SHORT)	30.4	58.0	**16.4**	**32.4**	17.6	**87.8**	**96.9**	1.0	25.9	8.7	64.4
apache/commons-math	63	449.9	42.4	9.4	98.3	28.9	30.3	0.3	36.8	4.7	58.2
addthis/stream-lib	7	24.0	5.4	22.6	106.4	47.5	48.4	0.0	88.8	2.1	9.1
apache/commons-io	13	99.2	23.4	23.5	132.0	43.4	43.9	0.0	85.0	1.2	13.8
brettwooldridge/HikariCP	18	26.4	22.4	84.7	132.9	95.2	96.6	0.8	92.9	1.5	4.8
opentripplanner/OpenTripPlanner	9	136.0	76.4	56.2	179.3	82.3	85.4	1.8	83.9	3.7	10.5
undertow-io/undertow	28	220.1	151.5	68.8	181.0	80.5	82.9	1.1	82.4	2.9	13.6
Graylog2/graylog2-server	14	187.6	25.8	13.8	284.0	103.5	106.3	1.8	6.0	2.6	89.6
apache/commons-pool	16	20.0	6.7	33.4	303.1	56.8	57.0	0.0	96.1	0.3	3.5
openmrs/OpenMrs	20	244.1	101.7	41.7	315.0	48.1	49.8	1.8	83.6	3.5	11.1
aws/aws-sdk-java	7	134.1	58.0	43.5	424.0	96.5	97.2	0.2	45.2	0.7	54.0
jankotek/mapdb	7	173.6	81.7	47.3	449.1	67.3	68.5	0.6	77.9	1.6	19.9
Average(LONG)	18.4	155.9	**54.1**	**40.5**	236.8	**68.2**	**69.7**	0.8	70.8	2.3	26.2
Average(OVERALL)	26.2	91.7	**29.4**	**35.2**	93.0	**81.0**	**87.6**	0.9	41.3	6.5	51.3

impacted tests that STARTS selected to run, relative to the number of all tests. We also measured the percentage of end-to-end time taken by STARTS relative to the end-to-end time for running all tests, i.e., RetestAll. The STARTS end-to-end time includes the time to compile, perform selection, run the impacted tests, and update dependencies for the next run, while the RetestAll time is compile time plus time to run all tests. We include compile time because, after a change, a continuous integration system, e.g., Travis [201], typically also compiles the application. We wanted to evaluate any savings in the overall build time when using STARTS. Table 4.1 shows for each project (sorted by increasing RetestAll time), the number of versions evaluated (SHAs [#]), average number of all tests across all versions (ALL), average number of tests selected by STARTS (Selected [#]), average percentage of all tests selected by STARTS (Selected [%]), RetestAll time (RTA[s]), and average percentage of RetestAll time that STARTS takes, for both the "online" mode (Online Time [%]) (Section 4.2.2) that includes time for the a, e, and g phases, and the "offline" mode (Offline Time

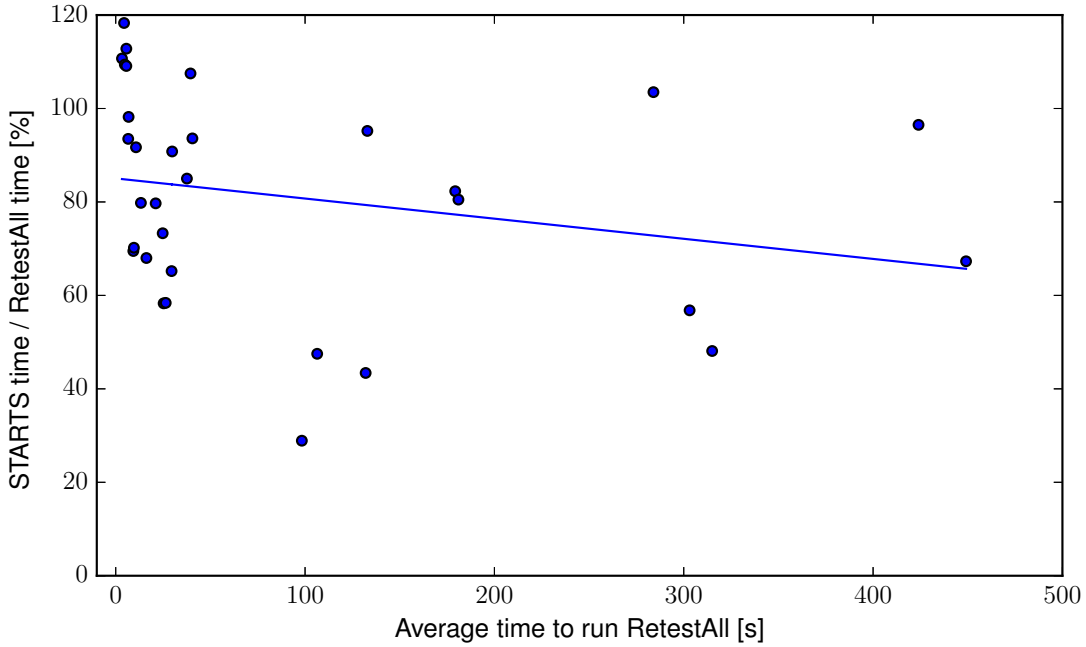


Figure 4.2: Correlation between project end-to-end test time vs. percentage of time to run STARTS

[%]) that excludes time for the g phase. The last columns break down STARTS time into a, e, g and compilation (Comp.) times. In the offline mode, a developer can get test results faster by not having to wait until STARTS finishes the computation of dependencies before seeing those test results—dependency and transitive closure computation can be removed from the developer's critical path.

We divide the projects in Table 4.1 into *short-running* if RetestAll takes less than one minute (upper part) and *long-running* if RetestAll takes more than one minute (lower part). Table 4.1 shows that STARTS runs fewer tests compared with RetestAll: STARTS selects between 4.4% (apache/commons-collections) and 84.7% (brettwooldridge/HikariCP) of all tests, with an average of 35.2% of all tests across all projects. Table 4.1 shows that STARTS also provides time savings, with an average end-to-end time of 81.0% of RetestAll time in the offline mode, and 87.6% of RetestAll time in the online mode. STARTS provides greater time savings for long-running projects (68.2% in the offline mode and 69.7% in the online mode) than for short-running projects (87.8% in the offline mode and 96.9% in the online mode). STARTS is *more* expensive than RetestAll (i.e., the offline percentage of RetestAll time is greater than 100%) in six (of 21) short-running projects and only one (of 11) long-running project. As expected, STARTS is better suited for long-running projects. Figure 4.2 plots the correlation between the average RetestAll time per project (x-axis) and the percentage time savings from the STARTS offline mode (y-axis); the Kendall-τ_b value is

-0.3, and $p < 0.01$, a weak negative correlation. Finally, the breakdown of the end-to-end time shows that STARTS spends most of its non-compilation time in the e phase (41.3% of end-to-end time, on average), while a and g take up much smaller percentages. The time for short-running projects is dominated by compilation and these projects likely cannot benefit much from any RTS, including STARTS

4.4 LIMITATIONS

In our previous study [117], we found that static RTS performed comparably with dynamic RTS (we evaluated against Ekstazi) in terms of time. However, we also found that static RTS is as expected, less precise than dynamic RTS and can be unsafe. (An RTS technique is *precise* if it selects to run *only* the impacted tests, and *safe* if it does not miss to select an impacted test.) We also found that reflection was the only cause of unsafety of static RTS when compared with Ekstazi. STARTS does not yet address these safety and precision limitations of static RTS. STARTS can be unsafe when the path between tests and changed types can only be reached via reflection, and is inherently imprecise because the static dependencies it finds among the types in the application may not be runtime dependencies. STARTS also assumes that there is no test-order dependence [77, 223].

4.5 SUMMARY

This chapter presented STARTS, our tool for static change-impact analysis and RTS. STARTS performs change-impact analysis and RTS at the class level, inspired by the results from Chapter 3. Our evaluation showed that change-impact analysis in STARTS is quite efficient—6.5% of end-to-end RTS time, on average. Therefore, we use STARTS as the change-impact analysis component in evolution-aware RV techniques, and for comparing and combining evolution-aware RV with RTS, as we discuss next in Chapter 5.

CHAPTER 5: EVOLUTION-AWARE RUNTIME VERIFICATION

The results from Chapter 2 showed that Runtime Verification (RV) can help to find many bugs by monitoring program executions against formal properties. Developers should ideally use RV whenever they run tests, to find more bugs earlier. However, the results from Chapter 2 also showed that, despite tremendous research progress over the last two decades, RV still incurs high overhead in (1) machine time to monitor properties, and (2) developer time to wait for and inspect violations from test executions that do not satisfy the properties. Moreover, all prior RV techniques consider only one program version and wastefully re-monitor unaffected properties and code as software evolves.

In this chapter, we present the first evolution-aware RV techniques that reduce RV overhead across multiple program versions. Regression Property Selection (RPS) re-monitors only a subset of properties, namely those that can be violated in parts of code affected by changes, reducing machine time and developer time overhead of RV. Violation Message Suppression (VMS) simply shows only new violations to reduce developer inspection time after code changes; it does not reduce machine time overhead. Regression Property Prioritization (RPP) splits RV in two phases: properties more likely to have violations that help find bugs are first monitored in a critical phase to provide faster feedback to the developers; the rest are monitored in a background phase. Both RPS and VMS utilize the class-level static change-impact analysis technique implemented in STARTS, as discussed in chapters 3 and 4, to compute the parts of code affected by the changes.

We compare our techniques with the evolution-*unaware* (*base*) RV when monitoring test executions in 200 versions of 10 open-source projects. RPS and the RPP critical phase reduce the average RV overhead from 9.4× (for base RV) to 1.8×, without missing any new violations. VMS reduces the average number of violations 540×, from 54 violations per version (for base RV) to one violation per 10 versions. Our evolution-aware RV techniques can be used together and they are complementary to techniques that make base RV faster on single program versions. We also evaluated to what extent regression test selection (RTS) alone can reduce the overhead of RV as software evolves, since RTS already selects to rerun a subset of tests after code changes. The RTS technique that we used for evaluation is the one that we implemented in STARTS, as discussed in Chapter 4. The results show that, compared with base RV, RTS alone did not achieve as much overhead reduction as, and should be combined with, our evolution-aware RV techniques.

```
1 Collections_SynchronizedCollection(Collection c, Iterator i) {
2  Collection c;
3  creation event sync after() returning(Collection c):
4   call(*Collections.synchronizedCollection(Collection)){ this.c = c;}
5  event syncMk after(Collection c) returning(Iterator i):
6   call(*Collection+.iterator())&&target(c)&&Thread.holdsLock(c){}
7  event asyncMk after(Collection c) returning(Iterator i):
8   call(*Collection+.iterator())&&target(c)&&!Thread.holdsLock(c){}
9  event access before(Iterator i) :
10  call(*Iterator.*(..))&&target(i)&&!Thread.holdsLock(this.c){}
11  ere: (sync asyncMk) | (sync syncMk access)
12  @match{ RVMLogging.out.println(/*violation message*/); }
13 }
```

(a) Collections_SynchronizedCollection (CSC)

```
1 StringTokenizer_HasMoreElements(StringTokenizer s) {
2  event hasnexttrue after(StringTokenizer s) returning(boolean b):
3   (call(boolean StringTokenizer.hasMoreTokens()) ||
4   call(boolean StringTokenizer.hasMoreElements()))&&target(s)&&b
5  event next before(StringTokenizer s):
6   (call(* StringTokenizer.nextToken()) ||
7   call(* StringTokenizer.nextElement())) && target(s)
8  ltl: [ ](next => (*) hasnexttrue)
9  @violation { RVMLogging.out.println(/*violation message*/); }
10 }
```

(b) StringTokenizer_HasMoreElements (STHME)

```
1 URLDecoder_DecodeUTF8() {
2  event decode before(String enc) :
3   call(* URLDecoder.decode(String, String)) && args(*, enc) {
4    if (enc.equalsIgnoreCase("utf-8") || enc.equalsIgnoreCase("utf8"))
5     return;
6    RVMLogging.out.println(/*violation message*/); }
7 }
```

(c) URLDecoder_DecodeUTF8 (URLD)

```
1  class A {
2   String a(List i, String sep) {
3    String o = "";
4    for (Object a : i) {
5     o += a.toString() + sep;
6    } return o; }}
7
8  class B extends A {
9   String b(List l) {
10   String i;
11 - i = a(l, " ");
12 + i = a(Collections.synchronizedList(l), " ");
13   return i.trim(); }
14  Boolean flag() { return true; }}
15
16 class C {
17  String c(List<String> l) {
18   B b = new B(); D d = new D();
19   String s = b.b(l);
20   return d.d(s, b.flag()) + ": " + s; }}
21
22 class D {
23  String d(String s, boolean flag) {
24  StringTokenizer t = new StringTokenizer(s);
25  String out = "";
26  if (flag) {
27   if(t.hasMoreTokens()){out=t.nextToken();}
28  } else { out = t.nextToken(); }
29  return out; }}
30
31 class E {
32  void e(String u,String e) throws Exception{
33   D d = new D(); assert(!u.isEmpty());
34   String url = d.d(u, false);
35   if (url.startsWith("https")) {
36     String s = URLDecoder.decode(url, e);
37     System.out.print(s); }}}
```

(d) Example evolving code

Figure 5.1: Example properties and evolving code that we use to illustrate base RV and evolution-aware RV techniques

5.1 EXAMPLE

Our running example is necessarily detailed to show the specifics of RV and programs that evolution-aware RV techniques exploit, and to highlight differences between the RPS variants in later sections. We illustrate properties that we monitor, how evolution-unaware base RV works in the JavaMOP [92, 99, 132] tool used in our experiments, and violations.

5.1.1 Examples of Monitored Properties

In our running example, we use the three properties in figures 5.1a–5.1c, written in JavaMOP syntax [90]; they helped find several confirmed bugs [118]. Properties have three parts: (1) *events*: relevant method calls or field accesses, (2) a *specification*: logical formula over the

events, and (3) a *handler*: action to take when events match (or violate) the specification.

Collections_SynchronizedCollection (CSC): CSC checks that code synchronizes on a `synchronized Collection` before iterating over it [42]. Not synchronizing on such `Collection` before iterating "may result in non-deterministic behavior" [41]. CSC defines four *events* in lines 3–10 of Fig. 5.1a: (1) `sync` (lines 3–4) occurs when `Collections.synchronizedCollection` is called to create a `Collection`, `c`, (2) `syncMk` (lines 5–6) occurs when `c.iterator` is called to obtain an `Iterator i` in a thread that holds the lock on `c`, (3) `asyncMk` (lines 7–8) occurs when `c.iterator` is called without first locking on `c`, and (4) `access` (lines 9–10) occurs when accessing `i` from a thread that does not hold `c`'s lock. When `sync` occurs, JavaMOP creates a *monitor* object to listen for CSC events (hence the `creation` keyword).

CSC's specification (line 11 of Fig. 5.1a) is an Extended Regular Expression which matches if the code either (1) creates `c` (`sync` event) and obtains `i` without first locking on `c` (`asyncMk` event), or (2) creates `c` (`sync` event) and obtains `i` from a thread that locks on `c` (`syncMk` event) but accesses `i` from a thread that does not lock on `c` (`access` event). When a CSC monitor receives an event that causes its specification to match, its handler (line 12) is invoked. The handler can be any code, but most properties, including CSC, just print a violation to warn developers of a potential bug.

StringTokenizer_HasMoreElements (STHME): STHME checks that getting tokens from `StringTokenizer`, `st`, is only done after checking that `st` has more elements [193]. STHME (Fig. 5.1b) defines two events: (1) `hasnexttrue` (lines 2–4) occurs when `st.hasMoreElements` or `st.hasMoreTokens` is invoked and returns `true`, and (2) `next` (lines 5–7) occurs when `st.nextElement` or `st.nextToken` is invoked. STHME specification (line 8) is a past-time LTL formula [129] stating that a `next` event on `st` must be preceded by a `hasnexttrue` event on `st`. When STHME specification does not hold, line 9 prints a violation.

URLDecoder_DecodeUTF8 (URLD): URLD checks that URLs are decoded from UTF-8, to avoid producing incompatible URLs [202, 203]. URLD's only event, `decode` (lines 2–5 in Fig. 5.1c), occurs if URL is decoded with non-UTF-8 encoding. The handler on line 6 prints a violation on each `decode` event.

5.1.2 Base RV, Causes of Overhead, and Property Violations

We describe the example Java code in Fig. 5.1d and violations that occur when JavaMOP is used to monitor its execution against the CSC, STHME, and URLD properties. The example code is hypothetical, created to illustrate our techniques.

Example Code: Fig. 5.1d shows five classes—`A`, `B`, `C`, `D`, and `E`—and two versions—line 11 in the old version is replaced with line 12 in the new version. `A.a()` concatenates the string

```
1  public class TC {
2   @Test public void testC() {
3    B b = new B(); C c = new C(); D d = new D();
4    List<String> l1 = Arrays.asList("1", "2");
5    assert(b.b(l1).equals("1 2"));
6    assert(c.c(l1).equals("1: 1 2"));
7    assert(d.d("1 2", false).equals("1")); } }
8
9  public class TE {
10  @Test public void testE() throws Exception {
11   E e = new E(); String u = "https://bing.com";
12   assert(e.e(u + " b", "ISO−8859−1").equals(u)); } }
```

Figure 5.2: Tests for code in Fig. 5.1d

representation of all elements in its input `List`. `B` extends `A` and `B.b()` invokes `A.a()` to get a string representation of the input `List`, which it then trims to remove leading or trailing white space. `C.c()` first invokes `B.b()` to obtain a string representation of its input `List`, which it prints after prefixing with the first sub-string, obtained from `D.d()`. `D.d()` tokenizes the input string and returns the first token; for performance reasons, it only checks that the input string has more than one token if its caller sets `flag` (e.g., the caller may already ensure non-emptiness). `E.e()` decodes an encoded HTTPS URL from a string after ensuring the string is not empty and invoking `D.d()` to get the first sub-string.

Monitoring and Causes of RV Overhead: We use the code in Fig. 5.1d to describe three RV concepts: instrumentation, monitor creation, and event/violation handling. Let us consider what happens when the tests in Fig. 5.2 are run on the old version of Fig. 5.1d. During class loading, JavaMOP instruments all statements in classes `A` through `E` that can generate events mentioned in the properties. The instrumentation causes events to be triggered during execution. Example instrumentation points in Fig. 5.1d include (1) before creating an `Iterator` on line 4 which may trigger CSC events, (2) after `hasMoreTokens` and before `nextToken` on line 27, and before line 28, which may all trigger STHME events, and (3) before line 36 which may trigger URLD events. At runtime, monitors are created to listen for and handle events. In the old version, only STHME and URLD monitors are created; `creation event` for CSC never occurs because `List l` on line 11 is not a `synchronized Collection`. One STHME monitor is created when the first relevant event occurs on each `StringTokenizer`; only one URLD monitor is created at the start of execution (unlike CSC and STHME, URLD has no parameters). Base RV induces high runtime overhead due to managing very many monitors and dispatching even more events to monitors [98, 132], e.g., with base RV, one project in our evaluation with 78 thousand lines of code created over 232 million monitors, which received almost 3 billion events.

Violations: When events occur that match or violate a monitor's specification, the violation

Specification Collections_SynchronizedCollection has been violated on line B.b(B.java:11). Documentation for this property can be found at https://runtimeverification.com/monitor/annotated-java/__properties/html/java/util/Collections_SynchronizedCollection.html
A synchronized collection was accessed in a thread-unsafe manner.

Figure 5.3: An example property violation

handler prints a violation, like in Fig. 5.3. A violation contains the violated property name, the location (i.e., fully qualified class name, method, source file name, and line number) of the last event that caused the violation, a URL for the property definition, and a sentence describing the violation. These help developers to reason whether a property violation is a true bug or false alarm.

We distinguish between *violation instances*, the *list* of violations, and the set of *violations*. Violation instances repeat, e.g., if property-violating code is in a loop or executed by multiple tests. We map violation instances of the same property that occur at the same location to the same violation. Developers may prefer to only see violations, but seeing all violation instances can help in debugging. Running tests in Fig. 5.2 on old version of Fig. 5.1d generates two violations from three violation instances. Lines 7 and 12 in Fig. 5.2 cause two instances of a STHME violation by executing `t.nextToken` on line 28 of Fig. 5.1d without calling `t.hasMoreTokens`. Line 12 in Fig. 5.2 causes one instance of a URLD violation by executing line 36 of Fig. 5.1d to decode a non-UTF-8 encoded URL. It can be time consuming to inspect/debug violations [118]. We next discuss evolution-aware RV techniques which aim to reduce runtime overhead of RV and show fewer violations as software evolves.

5.2 EVOLUTION-AWARE RV TECHNIQUES

We describe our evolution-aware RV techniques which leverage software evolution to reduce the runtime overhead of base RV across multiple program versions and to focus developers on new violations after a change. Base RV (illustrated through the example in Section 5.1) is evolution-unaware. For example, running base RV on the new version of code in Fig. 5.1d would re-monitor all available properties and re-incur the entire overhead wastefully because the code change does not affect (i.e., alter the behavior of) all classes, e.g., `E` is unaffected. Further, properties whose events are only generated from unaffected classes cannot have any new violations after the code change. Finally, it may be desirable to monitor on the developer's critical path, from when they launch tests to when they see the test results, only properties that are more likely to find bugs than others, e.g., based on a project's history.

Section 5.2.1 defines *safety* and *precision*, two notions that we use in this dissertation to analyze and measure the quality of our evolution-aware RV techniques. Section 5.2.2

describes RPS, our technique to re-monitor only properties that can have new violations after a code change, and also includes our definition of affected classes and how RPS uses affected classes to select the subset of properties to re-monitor in a new program version. Section 5.2.3 describes various RPS variants. Sections 5.2.4 and 5.2.5 describe our other two evolution-aware RV techniques, VMS and RPP, respectively. RPS, VMS, and RPP can be used separately or together, and we illustrate them throughout this section using the example from Fig. 5.1.

5.2.1 Safety and Precision

Safety measures loss in violation-finding (and thus potential bug-finding) ability. Precision measures minimality. We define safety and precision relative to base RV and *relevant* violations. In this dissertation, relevant violations are *new violations*—violations that are in the new version, but not in the old version, after accounting for violations that merely changed line numbers in the code. Definition 5.1 allows developers to plug in other notions of relevant violations.

Definition 5.1. *Relevant Violation: Relevant violations for an evolution-aware RV technique are those due to the changes.*

Definition 5.2. *Safety: An evolution-aware RV technique is safe if it finds* all *relevant violations that base RV finds.*

Definition 5.3. *Precision: An evolution-aware RV technique is precise if it finds* only *relevant violations that base RV finds.*

5.2.2 Regression Property Selection (RPS)

RPS reduces accumulated base RV overhead by re-monitoring only properties that can be violated in parts of code affected by changes [115]. For RPS to be useful, its end-to-end time (i.e., time to select properties plus time to re-monitor selected properties) must be less than base RV time. Thus, we consider changes and affected parts of code at the class-level granularity, which was more effective than only finer-granularity levels (e.g., statements or methods) for other evolution-aware techniques [69, 117, 218]. The reason is that the analysis at the class level achieved a better balance of efficiency (class-level analysis is faster than analyses at finer granularity) and precision (class-level analysis may capture more than necessary because it is coarser grained).

The notion of *affected classes* is central to RPS; it relates code changes with the properties. Intuitively, a property should be re-monitored only if its events can be generated from some class affected by the code change. That is, affected classes are those that can generate events that lead to new violations after code changes. Conversely, a class that is *unaffected* by a change cannot generate an event that leads to a new violation. Formally, RPS variants compute affected classes as those that satisfy some of the following conditions, which capture when a class may generate events that lead to new violations after a code change:

Definition 5.4. *Affected Class: For RPS, a class C is affected by a change if (1) C changed, (2) C transitively depends (via inheritance or use) on a class that changed, or (3) a class that satisfies (1) or (2) can pass objects to C.*

Condition 3 captures classes whose control flow may change (leading to new events and violations) if received objects change. For example, in Fig. 5.1d, `D` does not depend on the changed class (`B`) or its transitive dependents (`C` and `TC`); if only `B.flag()` changes to return `false` on line 14, then the "else" branch on line 28, instead of the "then" branch on line 27 will execute, leading to a STHME violation.

Definition 5.5. *Regression Property Selection (RPS): A technique to select and re-monitor, in a new program version, only properties that may have new violations.*

RPS has four steps: (1) construct a class dependency graph (intertype relationship graph) from the new program version, (2) find affected classes, (3) select properties, and (4) re-monitor selected properties.

Definition 5.6. *Class Dependency Graph (intertype relationship graph): A graph that has a node for each class in the program and an edge from class C to class C' if C depends on C' via inheritance or use.*

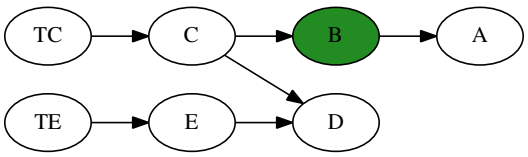


Figure 5.4: Class dependency graph (intertype relationship graph) for Figures 5.1d and 5.2. Edges mean "depends on"; the changed class is colored

Step 1: RPS constructs the intertype relationship graph in Fig. 5.4 for the new version of the code in Fig. 5.1d and the tests in Fig. 5.2.
Step 2: Strong RPS computes affected classes from the intertype relationship graph as $\mathit{affected}(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^* \circ E^*$, where $\mathcal{T}_c$ is the set of changed and new classes, E is the

set of edges in the intertype relationship graph, * is the reflexive and transitive closure, $\circ$ is the relational image, and $^{-1}$ is the inverse relation. Observe that *affected*($\mathcal{T}_c$) captures the three conditions in Definition 5.4. In our example, $\mathcal{T}_c = \{\texttt{B}\}$; only `B` changed (Condition 1). $\mathcal{T}_c \circ (E^{-1})^* = \{\texttt{B}, \texttt{C}, \texttt{TC}\}$; `TC` and `C` transitively depend on $\mathcal{T}_c$ (Condition 2). Lastly, *affected*($\mathcal{T}_c$) = $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{TC}\}$; `A` and `D` may generate new events due to changes to `B` or the interaction of `C` with `B` (Condition 3). $\texttt{E}, \texttt{TE} \notin$ *affected*($\mathcal{T}_c$) since they cannot generate new events. Although elided in our example due to space limits, newly added classes are in $\mathcal{T}_c$, so RPS re-monitors properties that may be violated in newly added classes.

Steps 3 and 4: RPS re-monitors only CSC and STHME in the new version. No (new) events for URLD are generated in affected. So, RPS saves the time to re-monitor URLD (if both tests are run) and developer time for (re-)inspecting URLD violations. Any URLD violations must be in `E` and cannot be new violations, because $\texttt{E} \notin$ *affected*($\mathcal{T}_c$).

Discussion of RPS: If a property was not instrumented into the old version, but code changes can cause it to be violated in affected, RPS selects it, e.g., CSC is selected by strong RPS. Two CSC violation instances occur in the new version in Fig. 5.1d; lines 4–6 iterate over the `synchronized Collection` initialized on line 12 without locking on it, matching the left disjunct in CSC's specification (line 11, Fig. 5.1a), so the handler (line 12) prints the violation in Fig. 5.3.

Base RV does not consider changes, dependencies, or classes that generate events for each property. After each change (e.g., from line 11 to line 12 in Fig. 5.1d), base RV re-monitors all properties and shows old and new violations. In our example, base RV shows three violations: the two STHME and URLD violations from the old version, plus the new CSC violation. RPS shows only the old STHME violation, plus the new CSC violation. Note that RPS by itself is not precise; it does *not* show only new violations. Showing only new violations is the goal of VMS (Section 5.2.4).

5.2.3 RPS Variants

RPS determines (1) *what* properties to select and (2) *where* in the program to instrument selected properties. The *strong RPS* described in Section 5.2.2 is safe under certain assumptions: it selects to re-monitor *all* properties for which events can be generated from *all* affected classes ("what") and instruments them throughout the program ("where"), including third-party libraries and even unaffected classes. However, that strong RPS variant is imprecise (it may instrument and monitor selected properties in unaffected classes). We describe here a second, more precise strong RPS variant. *Weak RPS* variants trade some safety for further overhead reduction. Weak RPS variants differ in what affected classes they

use for selecting properties and where they instrument selected properties.

Strong RPS Safety Assumptions: Strong RPS is safe under the following assumptions: (1) the intertype relationship graphis complete, (2) there are no test order dependencies [21, 77,223], and (3) dynamic language features, e.g., reflection and classloading, do not introduce additional intertype relationship graph edges.

Notation: Subscripts show how affected classes are computed. ps_1 computes $\mathit{affected}_1(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^* \circ E^*$ (Definition 5.4). ps_2 computes $\mathit{affected}_2(\mathcal{T}_c) = \mathcal{T}_c \circ ((E^{-1})^* \cup E^*)$, which consists of only classes that either depend transitively on $\mathcal{T}_c$ (dependents) or $\mathcal{T}_c$ transitively depends on (dependees); $\mathit{affected}_2$ is more unsafe than $\mathit{affected}_1$ because it omits condition 3 from Definition 5.4 to not include classes, e.g., `D` in Fig. 5.4, that may generate new events because they receive objects from dependents of $\mathcal{T}_c$. ps_3 relaxes Definition 5.4 even further by omitting condition 3; it computes $\mathit{affected}_3(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^*$, i.e., only dependents of $\mathcal{T}_c$.

Once the corresponding set of affected classes (affected) has been used to select the properties to re-monitor (namely properties whose events may be generated from $\mathit{affected}(\mathcal{T}_c)$), we obtain more variants by choosing "where" to instrument the selected properties. We can reduce where to instrument the selected properties, in order to obtain more reduction of base RV overhead, at two levels: (1) do not instrument the selected properties in unaffected classes in the program but still instrument all third-party library classes loaded into the JVM, and (2) do not instrument the selected properties in any third-party library class.

For the first level of instrumentation reduction, we use the superscript `c` to show that unaffected classes in the program (i.e., complement of $\mathit{affected}(\mathcal{T}_c)$) are not instrumented: ps_1^{c} excludes $(\mathit{affected}_1)^{\mathsf{c}}$, ps_2^{c} excludes $(\mathit{affected}_2)^{\mathsf{c}}$, and ps_3^{c} excludes $(\mathit{affected}_3)^{\mathsf{c}}$. To see the benefit of not instrumenting $\mathit{affected}(\mathcal{T}_c)^{\mathsf{c}}$, consider ps_1 and ps_1^{c}, which are both safe. ps_1^{c} is safe because unaffected classes cannot generate any new events or alter the sequence of events for the selected properties, so they cannot have new violations. However, ps_1^{c} can be more efficient and more precise (i.e., show fewer old violations) than ps_1 if selected properties can generate events from classes in $(\mathit{affected}_1)^{\mathsf{c}}$. For example, in the intertype relationship graph of Fig. 5.4, if a selected property p can generate events from $\mathtt{B} \in \mathit{affected}_1$ and $\mathtt{E} \in (\mathit{affected}_1)^{\mathsf{c}}$, and tests `TC` and `TE` are run, ps_1^{c} can save the time to monitor p in `E`. (Note that when safety assumptions of strong RPS do not hold, ps_1 is safer than ps_1^{c}; by instrumenting selected properties in unaffected classes, ps_1 can find some violations that ps_1^{c} miss.) On the other hand, not instrumenting $\mathit{affected}(\mathcal{T}_c)^{\mathsf{c}}$ can make weak RPS variants more unsafe—an weak RPS variant that instruments all classes has a chance to find some violations from instrumented classes that are not in the computed $\mathit{affected}(\mathcal{T}_c)$.

The second level of instrumentation reduction does not instrument any third-party library class. We denote weak RPS variants that exclude *all* third-party library classes with ℓ in

Table 5.1: "What" properties RPS variants select

What	ps_1	ps_2	ps_3
properties in Δ	✓	✓	✓
properties in dependents of Δ	✓	✓	✓
properties in dependees of Δ	✓	✓	✗
properties in dependees of dependents of Δ	✓	✗	✗

Table 5.2: "Where" RPS variants instrument properties

Where ($i \in \{1, 2, 3\}$)	ps_i	ps_i^{c}	ps_i^{ℓ}	$ps_i^{\mathsf{c}\ell}$
affected	✓	✓	✓	✓
$affected(\mathcal{T}_c)^{\mathsf{c}}$	✓	✗	✓	✗
third-party library classes	✓	✓	✗	✗

the superscript. For example, $ps_3^{\mathsf{c}\ell}$ means that $affected_3$ is used to select properties, classes in $(affected_3)^{\mathsf{c}}$ are not instrumented and third-party library classes are also not instrumented. ps_3^{ℓ} means that $affected_3$ is used to select properties and only third-party library classes are not instrumented. In sum, we evaluate strong RPS (ps_1, ps_1^{c}) and 10 weak RPS variants: ps_2, ps_3, ps_2^{c}, ps_3^{c}, ps_1^{ℓ}, ps_2^{ℓ}, ps_3^{ℓ}, $ps_1^{\mathsf{c}\ell}$, $ps_2^{\mathsf{c}\ell}$, and $ps_3^{\mathsf{c}\ell}$. Tables 5.1 and 5.2 distinguish RPS variants in terms of what part of the intertype relationship graph is used for selecting properties, and where the selected properties are instrumented; ✓ means inclusion and ✗ means exclusion.

Efficiency/Safety Tradeoff: Weak RPS variants trade some safety for lower runtime overhead. Fig. 5.5 shows two lattices of RPS variants; lower variants can be less safe (left lattice) but more efficient (right lattice) than higher ones. ps_2 computes $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{TC}\}$ as $affected_2$. `D` is not in $affected_2$, so ps_2 can miss new STHME violations, e.g., when changing only `true` to `false` on line 14 in Fig. 5.1d—ps_2 does not even re-monitor STHME for this change. If such cases are rare, then ps_2 can be safe but have lower overhead than strong RPS. In general, ps_2 can be unsafe if there is data flow to classes that are not dependents or dependees of $\mathcal{T}_c$. ps_3 computes $affected_3$ as $\{\mathtt{B}, \mathtt{C}, \mathtt{TC}\}$—dependents of $\mathcal{T}_c$—which includes

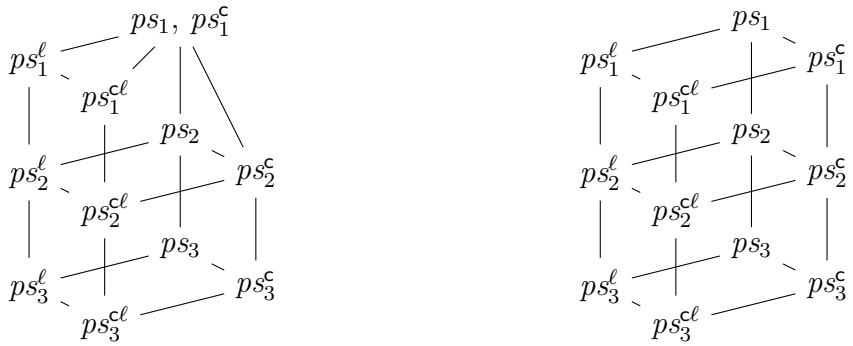


Figure 5.5: Lattices of RPS variants. Left lattice ordered by "less safe than". Right lattice ordered by "more efficient than"

neither $\mathcal{T}_c$'s dependees, e.g., `A`, nor dependees of $\mathcal{T}_c$'s dependents, e.g., `D`. Therefore, ps_3 can be more unsafe than ps_2, e.g., if `A` changes such that a new violation results from events that are local to `A`, ps_2 will find that new violation, but ps_3 will not find it. Technique ps_3^{c} could miss new violations that ps_2^{c} finds. In fact, ps_3^{c} misses the new CSC violation after the change in Fig. 5.1d; it selects to re-monitor CSC but does not instrument `A`, so `asyncMk` is not triggered. Excluding third-party library classes from instrumentation can also be unsafe, e.g., if `A` or `D` are third-party library classes. Weak RPS variants that do not instrument $\mathit{affected}(\mathcal{T}_c)^{\mathsf{c}}$ can be faster but safe if changes only lead to new violations in $\mathcal{T}_c$ and its dependents. Lastly, for weak RPS, false alarms can result from excluding classes from instrumentation. For example, if the STHME property's `hasnexttrue` event is triggered from a third-party library class that is not instrumented, and the `next` event is triggered in $\mathit{affected}(\mathcal{T}_c)$, a violation will occur even though the program satisfies the STHME property.

5.2.4 Violation Message Suppression (VMS)

VMS improves base RV by showing only new violations. Showing only new violations right after a change is more effective than showing old plus new violations to get developers to act—they are still in the mental context of the change and are the ones who can best address new violations [146].

Definition 5.7. *Violation Message Suppression (VMS): A technique to show, in a new program version, only new violations that did not occur in an old version.*

Base RV shows three violations in the new version of the example in Fig. 5.1d: line 4 (two instances), line 28 (two instances), and line 36 (one instance). The latter two were in the old version, and their line numbers did not change. (More generally, VMS does not simply check equality of line numbers but builds a likely mapping between old and new line numbers based on code context.) In the new version, VMS shows only the violation on line 4, instead of showing all three. VMS can be used with RPS to reduce the old violations shown by RPS. Running RPS on new version of Fig. 5.1d will show two violations (lines 4 and 28); VMS shows only one (line 4).

VMS' inputs are the violations from the old and new versions, plus the source files in both versions. Each violation, $v = \langle p, c, l \rangle$, contains a triple of the property name (p) that was violated, and the class (c) and line number (l) of the last event that violated the property. Let V_1 and V_2 be the set of violations from monitoring the old version (P_1) and the new version (P_2), respectively. VMS computes V_{new}, the set of new violations that are in P_2 but not in P_1. VMS does not simply compute $V_2 \setminus V_1$ that may report many old violations for

which only the line numbers changed. Using only line numbers to match statements in two code versions performs poorly [168].

For each class $C_\delta \in \mathcal{T}_c$, where $\mathcal{T}_c$ is the set of changed classes (including newly added and renamed classes), VMS first creates a mapping, M_{C_δ}, from line numbers in the source file of C_δ in P_2 to line numbers with the likely same statement in the corresponding source file in P_1. Each line number in P_2 maps to at most one line number in P_1; some line numbers in P_2 may not be in M_{C_δ}. Note that M_{C_δ} is likely (i.e., not exact) as it is based on simple syntactic and not semantic equivalence; the latter is rather challenging and does not scale currently [75,128]. M_C is identity if C did not change. Then, $V^{\mathcal{T}_c}_{new}{=}\cup_{C_\delta \in \mathcal{T}_c} VMS(V_1, V_2, M_{C_\delta})$, where $VMS(V_1, V_2, M_{C_\delta}){=}\{\langle p, C_\delta, l\rangle{\in}V_2 \mid \nexists l'{\in}M_{C_\delta}(l) \vee \langle p, C_\delta, M_{C_\delta}(l)\rangle{\notin} V_1\}$. Let $\mathcal{T}'_c$ be the set of unchanged classes. New violations in $\mathcal{T}'_c$ are $V^{\mathcal{T}'_c}_{new}{=}\cup_{C\in\mathcal{T}'_c} VMS(V_1, V_2, M_C)$. $V_{new} = V^{\mathcal{T}_c}_{new} \cup V^{\mathcal{T}'_c}_{new}$ is the output of VMS. $V^{\mathcal{T}'_c}_{new}$ is non-empty when interactions with changed classes cause new violations in $\mathcal{T}'_c$, or when test non-determinism i.e., "flakiness" [20,21,77,132,181] leads to non-determinism during monitoring.

Discussion of VMS: VMS can save developer time for inspecting violations but slightly increases machine time, e.g., VMS increases time by <1% in our experiments. As we showed with our example, VMS can further reduce violations shown by RPS.

5.2.5 Regression Property Prioritization (RPP)

Developers may be more interested in violations of critical properties than other violations, e.g., violations of properties that previously helped find bugs may be more critical. RPP partitions RV into two phases: a *critical* phase and a *background* phase. After a code change, the critical phase immediately re-monitors (manually or automatically selected) critical properties and provides results to developers. The background phase separately re-monitors other properties. Developers get delayed feedback if non-critical properties are violated. RPP allows (manually or automatically) moving properties between the phases as properties become more or less critical during software evolution. To evaluate RPP, we consider previously violated properties as critical. RPP is inspired by regression test prioritization [49,86,175,190,215], but we are first to propose RPP for reducing RV overhead as software evolves.

Discussion of RPP: The benefit of RPP is to remove the re-monitoring of non-critical properties from developers' critical path (from the moment of submitting code changes to the moment of getting feedback). RPP's disadvantage is that it delays the time for developers to get feedback if non-critical properties are violated. RPS and VMS can be used with RPP—RPP merely first runs some subset of selected properties.

5.3 IMPLEMENTATION

We present our implementation of RPS, VMS, and RPP.

5.3.1 Regression Property Selection (RPS)

Building IRG, Computing Changes and Affected Classes: We used STARTS [117, 119] to build intertype relationship graphs, compute $\mathcal{T}_c$, find *affected*($\mathcal{T}_c$) in P_1, and persist checksums of classes in P_1 to disk. The checksums are used to compute the classes that changed between P_1 and P_2. STARTS is a publicly available regression test selection (RTS) tool that implements most of these steps. By default, STARTS computes $affected_3$, which suffices for RTS [107,117,119,149], but is not sufficient for strong RPS. We extended STARTS to compute $affected_1$ and $affected_2$. We chose STARTS because it is static and fast—it requires neither test runs nor code instrumentation to find dependencies among classes, or compute *affected*($\mathcal{T}_c$). We monitor test executions, so using a dynamic technique to compute dependencies or *affected*($\mathcal{T}_c$) would incur additional overhead. Also, instrumentation performed by a dynamic technique could interfere with JavaMOP instrumentation.

Monitoring: We used JavaMOP [99, 132] to monitor test executions against formal properties. JavaMOP is publicly available [92], uses AspectJ for load-time instrumentation, and allows monitoring many properties in one execution. JavaMOP was used in several RV studies [26, 45, 89, 115, 118, 132, 163, 166]. In each version, we follow publicly available instructions [91] to build and attach a JavaMOP agent [145] with selected properties to the JVM that executes tests.

Selecting Properties to Re-monitor: The properties re-monitored are those for which *affected*($\mathcal{T}_c$) can generate events. To select properties, we first used the AspectJ compiler to very quickly and statically weave all available properties into *affected*($\mathcal{T}_c$), and record properties whose aspects get weaved. If aspects from a property do not get weaved into any class in *affected*($\mathcal{T}_c$), its events cannot be generated from *affected*($\mathcal{T}_c$) at runtime. Time to select properties is part of RPS end-to-end time, so we optimized static weaving to be as fast as possible—only 3.3s on average in our experiments.

5.3.2 Violation Message Suppression (VMS)

VMS implementation is straightforward: (1) take violations from P_1 and P_2, (2) remove violations generated in P_2 if line mapping can map the same violation to a likely corresponding line number in P_1 (after taking care of renames), and (3) report any remaining

Table 5.3: Projects in our study

Name	#Test	KLOC	t[%][s]	t_{mop}/t_{tests}
commons-dbcp	26	20.1	56.5	2.0
imglib2	74	44.2	11.3	3.7
commons-lang	130	69.5	22.4	3.9
jackson-core	79	31.7	11.2	5.6
commons-io	96	29.2	106.5	5.8
commons-math	432	180.4	93.6	6.4
imaging	63	37.6	18.4	6.4
javapoet	17	7.9	10.7	7.2
stream-lib	24	8.4	127.1	12.2
opentripplanner	126	78.7	55.1	40.5
$\overline{\mathbf{x}}$	106.7	50.7	51.3	9.4

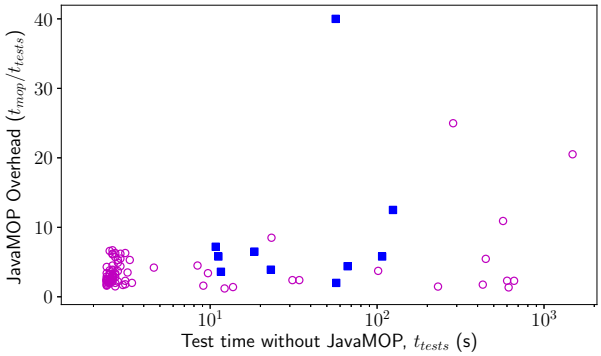


Figure 5.6: Test time vs. base RV overhead for several projects

violations generated in P_2 as likely new violations. Our line mapping extends the jDiff utility of jEdit [96], a Java implementation of Myers' classic algorithm [142].

5.3.3 Regression Property Prioritization (RPP)

We considered critical properties to be those that were violated in the project's history. In the first version of each project, there is no history, so there is a choice to monitor all properties in either the critical or background phase. It is not clear which of these choices is better; monitoring all properties in the either phase for the first version unfairly increases its average overhead. Therefore, we split properties into critical and background phases after the first version, depending on whether they were violated in the first version. We do not include the first version when computing the average overheads of each phase. From the second version onward, if a property gets violated in the background phase, our RPP implementation moves it to the critical phase in the next version. We leave it as future work to investigate criteria for moving properties which have not been violated after a while from the critical phase back to the background phase.

5.4 EVALUATION

Before we answer the research questions, we describe our experimental setup.

5.4.1 Research Questions

We answer the following research questions:

- **RQ1:** How much does RPS reduce the machine time overhead of base RV?

- **RQ2:** How many violations does VMS show and how safe are RPS variants?
- **RQ3:** How much does RPP reduce time for developers to get feedback on critical properties?

5.4.2 Experimental Setup

Projects: Table 5.3 shows 10 open-source, Maven-based Java projects from GitHub used in our study, 9 of which we also used in prior work [117, 119, 182]. #Test is average number of test classes used (we skipped very few test classes from 6 projects due to problems with JavaMOP instrumentation), KLOC is average thousands of lines of code, t[%][s] is average test time, and t_{mop}/t_{tests} is average base RV overhead.

Properties: We used 199 manually written properties found to be good in our prior study [118]. The properties were written to formalize Java APIs [113, 132] and are publicly available [161].

Versions: We started from a recent commit in each project and went back into the history, to select 20 commits/versions where (1) at least one `.java` file changed, (2) all tests pass without JavaMOP, and (3) all tests pass with JavaMOP.

Running Experiments: We wrote scripts to automate running tests, collect violations and measure time for three configurations on each version: (1) without JavaMOP, (2) with base RV, and (3) with each evolution-aware RV technique. For RPS, the most common case is that `.java` file changes modify the bytecode, so properties may need to be re-monitored. If `.java` file changes do not modify bytecode, we skip tests (no re-monitoring); time is only spent to check for changes. If changes affect bytecode, but no properties are selected to be re-monitored, all tests are run without JavaMOP, and the end-to-end time is the time to compute changes, find *affected*($\mathcal{T}_c$), check if properties need re-monitoring, and run tests.

5.4.3 RQ1: Overhead reduction from RPS

We present RV overhead in multiples (×), as the ratio t_{mop}/t_{tests}, where t_{mop} is time with JavaMOP and t_{tests} is time without JavaMOP. We first show on a sample of 89 projects whether the high overhead induced by base RV can be seen in open-source projects with short- (<10s), medium- (10s–300s), and long-running (>300s) tests. These 89 projects were sampled from our prior studies [117–119, 182] and from the Apache continuous integration server. Fig. 5.6 plots t_{tests} (x-axis, *log* scale, in seconds) vs. t_{mop}/t_{tests} (y-axis). Projects in all three categories exhibit high overhead, so high base RV overhead is not a fixed cost that is more pronounced in projects with shorter-running tests. Squares show projects in this

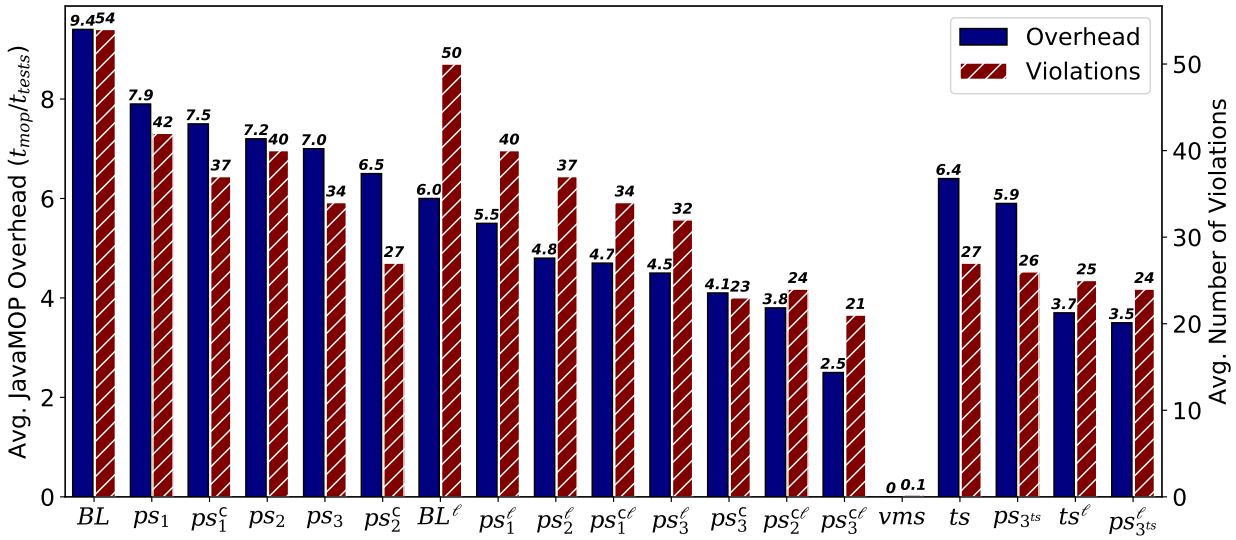


Figure 5.7: Runtime overheads of, and violations from base RV (BL), RPS variants (ps), VMS, and RTS (ts) with 161 properties

study. We did not evaluate our techniques on the other projects because (1) $t_{mop} - t_{tests}$ is too small for RPS to be beneficial, (2) test-running times are high for long-running projects which requires more resources than we have to evaluate them, or (3) we could not get 20 versions that satisfy our criteria.

Solid bars in Fig. 5.7 show average runtime overhead of base RV (BL) and the RPS variants (ps) discussed in Section 5.2.3. All overheads are computed from end-to-end time including time for analysis, running tests, and monitoring test executions. The results show several points. First, all RPS variants reduced the average base RV overhead, which is 9.4×. Strong RPS variants, ps_1 and ps_1^{c}, have 7.9× and 7.5× overhead, respectively. As expected, weak RPS variants with fewer classes in $\mathit{affected}(\mathcal{T}_c)$ achieve more reduction. $ps_3^{\mathsf{c}\ell}$ is the most efficient weak RPS variant, with 2.5× overhead. Second, comparing BL and BL^{ℓ} shows that base RV spends about 36% of overhead on third-party library code: $(BL-BL^{\ell})/BL$. Since ps_1^{c} is safe under certain assumptions, and, as we show in Section 5.4.4, excluding unaffected and third-party library classes was safe in our experiments, $ps_1^{\mathsf{c}\ell}$ may, in general, achieve the best efficiency/safety tradeoff among weak RPS variants.

We also evaluated how much regression test selection (RTS) [33, 52, 53, 69, 72, 81, 117, 174, 215, 218] can reduce base RV overheads during software evolution. RTS is a general approach (independent of RPS) for reducing the overhead of regression testing by re-running only a *subset of tests* whose behavior can differ after code changes. We previously noted that RTS can also reduce base RV overhead, since RTS already reduces testing overhead [115]. We evaluate a static class-level RTS technique, implemented in STARTS, which uses the same

intertype relationship graphas RPS. (Other RTS techniques compute dependencies dynamically [53, 69, 218], but it was challenging to evaluate them because their instrumentation often clashed with JavaMOP.) Since RTS can select more tests than those whose behavior differs after a code change, it may be imprecise as an evolution-aware RV technique (Section 5.2.1). So, we also evaluated RPS plus RTS to see how these two can together further reduce base RV overhead.

The four rightmost solid bars in Fig. 5.7 show the overhead with RTS, with and without libraries. *ts* shows combination of RTS with base RV, i.e., rerun a *subset of tests* but re-monitor *all properties*, while $ps_{3^{ts}}$ shows RPS (using variant ps_3) plus RTS, i.e., rerun a *subset of tests* and re-monitor a *subset of properties*. When measuring the overhead, we used end-to-end RTS time, which includes the time to select the tests. Combining base RV with RTS has 6.4× overhead, compared with 9.4× for base RV. RPS plus RTS gives lower overhead (5.9×) than RTS alone, showing that RPS can provide value even where RTS is used. Since RTS can be unsound, it may incorrectly miss to select tests [52, 69, 117, 174], which makes RTS an unsafe evolution-aware RV technique. Finally, as we show in RQ2, RTS by itself is imprecise; it should be combined with VMS to show only new violations.

5.4.4 RQ2: VMS and RPS Safety

We discuss the results of VMS and how we used these results to guide our manual checking of RPS safety. The striped bars in Fig. 5.7 show average number of violations from all techniques evaluated in Section 5.4.3; the *vms* bar shows VMS average. The most significant result is that VMS is orders of magnitude more precise than RPS. For four projects, no new violation occurred in the range of versions.

Further, library exclusion results in very little difference in the average number of violations, which is good, because most violations can still be found when libraries are excluded. Very few violations in the libraries makes sense because libraries are widely used and tend to be better tested. We manually checked all violations that are not generated when libraries are excluded and found 87.5% of them to be in the third-party libraries themselves, and not in the project code. Among these, only one was a new violation and it was due to a library version change—all events leading to the violation were in the new version of the library, so library exclusion did not lead to missing any new violation in the projects. All violations that are missed in projects when excluding libraries were not new violations, providing some justification for excluding libraries when one does not care about violations in libraries, and partial explanation for why library exclusion did not lead to missing new violations.

RPS Safety: We manually confirmed safety of RPS variants by checking if all new violations

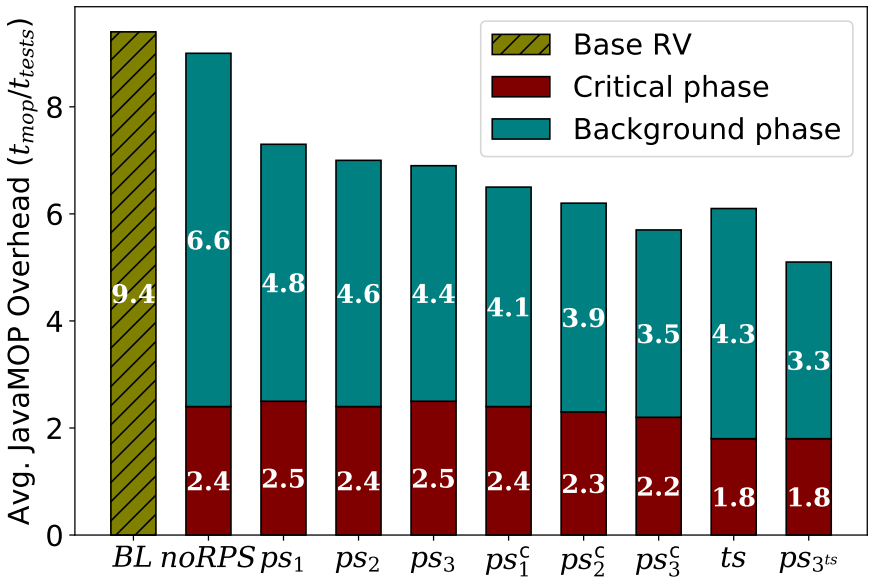


Figure 5.8: Runtime overheads with Regression Property Prioritization's critical (*cr*) and background (*bg*) phases

from VMS were also reported by each variant. VMS reported a total of 33 new violations in 16 of the 200 versions across all projects. Of these, 5 were due to flakiness, which we confirmed by re-running several times (i.e., these 5 violations could also have happened in the old version). All RPS variants found 27 of the new violations, but all missed one new violation. The one new violation that all variants missed was the aforementioned library version change. It is surprising that weak RPS variants were safe in our experiments, since they are theoretically unsafe. Therefore, we carried out further manual inspection of the changes involved in the 16 versions with new violations, to see why weak variants were safe. We found that all new violations happened due to events in $affected_3$, a subset of $affected_1$ and $affected_2$, so all the variants were able to catch the new violations. This further explains why weak RPS variants were safe in our experiments (Section 5.4.3 showed that excluding library classes did not lead to missing new violations in the projects).

5.4.5 RQ3: RPP Effectiveness

Fig. 5.8 shows RPP results, where the RPS variants do not exclude libraries. The first finding is that RPP alone (*noRPS*) overhead for monitoring background properties (*bg*) is roughly three times more than its overhead for monitoring critical properties (*cr*). The second finding is that overheads for *cr* and *bg* do not sum up to base RV overhead. For all projects, but one, *cr*+*bg* of *noRPS* is greater than *BL*. Being greater is expected because

time to run tests is repeated between *cr* and *bg*. The surprising project is opentripplanner, where the sum of *cr* and *bg* is less than *BL* for monitoring all properties together, likely due to reduced memory pressure when the properties are split. Finally, *cr* with $ps_{3^{ts}}$ has only 1.8× overhead, compared with 9.4× for base RV.

5.5 DISCUSSION

We highlight internal details of RPS and properties that contributed the most monitors and events in each project.

RPS Internals: Our analysis of data from running RPS shows that changes are small compared to the size of the program, and that our analysis is very fast compared to the time for monitoring. The data here is for ps_1. The average intertype relationship graph in our experiments had 720 nodes and 3706 edges. On average, $\mathcal{T}_c$ contained 7 nodes each version, leading to an average of 233 nodes in *affected*($\mathcal{T}_c$). The total analysis time was 4.3% of the end-to-end time—this includes the time to find *affected*($\mathcal{T}_c$), repackage the Jar file with selected properties, and to find the classes from which new events may be generated after a code change. The rest of the time is spent on monitoring.

Different properties dominated base RV overhead: We measured the number of monitors created, events triggered, and the top two properties that contribute the most monitors (Top M) and events (Top E). No property always dominated monitor creation or event generation, but two properties, `Iterator_HasNext` and `StringBuilder_ThreadSafe` are in Top M for all projects. No property dominates Top E. `Iterator_HasNext` and `StringBuilder_ThreadSafe` are quite common in Top E and generate most events in opentripplanner, the project with the highest base RV overhead. `Iterator_HasNext` helped find several bugs [118], so developers may still want to monitor it. We are not aware that `StringBuilder_ThreadSafe` previously helped find bugs.

5.6 THREATS TO VALIDITY

Internal: Due to non-determinism, program behavior may change from one run to another, even on the same program version. Such non-determinism may be intended in the code under test (e.g., threads could interleave in different ways in different runs), due to unintended test flakiness [20, 77, 108, 131, 153, 181, 184], or even due to program interactions with JavaMOP. We mitigated this threat by ensuring that the results were consistent across many experimental runs. The scripts and tools that we used in our experiments may contain

bugs. To mitigate this threat, multiple authors of our paper [116] reviewed the scripts, and we used JavaMOP and STARTS, which are both robust open-source tools which continue to be improved.

External: Our findings may not generalize to other RV tools, programs written in other programming languages, or programs which evolve differently than the ones we used. Certain aspects of our experimental design help address these threats. First, we evaluated JavaMOP, which is a popular and representative RV tool that also incorporates some ideas from other RV tools. Second, our evolution-aware RV techniques are not Java-specific and should apply to other programming languages. Lastly, we leave as future work to explore longer and more varied evolution histories of open-source projects.

Construct: STARTS, the tool that we used for static change-impact analysis, can miss to report some classes as affected if the dependencies between classes and the changed code happen only through dynamic language features such as reflection [117]. However, current results show that our evolution-aware RV techniques are safe. Further, our recent work on making STARTS reflection aware [183] began to address this threat.

5.7 SUMMARY

This chapter presented three evolution-aware RV techniques that we implemented to reduce RV overheads during software evolution—Regression Property Selection (RPS), Violation Message Suppression (VMS), Regression Property Prioritization (RPP)—and explored their efficiency/safety tradeoff, compared with state-of-the-art evolution-*un*aware RV (i.e., base RV) that monitors programs from scratch after every code change. Our techniques reduced base RV overhead from 9.4× to as low as 1.8×, were safe, and showed two orders of magnitude fewer violations than base RV. Our results provide strong evidence that taking evolution into account can help to significantly reduce base RV overhead when used during regression testing of evolving software.

CHAPTER 6: RELATED WORK

We describe research related to our work on properties (Section 6.1), change-impact analysis (Section 6.2), RTS (Section 6.3), STARTS (Section 6.4), and RV (Section 6.5).

6.1 STUDIES OF PROPERTIES

The properties used in prior research were either manually written [89, 113, 132] or automatically mined [19,37,43,63,105,111,112,122,143,144,155–157,165,207,209,224]. Robillard et al. [171] provide a survey of automatic property mining techniques. In this dissertation we studied the largest set of publicly-available manually written properties (those from Lee et al. [113,132,161]) and automatically mined properties (those from Pradel et al. [156,159,160]) that we could find. The sets of manually written and automatically mined properties that we used are properties of the standard Java library API.

Previous research considered how to mine properties with lower false alarm rates (FAR). Weimer and Necula [208], Le Goues and Weimer [111], and Yang et al. [212] use heuristics to filter candidate mined properties to produce properties which have lower FAR [111, 208] or which may be more interesting to developers [212]. Gabel and Su [64] proposed an automated technique for testing that mined properties are necessary for program correctness. Our study (Chapter 2) is different in focus, scale, type and, expressiveness of properties. Whereas the aforementioned research focused on increasing the likelihood of mining properties with low FAR, we focus on the bug-finding effectiveness of existing properties during RV of test executions. We use a larger number of open-source projects and tests. Further, properties used in the aforementioned research are project-specific, while we use project-agnostic properties of the standard Java library. Lastly, properties in the aforementioned research are mostly of the pattern $(ab)^*$—a occurs before b; a and b are events on the same object/type. We check properties that are more expressive (they are not limited to two events and can be expressed in different formalisms) and general (they can relate events on multiple objects/types).

Most of the literature on property mining that we surveyed (see Section 2.2.2) was either evaluated on benchmarks (e.g., 7 of 17 papers were evaluated on DaCapo) or on a small set of open-source projects (6 of 17 papers were evaluated on 4–7 open-source projects). Some papers did not use any projects but instead the mined properties were evaluated (in 4 of 17 papers) by means of recall and precision against existing properties derived from a small collection of classes or from prior work. Evaluating properties on a larger set of open-source projects can provide a more indicative picture of the bug-finding effectiveness

of the properties in code that developers commonly write.

Our *process* for evaluating mined properties is similar to Doc2Spec [224] whose authors obtained violations of mined properties in 138 open-source projects, manually filtered out false alarms (73.9% of the violations) and reported suspected bugs to the developers of the projects. Our work is different (1) in the way that the violations are obtained (they perform static analysis of selected client code that use the API from which the properties are mined, whereas we find dynamic violations while running tests that shipped with our subject projects), (2) in scope (they evaluate automatically mined properties, but we evaluate both manually written and automatically mined properties), and (3) in purpose (their goal was to evaluate *specifically* how good Doc2Spec is for finding bugs, whereas our goal is to evaluate how good *generally* are properties when they are monitored during test executions in open-source projects).

6.2 CHANGE-IMPACT ANALYSIS

Change-impact analysis is a well-studied topic concerned with identifying parts of code that may be affected by code changes [7, 30, 31, 66, 75, 148, 169, 179]. We provided a brief background on change-impact analysis in Section 3.3. Lehnert [121] provides a richer taxonomy of change-impact analysis techniques. Further, several reviews, surveys, and studies of change-impact analysis literature are available, e.g., by Cai and Santelices [36], by Lehnert [120], and by Li et al. [126]. Change-impact analysis is central to evolution-aware RV techniques. In this dissertation, we investigated the use of the firewall technique [107, 149] for computing the affected parts of code on which to focus RV after code changes.

6.3 REGRESSION TEST SELECTION

Our idea for evolution-aware RV [115] was inspired by RTS [69, 81, 107, 149, 169, 174, 176, 220], an evolution-centered technique for reducing regression testing costs. RTS has been studied for more than three decades. In this dissertation, we described how the change-impact analysis technique that we implemented works and how the change-impact analysis gives us a static class-level RTS tool, STARTS, for free. We described all of these, together with the large-scale evaluation that we performed, in chapters 3 and 4. Further, our implementation of STARTS allowed us to evaluate our evolution-aware RV techniques in settings where RTS is already being used (Section 5.4.3). To the best of our knowledge, the work presented in chapters 3 and 4 is the first extensive evaluation of class- and method-level static

RTS techniques and their comparison with the state-of-the-art class-level dynamic RTS for modern software projects. We discuss in this section some related work on dynamic and static RTS. Other researchers published more comprehensive surveys on regression testing in general, and RTS in particular [24, 51–53, 172, 215].

Dynamic RTS: Rothermel and Harrold [173, 174] investigated RTS for C programs. They dynamically collected coverage on the old version to perform RTS using a control-flow graph (CFG) analysis. Harrold et al. [81] further extended this work to handle Java language features and incomplete programs (their technique can work without requiring the analysis of third-party libraries that a project depends on). Because CFG analysis can be time consuming for large software, Orso et al. [149] proposed a two-phase analysis, a partitioning phase to filter out non-affected classes from an Intertype Relation Graph and a selection phase to perform CFG analysis only on classes retained after the first phase. They also evaluated a class-level RTS technique (called "HighLevel"), but it did not compute transitive dependencies on use edges; moreover, it computed dependencies on the supertypes of changed types and not only on the subtypes.

To improve the efficiency of dynamic RTS, a number of techniques at coarser-granularity levels (e.g., method- or class-level) rather than the finer-granularity CFG level were proposed. Ren et al. [169] and Zhang et al. [220, 221] applied change-impact analysis at the method-level, based on call graphs, to improve RTS. Recently, Gligoric et al. [69] proposed Ekstazi, a dynamic RTS technique at the class/file level. Although Ekstazi performs coarser-level analysis and may select more tests than prior work, it was demonstrated to have a sufficiently lower end-to-end testing time to be adopted by some open-source projects. Zhang [218] proposed a hybrid class- and method-level dynamic RTS approach, and showed that it is more precise and faster than Ekstazi. While these dynamic RTS techniques can be safe, they require dynamic test coverage information which may be absent, costly to collect, or require prohibitive instrumentation (e.g., for non-deterministic or real-time code). All of these will likely make it costly to combine dynamic RTS with RV. Therefore, in our work, we investigated static RTS techniques.

Static RTS: Kung et al. [107] first proposed static RTS based on class firewall, i.e., the statically computed set of classes that may be affected by a change. Ryder and Tip [176] proposed a call-graph-based static change-impact analysis technique and evaluated only one call-graph analysis (0-CFA) on 20 versions of one project [170]. Badri et al. [11] further extended call-graph-based change-impact analysis with a CFG analysis to enable more precise impact analysis, but they did not evaluate it on RTS. Skoglund and Runeson [187, 188] performed a case study of class firewall, but they used dynamic coverage information together with class firewall to perform RTS, whereas we apply the class firewall purely statically.

Although the literature contains many static RTS techniques, extensive studies of these techniques are lacking. In particular, prior to our study (Chapter 3), evaluations of static RTS techniques on modern open-source projects were lacking, so it was not clear how static and dynamic RTS techniques compare in terms of safety, precision, and overhead. Our results show that the static RTS that we implement in STARTS performs similarly as the state-of-the-art dynamic RTS approach, Ekstazi, in terms of overhead. However, the results also show that, compared with Ekstazi, STARTS is less precise and unsafe due to reflection in a small number of cases. We have recently started exploring ways to make static RTS safer with respect to reflection [183]. This and other potential improvements to STARTS (see Section 7) will benefit the evolution-aware RV techniques that we build on top of STARTS. The static RTS techniques presented in this dissertation are representative of all prior work on class-firewall-based analyses [107] and call-graph-based analyses [176].

6.4 ON THE GROWING RESEARCH IMPACT OF STARTS

Since the time we evaluated and released STARTS, there have been other projects that build on or use STARTS. Thus, there is a growing body of work that could benefit from safety (and precision) improvements of static RTS. Chen and Zhang [39] studied the use of STARTS for reducing mutation testing costs. We used STARTS as a central component of techniques for evolution-aware RV [115]—STARTS was the change-impact analysis component when we adapted RV to the context of software evolution by focusing RV and its users on affected parts of code [116] (Chapter 5). We compared the class-level RTS in STARTS with the module-level RTS commonly used in large software ecosystems, e.g., at Facebook, Google and Microsoft [76]. We found that STARTS can reduce the test-selection rate in those ecosystems by 10x. Zhu et al. [226] used STARTS as part of the RTSCheck framework for checking RTS tools. Finally, we recently published our work on making static RTS safer by making STARTS reflection aware [183].

Several researchers recently published Master's theses and a doctoral dissertation where STARTS played a role [1,78,103,130,214]. Hadzi-Tanovic [78] published our initial results on reflection-aware static RTS. Karlsson [103] evaluated the safety, performance, and precision impact on static RTS of limiting the transitive closure computation on the IRG to fixed depths, instead of using the full transitive closure. Lundsten [130] described a preliminary comparison of STARTS with a machine-learning based RTS approach inspired by Machalica et al. [133]. Yilmaz [214] compared STARTS with an information-retrieval based RTS approach. Lastly, Al-Refai [1] compared RTS based on UML class diagrams with the RTS performed by STARTS.

6.5 RUNTIME VERIFICATION

Many RV techniques and tools were proposed in almost two decades since the first papers on what is now called RV [55, 83, 84], mostly concerned with speeding up RV on *one program version*. Example techniques (1) improve the efficiency of synthesizing monitors [85], (2) improve the efficiency of monitor garbage collection [98, 100, 132], (3) create a virtual machine to make RV more efficient in production [9], (4) reduce RV overhead by sharing information among monitors [45, 132, 163], (5) support efficient monitoring of properties written in different formalism [13, 82, 138, 139], (6) analyze observed executions in monitors to infer characteristics of unseen executions [28, 29], (7) allow RV to monitor multiple properties in one execution [100, 132], (8) reduce the time that RV wastes in loops [162], etc. Tools include Eagle [12], JavaMOP [92, 99, 132], jMonitor [102], JPaX [83], MarQ [166], MOP-Box [26], Mufin [45], Ruler [13], and TraceMatches [2, 27]. We used JavaMOP because it is publicly available [92] and developed in the same group/department. The interested reader may find overviews of the RV research landscape elsewhere, including a comparison with other quality assurance techniques [124], an overview [16], a tutorial [56] and a taxonomy of RV tools [57]. Further, the international competition on runtime verification has been held since 2014 to provide community impetus for improving existing RV tools and validating new ones [14, 15, 17, 58, 167].

Javed and Binder [94] recently conducted a study of multiple RV tools when monitoring test executions in many open-source projects. Specifically, they use JavaMOP, MarQ, and Mufin to monitor two manually written properties while running 286,638 tests in 1,777 open-source projects. They measure the runtime overhead and memory consumption of these tools but did not evaluate the bug-finding effectiveness of the properties on these projects. Further, they monitor these two properties one at a time, not simultaneously as we do for 199 properties in this dissertation. We are encouraged to see that, following our large-scale study of performing RV while running developer written and automatically generated tests in open-source projects [118], other researchers started conducting large-scale evaluations of other RV tools apart from JavaMOP.

The work presented in this dissertation is a culmination of research towards realizing the idea of evolution-aware RV which we proposed [115]. We initially proposed three techniques to achieve evolution-aware RV, all of which depended on efficient and effective change-impact analysis. Therefore, we implemented and evaluated STARTS, a static class-level change-impact analysis and RTS tool (chapters 3 and 4). The first technique that we proposed was Regression Property Selection (RPS), which we have now realized (Chapter 5). The second technique that we proposed was Regression Monitor Selection (RMS), which prevents

monitors for selected properties from being created in parts of code that are not affected by the code changes. The instrumentation reduction approaches in Section 5.2.3 already achieve this goal. We leave as future work to compare these instrumentation reduction approaches with RMS as originally proposed. The third technique that we proposed was to simply combine RTS with RV, which we evaluated in Section 5.4.3—this combination alone did not achieve much overhead reduction compared with performing RV from scratch after every code change. Regression Property Prioritization (RPP) and Violation Message Suppression (VMS), were not in our original evolution-aware RV proposal [115] but were inspired by our experience while using RV on multiple program versions.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

We presented the first results on using Runtime Verification (RV) to find more bugs from tests that developers already have. Although the potential for RV to be used during software testing was previously mentioned, there was no prior investigation of the effectiveness and efficiency of using RV during software testing as an early-stage bug-finding approach. Further, all prior research on RV considered only one program version, although rapid change and continuous integration are hallmarks of today's software development. Thus, using existing RV techniques during testing of rapidly-evolving software would needlessly, repeatedly, and wastefully incur overhead in re-monitoring parts of code that are not affected by code changes. The contributions of this dissertation address these problems.

First, we conducted a large-scale study of the efficiency and effectiveness of performing RV during software testing. The results of the study show that RV helped find many bugs that existing tests alone could not catch and developers accepted most of the pull requests that we submitted to fix such bugs. However, the results also show that RV incurs high overhead, both in terms of execution time and the time that developers spend waiting for and inspecting violations. Second, we proposed the *idea* of evolution-aware RV as a way to reduce the overhead incurred by RV across multiple program versions, by re-monitoring only the parts of code affected by code changes. Third, since our proposed ideas for evolution-aware RV depend critically on fast change-impact analysis, we implemented class- and method-level static change-impact analyses. Our evaluation of these change-impact analyses in the context of RTS showed that class-level static RTS both outperformed method-level static RTS and performed similarly as the state-of-the-art dynamic RTS technique. Fourth, we implemented *techniques* for evolution-aware RV that amortize the runtime overhead of RV across multiple program versions and show developers only new violations that occur after code changes. Our evolution-aware RV techniques reduced accumulated runtime overhead of RV by up to 10× and showed two orders of magnitude fewer violations, without missing any new violation in the projects and versions that we evaluated.

The results in this dissertation provide compelling evidence that taking software evolution into account is very effective for reducing the overhead of RV across multiple program versions. The following are some avenues for future work, building on the work presented in this dissertation:

Better Properties: We provided a set of recommendations for future work on obtaining properties that are more effective at finding true bugs when performing RV during software testing (Section 2.5.1). It is essential for the RV community to come up new approaches

for repairing existing properties and for mining more effective properties. The adoption of RV in practice hinges on the quality of the properties.

Library-Level Properties vs. Project-Specific Properties: We plan to evaluate whether project-specific properties that are manually written from a project's API documentation or automatically mined from a project have a lower false alarm rate (FAR), compared to the standard library API-level properties that we used in this dissertation.

Mixed-Granularity Change-Impact Analysis: To further speed up our evolution-aware RV techniques, we plan to improve the precision of change-impact analysis. One idea is to perform a method-level analysis on top of the current class-level analysis, as Zhang recently did for RTS in HyRTS [218].

Static RTS: Open research directions that we are interested to tackle are how to make static RTS more precise, how to make static RTS safer with respect to other potential sources of unsafety (such as dependency of tests on external files or native code), and how to apply static RTS to other programming languages. Future plans for STARTS include (1) developing faster checksum and dependency storage formats; (2) supporting other build systems, such as Bazel or Gradle; (3) making STARTS usable in continuous-integration systems, such as Jenkins or Travis; (4) evaluating STARTS on larger projects than those we evaluated; (5) creating a STARTS IDE plugin to promote greater community adoption; and (6) improving the scalability of STARTS in ultra-large software ecosystems [76].

A Tool for Evolution-Aware RV: We plan to develop a robust open-source evolution-aware RV tool that others can use in their own software development and research. We already released STARTS as part of our dissertation work [191]. STARTS is already being used in other research [1, 39, 76, 78, 103, 130, 183, 214, 226].

Further Analysis of Evolution-Aware RV Results: One surprising result from our evaluation of evolution-aware RV techniques is that weak RPS techniques that we designed to trade off some safety for better efficiency were safe in our experiments. We plan to experiment with more projects and longer version histories to better empirically characterize the safety of the weak RPS techniques in practice.

Scaling Evolution-Aware RV to Larger Software Systems: As the first line of work on evolution-aware RV, we have so far studied RV with, and evaluated our evolution-aware RV techniques on, medium-sized projects. We plan to investigate how to scale RV, and our change-impact analysis, RTS, and evolution-aware RV techniques to much larger projects, in terms of lines of code, test-running time, and RV overhead.

Evolution-Aware RV in Developer Settings: We plan to adapt and deploy evolution-aware RV to real continuous integration (CI) pipelines of real-world projects. We also plan to integrate evolution-aware RV into IDEs, for use as developers write their code. These

will enable studies of evolution-aware RV technique usage by developers. We previously studied how developers perform RTS inside IDEs [71] and that experience can help us in studying IDE-based evolution-aware RV as well.

Evolution-Awareness for other Formal Methods: Evolution-aware RV merely scratches the surface when it comes to making dynamic-analysis based formal methods more usable during regression testing. We plan to investigate principles and techniques for evolution awareness in other formal methods apart from RV.

Given the large number of bugs that RV helped find during our research, the significant reduction in RV overhead achieved by evolution-aware RV, and the plans that we have for further improvements, we expect our research could lead to a new era where lightweight formal methods like runtime verification are routinely used by developers. We believe that using RV during software testing can greatly improve software quality, that evolution-aware RV opens a new direction for integrating RV into everyday software testing, and that it is time for research to start paying more attention to development-time RV, in addition to deployment-time RV which was previously well studied.

REFERENCES

[1] M. N. Al-Refai, "Towards model-based regression test selection," Ph.D. dissertation, Colorado State University, USA, 2019.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005, pp. 345–364.

[3] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for JavaScript," *ACM Computing Surveys*, vol. 50, no. 5, pp. 66:1–66:36, Sep. 2017.

[4] "Apache Camel," http://camel.apache.org/.

[5] "Apache Commons Math," https://commons.apache.org/proper/commons-math/.

[6] "Apache CXF," https://cxf.apache.org/.

[7] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *International Conference on Software Engineering*, 2005, pp. 432–441.

[8] T. Armerding, "Hard questions raised when a software 'glitch' takes down an airliner," *Forbes*. [Online]. Available: https://www.forbes.com/sites/taylorarmerding/2018/11/20/hard-questions-raised-when-a-software-glitch-takes-down-an-airliner/#73345ac67b1d

[9] M. Arnold, M. Vechev, and E. Yahav, "QVM: An efficient runtime for detecting defects in deployed systems," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008, pp. 143–162.

[10] "ASM," http://asm.ow2.org/.

[11] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: A control call graph based technique," in *Asia-Pacific Software Engineering Conference*, 2005, pp. 167–175.

[12] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *Verification, Model Checking, and Abstract Interpretation*, 2004, pp. 44–57.

[13] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: From Eagle to RuleR," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 675–706, 2010.

[14] E. Bartocci, B. Bonakdarpour, and Y. Falcone, "First international competition on software for runtime verification," in *International Conference on Runtime Verification*, 2014, pp. 1–9.

[15] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Roşu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang, "First international competition on runtime verification: Rules, benchmarks, tools, and final results of CRV 2014," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 31–70, 2019.

[16] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," in *Lectures on Runtime Verification: Introductory and Advanced Topics*, 2018, pp. 1–33.

[17] E. Bartocci, Y. Falcone, and G. Reger, "International competition on runtime verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2019, pp. 41–49.

[18] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Symposium on Principles of Programming Languages*, 1993, pp. 384–396.

[19] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *Conference on Programming Language Design and Implementation*, 2011, pp. 211–221.

[20] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *International Conference on Software Engineering*, 2018, pp. 433–444.

[21] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *International Conference on Software Engineering*, 2014, pp. 550–561.

[22] J. Bell, G. E. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 770–781.

[23] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 498–516, 1997.

[24] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica*, vol. 35, no. 3, pp. 289–321, 2011.

[25] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 169–190.

[26] E. Bodden, "MOPBox: A library approach to runtime verification," in *International Conference on Runtime Verification (Tools Demonstration Track)*, 2011, pp. 365–369.

[27] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with Tracematches," in *International Conference on Runtime Verification*, 2007, pp. 22–37.

[28] E. Bodden, L. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," in *European Conference on Object-Oriented Programming*, 2007, pp. 525–549.

[29] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *International Symposium on Foundations of Software Engineering*, 2008, pp. 36–47.

[30] S. Bohner, "Impact analysis in the software change process: A year 2000 perspective," in *International Conference on Software Maintenance*, 1996, pp. 42–51.

[31] S. A. Bohner and R. S. Arnold, *Software change impact analysis*. IEEE Computer Society Press, Los Alamitos, 1996, vol. 6.

[32] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *International Symposium on Software Testing and Analysis*, 2002, pp. 123–133.

[33] L. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *Journal of Information and Software Technology*, vol. 51, no. 1, pp. 16–30, 2009.

[34] H. Cai, S. Jiang, R. Santelices, Y. Zhang, and Y. Zhang, "SENSA: Sensitivity analysis for quantitative change-impact prediction," in *International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 165–174.

[35] H. Cai, "Hybrid program dependence approximation for effective dynamic impact prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 334–364, 2018.

[36] H. Cai and R. Santelices, "A comprehensive study of the predictive accuracy of dynamic change-impact analysis," *Journal of Systems and Software*, vol. 103, pp. 248–265, 2015.

[37] D. Chen, Y. Zhang, R. Wang, X. Li, L. Peng, and W. Wei, "Mining universal specification based on probabilistic model," in *International Conference on Software Engineering and Knowledge Engineering*, 2015, pp. 471–476.

[38] F. Chen and G. Roşu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," in *International Conference on Runtime Verification*, 2003, pp. 108–127.

[39] L. Chen and L. Zhang, "Speeding up mutation testing via regression test selection: An extensive study," in *International Conference on Software Testing, Verification, and Validation*, 2018, pp. 58–69.

[40] W. G. Cochran, *Sampling techniques.* John Wiley & Sons, 1977.

[41] "Collections," https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedCollection-java.util.Collection-.

[42] "Collections_SynchronizedCollection," http://fsl.cs.illinois.edu/annotated-java/__properties/html/java/util/Collections_SynchronizedCollection.html.

[43] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *International Symposium on Software Testing and Analysis*, 2010, pp. 85–96.

[44] "CompleteSearch DBLP," http://www.dblp.org/search/index.php.

[45] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime monitoring with union-find structures," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2016, pp. 868–884.

[46] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *International Conference on Software Testing, Verification, and Validation*, 2013, pp. 302–311.

[47] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[48] M. B. Dwyer, R. Purandare, and S. Person, "Runtime verification in context: Can optimizing error detection improve fault diagnosis?" in *International Conference on Runtime Verification*, 2010, pp. 36–50.

[49] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[50] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[51] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Product-Focused Software Process Improvement*, 2010, pp. 3–16.

[52] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Journal of Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[53] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: A systematic review," in *International Symposium on Empirical Software Engineering and Measurement*, 2008, pp. 22–31.

[54] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: An industrial evaluation," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 335–353, 2016.

[55] U. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE S&P*, 2000, pp. 246–255.

[56] Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification," in *Engineering Dependable Software Systems*, 2013, pp. 141–175.

[57] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, "A taxonomy for classifying runtime verification tools," in *International Conference on Runtime Verification*, 2018, pp. 241–262.

[58] Y. Falcone, D. Ničković, G. Reger, and D. Thoma, "Second international competition on runtime verification," in *International Conference on Runtime Verification*, 2015, pp. 405–422.

[59] M. Fowler. (2006) Continuous Integration. [Online]. Available: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf

[60] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2011, pp. 416–419.

[61] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *International Symposium on Software Testing and Analysis*, 2013, pp. 291–301.

[62] ——, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Transactions on Software Engineering Methodology*, vol. 24, no. 4, pp. 23:1–23:49, 2015.

[63] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *International Conference on Software Engineering*, 2010, pp. 15–24.

[64] ——, "Testing mined specifications," in *International Symposium on Foundations of Software Engineering*, 2012, pp. 1–11.

[65] M. Gethers, H. Kagdi, B. Dit, and D. Poshyvanyk, "An adaptive approach to impact analysis from change requests to source code," in *International Conference on Automated Software Engineering*, 2011, pp. 540–543.

[66] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *International Conference on Software Engineering*, 2012, pp. 430–440.

[67] M. Gligoric, "Regression test selection: Theory and practice," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 2015.

[68] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2015, pp. 713–716.

[69] ——, "Practical regression test selection with dynamic file dependencies," in *International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[70] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *International Conference on Software Engineering*, 2010, pp. 225–234.

[71] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, "An empirical evaluation and comparison of manual and automated test selection," in *International Conference on Automated Software Engineering*, 2014, pp. 361–372.

[72] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *International Conference on Software Engineering*, 1998.

[73] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 6, pp. 685–746, 2001.

[74] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of Google," http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[75] A. Gyori, S. K. Lahiri, and N. Partush, "Refining interprocedural change-impact analysis using equivalence relations," in *International Symposium on Software Testing and Analysis*, 2017, pp. 318–328.

[76] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *International Symposium on Software Reliability Engineering*, 2018, pp. 112–122.

[77] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *International Symposium on Software Testing and Analysis*, 2015, pp. 223–233.

[78] M. Hadzi-Tanovic, "Reflection-aware static regression test selection," Master's thesis, University of Illinois at Urbana-Champaign, USA, 2018.

[79] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering Methodology*, vol. 24, no. 2, pp. 10:1–10:31, 2014.

[80] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *International Conference on Software Engineering*, 2012, pp. 738–748.

[81] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 312–326.

[82] K. Havelund, D. Peled, and D. Ulus, "First order temporal logic monitoring with BDDs," in *Formal Methods in Computer Aided Design*, 2017, pp. 116–123.

[83] K. Havelund and G. Roşu, "Monitoring Java programs with Java PathExplorer," in *International Conference on Runtime Verification*, 2001, pp. 200–217.

[84] ——, "Monitoring programs using rewriting," in *International Conference on Automated Software Engineering*, 2001, pp. 135–143.

[85] ——, "Synthesizing monitors for safety properties," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2002, pp. 342–356.

[86] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *International Conference on Software Engineering*, 2015, pp. 483–493.

[87] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 197–207.

[88] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *International Conference on Automated Software Engineering*, 2016, pp. 426–437.

[89] S. Hussein, P. Meredith, and G. Roşu, "Security-policy monitoring and enforcement with JavaMOP," in *Workshop on Programming Languages and Analysis for Security*, 2012, pp. 1–11.

[90] "JavaMOP4 Syntax," http://fsl.cs.illinois.edu/index.php/JavaMOP4_Syntax.

[91] "JavaMOPAgent Documentation," https://github.com/runtimeverification/javamop/blob/master/docs/JavaMOPAgentUsage.md.

[92] "JavaMOP," http://fsl.cs.illinois.edu/index.php/JavaMOP.

[93] "java.util.Collections," https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html.

[94] O. Javed and W. Binder, "Large-scale evaluation of the efficiency of runtime-verification tools in the wild," in *Asia-Pacific Software Engineering Conference*, 2018, pp. 688–692.

[95] "jdeps," https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html.

[96] "jEdit JDiff Plugin," http://plugins.jedit.org/plugins/?JDiffPlugin.

[97] "JGraphT," http://jgrapht.org/.

[98] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *Conference on Programming Language Design and Implementation*, 2011, pp. 415–424.

[99] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2012, pp. 1427–1430.

[100] D. Jin, P. O. Meredith, and G. Roşu, "Scalable parametric runtime monitoring," Computer Science Dept., UIUC, Tech. Rep., 2012.

[101] "Joda-Time," http://www.joda.org/joda-time/.

[102] M. Karaorman and J. Freeman, "jMonitor: Java runtime event specification and monitoring library," in *International Conference on Runtime Verification*, 2004, pp. 181–200.

[103] H. Karlsson, "Limiting transitive closure for static regression test selection approaches," Master's thesis, KTH Royal Institute of Technology, Sweden, 2019.

[104] A. Keller, H. Schippers, and S. Demeyer, "Supporting inconsistency resolution through predictive change impact analysis," in *International Workshop on Model-Driven Engineering, Verification and Validation*, 2009, pp. 1–10.

[105] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 178–189.

[106] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change impact identification in object oriented software maintenance," in *International Conference on Software Maintenance*, 1994, pp. 202–211.

[107] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8, no. 2, pp. 51–65, 1995.

[108] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *International Conference on Software Testing, Verification, and Validation*, 2019, pp. 312–322.

[109] J. Law and G. Rothermel, "Incremental dynamic impact analysis for evolving software systems," in *International Symposium on Software Reliability Engineering*, 2003, pp. 430–441.

[110] ——, "Whole program path-based dynamic impact analysis," in *International Conference on Software Engineering*, 2003, pp. 308–318.

[111] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 292–306.

[112] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *International Conference on Software Engineering*, 2011, pp. 591–600.

[113] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.

[114] M. Lee, "Change impact analysis of object-oriented software," Ph.D. dissertation, George Mason University, USA, 1998.

[115] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *International Conference on Software Engineering (New Ideas and Emerging Results Track)*, 2015, pp. 615–618.

[116] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *International Conference on Software Testing, Verification, and Validation*, 2019, pp. 300–311.

[117] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[118] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications," in *International Conference on Automated Software Engineering*, 2016, pp. 602–613.

[119] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STAtic Regression Test Selection," in *International Conference on Automated Software Engineering*, 2017, pp. 949–954.

[120] S. Lehnert, "A review of software change impact analysis," Ilmenau U. of Tech., Tech. Rep., 2011.

[121] ——, "A taxonomy for software change impact analysis," in *International Workshop on Principles of Software Evolution*, 2011, pp. 41–50.

[122] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining," in *International Conference on Automated Software Engineering*, 2015, pp. 81–92.

[123] C. Lemieux, "Mining temporal properties of data invariants," in *International Conference on Software Engineering (Student Research Competition Track)*, 2015, pp. 751–753.

[124] M. Leucker and C. Schallhart, "A brief account of runtime verification," in *Workshop on Formal Languages and Analysis of Contract-Oriented Software*, 2007, pp. 293–303.

[125] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *International Conference on Software Maintenance*, 1990, pp. 290–301.

[126] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Journal of Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[127] M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" *Journal of Systems and Software*, vol. 43, no. 1, pp. 19–27, 1998.

[128] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *Conference on Programming Language Design and Implementation*, 2014, pp. 294–304.

[129] "LTL Plugin4 Input Syntax," http://fsl.cs.illinois.edu/index.php/LTL_Plugin4_Input_Syntax.

[130] E. Lundsten, "EALRTS: A predictive regression test selection tool," Master's thesis, KTH Royal Institute of Technology, Sweden, 2019.

[131] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[132] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Roşu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *International Conference on Runtime Verification*, 2014, pp. 285–300.

[133] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *International Conference on Software Engineering, Software Engineering in Practice*, 2019, pp. 91–100.

[134] S. Matteson, "Report: Software failure caused $1.7 trillion in financial losses in 2017," *TechRepublic*. [Online]. Available: https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/

[135] "Maven is broken by design," https://blog.ltgt.net/maven-is-broken-by-design/.

[136] "Guide to naming conventions," https://maven.apache.org/guides/mini/guide-naming-conventions.html.

[137] "Introduction to Maven 2.0 Plugin Development," https://maven.apache.org/guides/introduction/introduction-to-plugins.html.

[138] P. Meredith and G. Roşu, "Efficient parametric runtime verification with deterministic string rewriting," in *International Conference on Automated Software Engineering*, 2013, pp. 70–80.

[139] P. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," in *International Conference on Automated Software Engineering*, 2008, pp. 148–157.

[140] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[141] J. Murdock, "Google self-driving car caused freeway crash after engineer modified its software," *Newsweek*. [Online]. Available: https://www.newsweek.com/former-google-engineer-allegedly-modified-self-driving-software-caused-crash-1174058

[142] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[143] A. C. Nguyen and S.-C. Khoo, "Extracting significant specifications from mining through mutation testing," in *International Conference on Formal Engineering Methods*, 2011, pp. 472–488.

[144] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 166–177.

[145] "java.lang.instrument," http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html.

[146] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Symposium on Logic in Computer Science*, 2018, pp. 13–25.

[147] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *International Conference on Software Engineering*, 2004, pp. 491–500.

[148] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *International Symposium on Foundations of Software Engineering*, 2003, pp. 128–137.

[149] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *International Symposium on Foundations of Software Engineering*, 2004, pp. 241–251.

[150] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2007, pp. 815–816.

[151] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .NET with feedback-directed random testing," in *International Symposium on Software Testing and Analysis*, 2008, pp. 87–96.

[152] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[153] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *International Conference on Software Maintenance and Evolution*, 2017, pp. 1–12.

[154] W. E. Perry, *Effective methods for software testing: Includes complete guidelines, checklists, and templates*. John Wiley & Sons, 2007.

[155] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *International Conference on Software Maintenance*, 2010, pp. 1–10.

[156] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *International Conference on Automated Software Engineering*, 2009, pp. 371–382.

[157] ——, "Leveraging test generation and specification mining for automated bug detection without false positives," in *International Conference on Software Engineering*, 2012, pp. 288–298.

[158] M. Pradel, "Dynamically inferring, refining, and checking API usage protocols," in *Companion Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009, pp. 773–774.

[159] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *International Conference on Software Engineering*, 2012, pp. 925–935.

[160] "Statically checking API protocol conformance with mined multi-object specifications (supplementary material)," http://mp.binaervarianz.de/icse2012-statically/.

[161] "FSL Specification Database," https://runtimeverification.com/monitor/propertydb.

[162] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2010, pp. 270–285.

[163] ——, "Optimizing monitoring of finite state properties through monitor compaction," in *International Symposium on Software Testing and Analysis*, 2013, pp. 280–290.

[164] "Randoop," https://randoop.github.io/randoop/.

[165] G. Reger, H. Barringer, and D. Rydeheard, "A pattern-based approach to parametric specification mining," in *International Conference on Automated Software Engineering*, 2013, pp. 658–663.

[166] G. Reger, H. C. Cruz, and D. Rydeheard, "MarQ: Monitoring at runtime with QEA," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 596–610.

[167] G. Reger, S. Hallé, and Y. Falcone, "Third international competition on runtime verification," in *International Conference on Runtime Verification*, 2016, pp. 21–37.

[168] S. P. Reiss, "Tracking source locations," in *International Conference on Software Engineering*, 2008, pp. 11–20.

[169] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of Java programs," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004, pp. 432–448.

[170] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby, "Chianti: A prototype change impact analysis tool for Java," Rutgers University CS Dept., Tech. Rep. DCS-TR-533, 2003.

[171] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[172] R. H. Rosero, O. S. Gómez, and G. Rodríguez, "15 years of software regression testing techniques—A survey," *International Conference on Software Engineering and Knowledge Engineering*, vol. 26, no. 5, pp. 675–689, 2016.

[173] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *International Conference on Software Maintenance*, 1993, pp. 358–367.

[174] ——, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[175] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *International Conference on Software Maintenance*, 1999, pp. 179–189.

[176] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 46–53.

[177] R. Santelices and M. J. Harrold, "Probabilistic slicing for predictive impact analysis," Georgia Institute of Technology, Tech. Rep., 2010.

[178] S. Shamshiri, R. Just, J. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges," in *International Conference on Automated Software Engineering*, 2015, pp. 201–211.

[179] M. Sherriff and L. Williams, "Empirical software change impact analysis using singular value decomposition," in *International Conference on Software Testing, Verification, and Validation*, 2008, pp. 268–277.

[180] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 246–256.

[181] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *International Conference on Software Testing, Verification, and Validation*, 2016, pp. 80–90.

[182] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real software evolution," in *International Symposium on Software Testing and Analysis*, 2018, pp. 84–94.

[183] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2019, pp. 187:1–187:29.

[184] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *International Symposium on Foundations of Software Engineering*, 2019, pp. 545–555.

[185] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 237–247.

[186] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

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

[188] ——, "Improving class firewall regression test selection by removing the class firewall," *International Journal on Software Engineering and Knowledge Engineering*, vol. 17, no. 3, pp. 359–378, 2007.

[189] "Supplementary material for paper [118]," http://fsl.cs.illinois.edu/spec-eval.

[190] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *International Symposium on Software Testing and Analysis*, vol. 27, no. 4, 2002, pp. 97–106.

[191] "STARTS—A tool for STAtic Regression Test Selection," https://github.com/TestingResearchIllinois/starts.

[192] S. Stolberg, "Enabling agile testing through continuous integration," in *Agile Conference*, 2009, pp. 369–374.

[193] "StringTokenizer_HasMoreElements," http://fsl.cs.illinois.edu/annotated-java/__properties/html/java/util/StringTokenizer_HasMoreElements.html.

[194] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for Alloy," in *International Conference on Software Testing, Verification, and Validation*, 2017, pp. 264–275.

[195] J. Sun, H. Xiao, Y. Liu, S.-W. Lin, and S. Qin, "TLV: Abstraction through testing, learning, and validation," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 698–709.

[196] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *International Conference on Software Testing, Verification, and Validation*, 2012, pp. 260–269.

[197] G. L. Thione, "Detecting semantic conflicts in parallel changes," Master's thesis, University of Texas at Austin, USA, 2002.

[198] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *International Conference on Automated Software Engineering*, 2009, pp. 283–294.

[199] N. Tillmann and J. de Halleux, "Pex—White box test generation for .NET," in *Tests and Proofs*, 2008, pp. 134–153.

[200] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 281–293.

[201] "Travis-CI," https://travis-ci.org/.

[202] "URLDecoder_DecodeUTF8," https://runtimeverification.com/monitor/annotated-java/__properties/html/java/net/URLDecoder_DecodeUTF8.html.

[203] "W3recommendation," https://www.w3.org/TR/html40/appendix/notes.html#non-ascii-chars.

[204] J. Wakefield, "Nest thermostat bug leaves users cold," *BBC*. [Online]. Available: https://www.bbc.com/news/technology-35311447

[205] "IBM WALA," http://wala.sourceforge.net.

[206] M. L. Wald, "Airline blames bad software in San Francisco crash," *The New York Times*. [Online]. Available: https://www.nytimes.com/2014/04/01/us/asiana-airlines-says-secondary-cause-of-san-francisco-crash-was-bad-software.html

[207] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *International Conference on Automated Software Engineering*, 2009, pp. 295–306.

[208] W. Weimer and G. Necula, "Mining temporal specifications for error detection," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 461–476.

[209] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Iterative mining of resource-releasing specifications," in *International Conference on Automated Software Engineering*, 2011, pp. 233–242.

[210] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.

[211] G. Xu and A. Rountev, "Regression test selection for AspectJ software," in *International Conference on Software Engineering*, 2007, pp. 65–74.

[212] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *International Conference on Software Engineering*, 2006, pp. 282–291.

[213] "Yet Another Simple Graph Library," https://github.com/TestingResearchIllinois/yasgl.

[214] U. Yilmaz, "A method for selecting regression test cases based on software changes and software faults," Master's thesis, Hacettepe University, Turkey, 2019.

[215] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[216] N. York, "Tools for continuous integration at Google scale," Jan. 2011, https://www.youtube.com/watch?v=b52aXZ2yi08.

[217] K. Zhai, B. Jiang, and W. K. Chan, "Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study," *IEEE Transactions on Services Computing*, vol. 7, no. 1, pp. 54–67, 2014.

[218] L. Zhang, "Hybrid regression test selection," in *International Conference on Software Engineering*, 2018, pp. 199–209.

[219] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *International Conference on Software Engineering*, 2013, pp. 192–201.

[220] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *International Conference on Software Maintenance*, 2011, pp. 23–32.

[221] ——, "FaultTracer: A change impact and regression fault analysis tool for evolving Java programs," in *International Symposium on Foundations of Software Engineering (Tool Demonstrations Track)*, 2012, pp. 1–4.

[222] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of JUnit test-suite reduction," in *International Symposium on Software Reliability Engineering*, 2011, pp. 170–179.

[223] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *International Symposium on Software Testing and Analysis*, 2014, pp. 385–396.

[224] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *International Conference on Automated Software Engineering*, 2009, pp. 307–318.

[225] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Journal of Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[226] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *International Conference on Software Engineering*, 2019, pp. 430–441.