An Empirical Evaluation and Comparison of Manual and Automated Test Selection

Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov
University of Illinois at Urbana-Champaign
{gliga,snegara2,legunse2,marinov}@illinois.edu

ABSTRACT

Regression test selection speeds up regression testing by re-running only the tests that can be affected by the most recent code changes. Much progress has been made on research in automated test selection over the last three decades, but it has not translated into practical tools that are widely adopted. Therefore, developers either *re-run all tests* after each change or perform *manual test selection*. Re-running all tests is expensive, while manual test selection is tedious and error-prone. Despite such a big trade-off, no study assessed how developers perform manual test selection and compared it to automated test selection.

This paper reports on our study of manual test selection in practice and our comparison of manual and automated test selection. We are the first to conduct a study that (1) analyzes data from manual test selection, collected in real time from 14 developers during a three-month study and (2) compares manual test selection with an automated state-of-the-research test-selection tool for 450 test sessions.

Almost all developers in our study performed manual test selection, and they did so in mostly ad-hoc ways. Comparing manual and automated test selection, we found the two approaches to select different tests in *each and every* one of the 450 test sessions investigated. Manual selection chose more tests than automated selection 73% of the time (potentially wasting time) and chose fewer tests 27% of the time (potentially missing bugs). These results show the need for better automated test-selection techniques that integrate well with developers' programming environments.

1. INTRODUCTION

Regression testing [8, 42, 43] is an important activity in software development. It checks that software changes do not break existing tests. To quickly detect changes that break tests, it is desirable to run tests frequently. However, regression testing is also expensive. Because software projects can have very many tests, re-running all the tests after every code change is often time-consuming [37, 38].

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2643019.

Regression test selection (RTS) aims to improve the efficiency of regression testing by selecting to re-run only a subset of the test suite, namely, those tests that can be *affected* [11, 12, 43] by the changes. An RTS technique is *safe* if it guarantees to select all affected tests [31, 40], i.e., the behavior of unselected tests after the code changes remains the same as before the code changes. One RTS technique is more *precise* than another RTS technique if it selects fewer (non-affected) tests.

While researchers have proposed many automated test-selection techniques over the last three decades [1, 3, 5, 7, 14, 15, 31, 40, 42], there is still no tool that implements these techniques in a way that is practical enough for widespread adoption. The few successful systems for automated regression testing, e.g., the TAP system at Google [37, 38] and the Scout system (formerly known as Echelon) at Microsoft [16, 17, 34], use techniques that are either imprecise (e.g., TAP selects, for each project, either all tests or no tests based on compile-time project dependencies) or unsafe (e.g., Scout does not guarantee that the behavior of unselected tests will remain the same, so Scout only prioritizes [6, 32] but does not select tests). These systems can work well for very large codebases, but most developers do not write code at the scale of Google or Microsoft. Rather, most developers work on smaller projects, for which running a coarse-grained technique like TAP would often be equivalent to re-running all tests. Such smaller projects would require a finer-grained technique for more precise RTS (e.g., finding affected tests based on control-flow edges or methods [2, 14, 29]).

The lack of practical RTS tools leaves two options for developers: either automatically re-run all the tests or manually perform test selection. Re-running all the tests is safe by definition, but it can be quite imprecise and, therefore, inefficient. In contrast, manual test selection, which we will refer to as *manual RTS*, can be both unsafe and imprecise: developers can select too few tests and thus miss to run some tests whose behavior differs due to code changes, or developers can select too many tests and thus waste time.

Despite the importance of RTS, we are not aware of any research that studies *if* and *how* developers perform manual RTS, and how manual and automated RTS compare. Our anecdotal experience shows that developers select to run only some of their tests, but we do not know how many developers do so, how many tests they select, why they select those tests, what automated support they use for manual RTS in their Integrated Development Environment (IDE), etc. Also, it is unknown how developers' manual RTS practices compare with any automated RTS technique proposed in the literature: how does developers' reasoning about affected tests compare to the analysis of a safe and precise automated RTS technique? The need for adoptable automated RTS tools makes it critical to study current manual RTS practice and its effects.

This paper presents the results of the first study of manual RTS and a first comparison of manual and automated RTS. Specifically, we address the following research questions:

RQ1. How often do developers perform manual RTS?

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

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

RQ4. How do developers commonly perform manual RTS?

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

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

To address the first set of questions about manual RTS (RQ1-RQ5), we extensively analyzed logs of IDE interactions recorded from a diverse group of 14 developers (working on 17 projects, i.e., some developers worked on multiple projects during our study), including several experts from industry [23]. These logs cover a total of 918 hours of development, with 5,757 test sessions and a total of 264,562 executed tests. A *test session* refers to a run of at least one test between two sets of code changes. We refer to test sessions with a single test as *single-test sessions*, and test sessions with more than one test as *multiple-test sessions*. To address RQ6, we compared the safety, precision, and performance of manual and automated RTS for 450 test sessions of one representative project, using the best available automated RTS research prototype [44].

Several of our findings are surprising. Regardless of the project properties (small vs. large, few tests vs. many tests, etc.), *almost all developers* performed manual RTS. 62% of all test sessions executed a single test, and of multiple-test sessions, on average, 59% had some test selection. The pervasiveness of manual RTS establishes the need to study manual RTS in more depth and compare it with automated RTS. Moreover, our comparison of manual and automated RTS revealed that manual RTS can be imprecise (in 73% of the test sessions, manual RTS selects more tests than automated RTS) and unsafe (in 27% of the test sessions, manual RTS selects fewer tests than automated RTS). Finally, our experiments show that current automated RTS may provide little time savings: the time taken by an automated RTS tool[1], per session, to select tests was 130.94±13.77 sec (Mean±SD) and the (estimated) time saved (by not executing unselected tests) was 219.86±68.88 sec. These results show a strong need for better automated RTS tools.

This paper makes the following contributions:

⋆ **Evidence.** We conducted the first study to show how developers perform manual RTS.

⋆ **Examination.** We closely examined manual RTS and its associated factors in practice.

⋆ **Comparison.** We performed the first comparison of manual and automated RTS.

2. EVALUATING MANUAL RTS

We present our methodology for analyzing manual RTS data to answer RQ1-RQ5, and summarize our findings.

2.1 Methodology

We analyzed the data collected during our previous field study [23], in which we unobtrusively monitored developers' IDEs and recorded their programming activities over three months. We had used the collected data in our prior research studies [21, 23, 39] on refactoring and version control; the work presented in this paper is the first to focus on the (regression) testing aspects.

To collect data, we asked our study participants to install our record-and-replay tool, `CodingTracker` [4], in their Eclipse (Indigo) IDEs. Throughout the study, `CodingTracker` recorded detailed code evolution data, ranging from individual code edits, start of each test, and test outcome (e.g., pass/fail) up to high-level events like automated refactoring invocations and test session executions. `CodingTracker` uploaded the collected data to our centralized repository using existing infrastructure [39].

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

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

2.1.1 *CodingTracker*

`CodingTracker` integrates well with one of the most popular IDEs, Eclipse [19]. Developers do not explicitly interact with `CodingTracker` during their workflow, and thus, the data recorded by `CodingTracker` is as close as possible to what developers normally do. `CodingTracker` collects information about all test sessions. Because test-selection data is available at every test session, we were able to capture developers' manual RTS decisions. Each test session includes a list of executed tests, their execution time, and their status on completion (pass or fail). Further, `CodingTracker` collects information about *code changes* between test sessions.

While `CodingTracker` logs provide a treasure trove of data, they have limitations. First, `CodingTracker` logs cannot fully confirm that developers performed manual RTS. In theory,

[1]This measures only the analysis time to identify the affected tests but *not* the time to collect coverage.

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

Figure 1: Statistics for projects used in the study; "Selective Sessions" is of multiple-test sessions; we exclude single-test sessions as they may not be "true" Selective Sessions - developer knows that not all affected tests are selected

developers could have installed some Eclipse plugin that would perform automated RTS for them. However, we are not aware of any automated RTS tool that works in Eclipse. Moreover, we have noticed significant time delays between code changes and the start of test sessions, which likely correspond to developers' *selection times* (i.e., time that developers spend reasoning about which tests to run) and not automated tool runs. Therefore, we assume that developers manually selected the tests in each test session. Second, `CodingTracker` collects information about *code changes but not entire project states*. The original motivation for `CodingTracker` was a study of refactorings [23], which needed only code changes, so a design decision was made for `CodingTracker` to *not* collect the entire project states (to save space/time for storing logs on disk and transferring them to the centralized repository). However, the lack of entire states creates challenges to exactly reconstruct the project as the developer had it for each test session (e.g., to precisely count the number of tests or to compile and run tests for automated RTS). Sections 2.1.3 and 3.1.3 discuss how we address these challenges.

2.1.2 Projects Under Analysis

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

Figure 1 is a summary of test-related data that we collected[2]. For each project, we first show the number of test sessions. Our analysis showed that a large number of these sessions execute only one test. We refer to such test sessions as *single-test sessions*. Further, we found that many of these single-test sessions execute only one test that had failed in the immediately preceding session. We refer to such sessions as *debug test sessions*. Next, we show the number of *available tests*, i.e., the total number of tests in the project at the time of a test session, discussed in more detail in Section 2.1.3. Then, we show the number of *selected tests*, i.e., a subset of available tests that the developer selected to execute, including the total number of selected tests that the developer executed throughout the study and the total execution time for all test sessions[3]. Finally, we show the percentage of *selective sessions*, i.e., *multiple-test sessions* where the number of selected tests is smaller than the number of available tests; in other words, the developer performed manual RTS in each such test session by selecting to execute only a subset of the tests available in that session.

The total test execution time with manual RTS is substantially lower than it would have been without manual RTS. The sum of the "$Time^{min}$" column in Figure 1 shows that, when manual RTS is performed, the total test execution time for all developers in our study was 2,113 minutes. In contrast, had the developers always executed all available tests, we estimate[4] that it would have resulted in a total test execution time of 23,806 minutes. In other words, had the developers not performed manual RTS, their test executions would have taken about an order of magnitude more time.

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

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

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

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

```
1  // Inputs: Session info extracted from CodingTracker logs
2  List⟨TestSession⟩ sessions;
3  Map⟨TestSession, Set⟨Pair⟨ClassName, MethodName⟩⟩⟩ executed;
4
5  // Output: Available tests for each test session
6  Map⟨TestSession, Set⟨Pair⟨ClassName, MethodName⟩⟩⟩ available;
7
8  // Compute available tests for each test session
9  ComputeAvailable()
10     Set⟨Pair⟨ClassName, MethodName⟩⟩ $T = \{\}$ // Current available tests
11     available = {}
12
13     foreach $s$: sessions
14         Set⟨Pair⟨ClassName, MethodName⟩⟩ $e$ = executed($s$)
15         if $|e| > 1$
16             $T = T \setminus \{(c, m) \in T \mid \exists (c, m') \in e\}$
17         $T = T \cup e$
18         available($s$) = $T$
```

Figure 2: Algorithm for computing a set of available test methods at each test session

when some of the subsequent plots or other numbers that we report include single-test sessions.

2.1.3 Challenges

`CodingTracker` was initially designed to study how code evolves over time [23], and thus it recorded only code changes and various file activities but not the entire state of the developers' projects. As a consequence, we could not easily extract the number of available tests for each test session: while `CodingTracker` did record the information about tests that are *executed*/selected, it had no explicit information about tests that were *not* executed. Therefore, we developed an algorithm to estimate the number of available tests (reported in Figure 1). We designed our algorithm to be conservative and likely *under-estimate* the number of available tests. In other words, developers likely performed even more manual RTS than we report.

Figure 2 shows the algorithm. The input to the algorithm is a list of test sessions extracted from the `CodingTracker` logs; each session is mapped to a set of executed tests, and each test is represented as a pair of a test class and test method name. The output is a mapping from test sessions to the set of available tests. Although we extract more information for each test session, e.g., execution time, that information is not relevant for this algorithm.

The algorithm keeps track of the current set of available tests, T, initialized to the empty set (line 10). For each test session, the algorithm adds to T the tests executed in that session (line 17); those tests are definitely available. The algorithm also attempts to find which tests may have been removed and are not available any more. For each multiple-test session, the algorithm removes from T all the tests whose class matches one of the tests executed in the current session s (line 16). The assumption is that executing one test from some class c (in a session that has more than one test) likely means that *all* tests from that class are executed in the session. Thus, any test from the same class that was executed previously but not in the current session was likely removed from the project. This assumption is supported by the fact that Eclipse provides rather limited support for selection of multiple tests from the same class as discussed in Section 2.2. For single-test sessions, the algorithm only adds the executed test to T; the assumption is that the same tests remain available as in the previous session, but the developer decided to run only one of the tests. Finally, T becomes the available set of tests for the current session (line 18). Note that our algorithm does not account for removed test classes, but these are very rare in our data set. For example, we inspected in detail project $\mathcal{P}_{14}$, one of the largest projects, and no test class was deleted.

2.2 Investigating Manual RTS

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

RQ1: How often do developers perform manual RTS?

Developers performed manual RTS in 59.19±35.16% (mean ± SD) of the test sessions we studied (column "Selective Sessions" in Figure 1). Note that we first compute selective session ratio for each developer, and then an *unweighted* arithmetic mean of those ratios (rather than weighting by the number of test sessions), because we do not want developers with the most test sessions to bias the results.

Across all 2,163 multiple-test sessions in our study, the average ratio of selected tests (tests that the developer executed) to available tests (tests that could have been executed), i.e., average *test selection ratio*, was only 35.07%. Note that this number is calculated from all test sessions as if they were obtained from a single developer. We show the distribution of test selection ratios for all test sessions for all the developers using violin plots [18] in Figure 3. A violin plot is similar to a boxplot but additionally shows probability density of the data at different values.

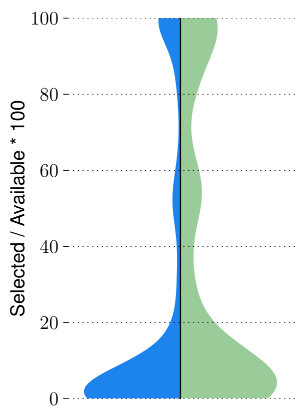

Figure 3: Distribution of test selection ratio with (left) and without (right) single-test sessions

The left part of Figure 3 shows the distribution of test selection ratios when single-test sessions are included, while the right part shows the distribution when single-test sessions are excluded. We show only one half of each violin plot due to space constraint; the missing halves are symmetric. It can be observed from the violin plots that manual RTS happens very frequently, and, most of the time, the test selection ratio is less than 20%.

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

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

Developers performed manual RTS regardless of the size of their test suites. We draw this conclusion because almost all developers in our study performed manual RTS, and they had a wide range of test-suite sizes. The average test-suite size in all 17 projects we studied was 174.27 tests (column "Available Tests" in Figure 1); the minimum was 6 tests, and the maximum was 1,663 tests. Considering that these projects are of small to medium size, and because they exhibit manual RTS, we expect that developers of larger projects would perform even more manual RTS.

We also consider the relationship between the size of recent code changes and the number of tests that developers select in each test session. One may expect that developers run more tests after large code changes. We correlate the test selection ratio with the *code change ratio* for all test sessions. The code change ratio is calculated as the percentage of AST node changes [23] since the previous test session over the total AST node changes during the entire study for a particular project. To assess correlation, we measure the Spearman's and Pearson's correlation coefficients[5]. The Spearman's and Pearson's coefficients are 0.28 (0.25 when single-test sessions are included) and 0.16 (0.16 when single-test sessions are included), respectively. In all cases, the p-value was below 0.01[6], which confirms that some correlation exists. However, the low values of coefficients imply a low correlation between the amount of code changes immediately before a test session and the number of manually selected tests in that session. This low correlation was a surprising finding as we had expected a higher correlation between code changes and the number of selected tests.

RQ3: What are common scenarios for manual RTS?

The most common scenario in which developers performed manual RTS was while debugging a single test that failed in the previous session. Recall that we refer to such test sessions as *debug test sessions*. As seen in Figure 1 (column "Single-Test Debug"), debug test sessions account for 2,258 out of the 5,757 total test sessions considered. One common pattern that we found in the data was that, after one or more tests fail, developers usually start making code changes to fix those failing tests and keep re-running only those failing tests until they pass. After all the failing tests pass, the developers then run most or all of the available tests to check for regressions. Another pattern is when a developer fixes tests one after another, re-running only a single failing test until it passes. Therefore, even if the developers had a "perfect" automated RTS tool to run after each change, such a tool could prove distracting when running many debug test sessions in sequence. Specifically, even if some code changes affect a larger number of tests, developers may prefer to run only the single test that they are currently debugging. The existence of other reasons for RTS, besides efficiency improvements, shows a need for a different class of tools and techniques that can meet these actual developer needs; we discuss this further in Section 4.

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

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

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

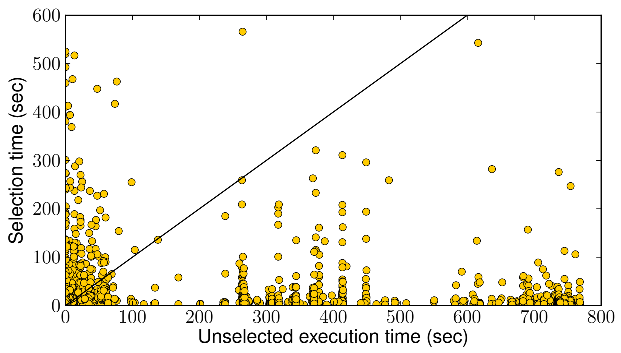

Figure 4: Relationship between selection time and (estimated) time to execute unselected tests; the plot also shows the identity line

RQ4: How do developers commonly perform manual RTS?

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

Manual RTS by Commenting: One approach used by the developers was to comment out unit tests they did not want to run. We observed that developers performed this type of selection at different levels of granularity. Some developers commented out individual test methods within a test class, while others commented out entire test classes from JUnit annotations that specify test suites. In both cases, the time overhead incurred by the developer in deciding which tests to run and in commenting out the tests, i.e., *selection time*, is likely to be non-negligible. In other words, selection time is an estimate of the time spent by developers to manually "analyze" and select which tests may be affected. Using the available `CodingTracker` data, we estimate selection time to be the time elapsed from the last code change that immediately preceded a test session and the start of the test session. We exclude selection time values greater than 10 minutes, as developers may re-run tests after taking a break from work. Our experiments with break times of 5 minutes and 20 minutes did not significantly change any of the outcomes of our study. In Figure 4, we show the correlation between selection time and (estimated) time to execute unselected tests (which is the time saved by not executing unselected tests). While the overall time savings due to manual RTS is significant, we found that in 31% of the cases (points above the identity line in Figure 4) developers could have saved more time by simply running all the tests.

Manual RTS by Selecting Various Nodes of Hierarchy: Developers also perform test selection by selecting a node of hierarchy in their IDE, e.g., they could select to run only a single test or all the tests from a single class or package. This is a critical RTS limitation in Eclipse—it restricts the developer to select to run only one node of hierarchy (in the limit this node represents the entire project such that the entire test suite for that project is run). In other words, the developer is not able to select to run an *arbitrary* set of tests or test suites. Related but different, in several projects, by browsing through the changes collected by `CodingTracker`, we noticed that developers were writing scripts (".launch" files in Eclipse) to group tests. Using a script has the same limitation as manually selecting a node of hierarchy. These limitations of Eclipse are shared by several popular IDEs as shown in Figure 5.

RQ5: How good is IDE support for manual RTS?

IDEs provide varying levels of support for performing manual RTS. The IDEs we investigated are: Eclipse[7], IntelliJ IDEA[8], NetBeans[9], and VisualStudio 2010[10].

Support for Arbitrary Manual RTS: Recall from the answer to RQ4 that, in several cases, the developers selected among tests by commenting out the tests within test classes or commenting out test classes within test suites. This likely means that developers would like to arbitrarily select tests within nodes of hierarchy. Also, our experience with running the automated RTS tool (as discussed in Section 3) shows that all affected tests may not reside in the same node of hierarchy. Thus, it is also important to be able to arbitrarily select tests across these nodes.

Figure 5 is a summary of available IDE support for selecting tests at different levels of granularity within and across nodes of hierarchy. All the IDEs allow developers to select a single test. Moreover, several IDEs offer support for arbitrary selection. IntelliJ allows to arbitrarily select tests by marking (in the GUI) each test to be run subsequently. This may be tedious for selecting among very many tests and is only available for arbitrarily selecting test classes across test packages or test methods within the same class. VisualStudio allows arbitrary selection by specifying regular expressions for test names which may match across multiple nodes of hierarchy. However, not all developers are familiar with regular expressions, and knowledge of all test names in the project is required to write them effectively. Still, based on our study, having this type of support seems very valuable, given that it is needed by the developers. More importantly, Eclipse lacks support for such arbitrary test selection.

Support for RTS across multiple test sessions: We showed in the answer to RQ3 that the most common pattern of manual RTS occurred during debug test sessions. It is likely that the changes made between debug test sessions affect more tests than the test being fixed. Indeed, we found this to be the case for project $\mathcal{P}_{14}$. It is possible that the developers do not select other tests affected by the changes due to additional reasoning required to identify such tests. Thus, their test selections during debug test sessions are likely to be unsafe and may lead to extra debug steps at a latter stage. Although VisualStudio provides some level of RTS automation, it has some shortcomings that we discuss in Section 4.

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

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

Figure 5: RTS capabilities of popular IDEs. (IntelliJ only partially supports arbitrary selection)

3. MANUAL VS. AUTOMATED RTS

We next discuss the results of our comparison of manual and automated RTS, by which we address question RQ6. We compare both approaches in terms of safety, precision, and performance using one of the largest projects from our study. As no industry-strength tool for automated RTS is available, we used `FaultTracer` [44], a recently developed state-of-the-research RTS prototype.

3.1 Methodology

We investigated in detail the data collected from one of our study participants, with the goal of comparing manual and automated RTS. We chose $\mathcal{P}_{14}$ from Figure 1 (for reasons described in Section 3.1.2). First, we reconstructed the state of $\mathcal{P}_{14}$ at every test session. Recall that `CodingTracker` does *not* capture the entire state of the project for any test session. We had to perform a substantial amount of work to find a code version that (likely) matched the point where the developer used `CodingTracker`. We acknowledge the help of the $\mathcal{P}_{14}$ developer who helped with this information, especially that the code moved from an internal repository to an external repository. It took several email exchanges to identify the potential version on top of which we could replay the `CodingTracker` changes while still being able to compile the entire project and execute the tests. Second, for each test session, we ran `FaultTracer` [44] on the project and compared the tests selected by the tool with the tests selected by the developer. Because `FaultTracer` is a research prototype, it did not support projects (in the general sense of the term "software projects") that are distributed across multiple Eclipse projects (in the specific terminology of what Eclipse calls "projects") even in the same Eclipse workspace. We worked around this limitation of `FaultTracer` by automatically merging all Eclipse projects from $\mathcal{P}_{14}$ into one project that `FaultTracer` could analyze.

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

To compare manual and automated RTS fairly and consistently, we accounted for the occurrence of multiple test sessions without intervening changes. This is because `Fault-`

[7]Kepler Service Release 1, build id: 20130919-0819.
[8]Version 12.1.6, build id: IC-129.1359.
[9]Version 7.4, build id: 201310111528.
[10]We selected VisualStudio 2010 rather than the latest version because VisualStudio 2010 was the only IDE that has ever supported automated RTS; interestingly enough, this automated RTS support has been removed from the IDE in subsequent releases!

Tracer would only select to run tests after detecting code changes between consecutive versions of the software. Our solution was to merge consecutive test sessions which had no intervening changes. Consider two consecutive test sessions, X and Y, with no intervening changes. Suppose that the tests and their outcomes for X are [test1:OK, test4:OK], and for Y are [test1:OK, test2:Failure, test3:OK]. Our merge would produce a union of the tests in X and Y, and if a test happens to have different outcome, the merge would keep the result from X; however, because the test runs happened without intervening changes, it is reasonable to expect that if some tests are re-run, their outcomes should be the same. We checked that, in our entire study, the test runs are largely deterministic and found a tiny percentage of non-deterministic tests (0.6%). The effect of non-deterministic tests on RTS is a worthwhile research topic on its own [20]. For the sessions X and Y shown above, the merged session would contain the tests [test1:OK, test2:Failure, test3:OK, test4:OK].

Having merged the manual test sessions, the number of test sessions for comparing manual and automated RTS we obtained was 683. We further limited our comparison to the first 450 of these 683 test sessions, due to difficulties in automating the setup of $\mathcal{P}_{14}$ to use FaultTracer to perform RTS between successive versions. As we studied a very large project, which evolved very quickly and had dependencies on environment and many third-party libraries, we could not easily automate the setup across all 683 merged test sessions. The 450 test sessions used constitute the largest consecutive sequence of test sessions which had the same setup. (We discuss other challenges in Section 3.1.3.) Across all 450 test sessions considered, $\mathcal{P}_{14}$ has (on average) 83,980 lines of code and 889.32 available tests.

3.1.1 FaultTracer

The inputs to FaultTracer are two program versions—old version P and new version P'—and the execution coverage of tests at version P (i.e., a mapping from test to nodes of extended control-flow graph [44] covered by the test). Let T be the set of tests in P. FaultTracer produces, as output, a set of tests $T' \subseteq T$ that are affected by the code changes between P and P'. The unselected tests in $T \setminus T'$ cannot change their behavior. Note that one also has to run new tests that are added in P' but do not exist in P.

We chose FaultTracer because it represents the state-of-the-research in RTS and implements a safe RTS technique. Also, FaultTracer works at a fine-granularity level (which improves its precision), because it tracks coverage at the level of an extended control-flow graph [44]. To identify code changes, FaultTracer implements an enhanced change-impact analysis. In addition, FaultTracer targets projects written in Java, the same programming language used in $\mathcal{P}_{14}$, so, there was a natural fit.

However, note that we chose FaultTracer from a very limited pool. To the best of our knowledge, there exists no other publicly available tool that performs test selection at such fine granularity level (e.g., statement, control-flow edge, basic block, etc.). Systems such as Google's TAP system [37,38] and Microsoft's Scout system [16,17,34] are proprietary. Moreover, TAP implements a coarse-grained analysis based on dependencies between modules, which would be overly imprecise for $\mathcal{P}_{14}$ that has only few modules.

3.1.2 Project under Analysis

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

3.1.3 Challenges

Because CodingTracker did not capture entire project state, we had to reconstruct the $\mathcal{P}_{14}$'s developer's workspace to be able to build and run tests for our analysis. Using timestamps from the CodingTracker logs, we looked for a commit in the developer's VCS which satisfied the following conditions: (1) the time of the commit matches the VCS commit timestamp recorded in the CodingTracker logs and (2) the code compiles after checking it out of the VCS and adding required dependencies. Finally, this checked-out version was imported into Eclipse and used as a basis for replaying the CodingTracker logs. By replaying the changes captured by CodingTracker on top of this initial state, we obtained the state of the entire project in every succeeding test session. Note that CodingTracker captures changes to both the project under test and the testing code, and thus, the reconstructed developer's workspace contained all the tests available at any given test session. We assume that the ability to replay the CodingTracker logs from the initial VCS commit till the end of the logs without any error means that it was a likely valid starting point. Thus, the reconstructed workspace is as close to the developer's workspace as it existed while CodingTracker monitored the developer's programming activity.

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

3.2 Comparing Manual and Automated RTS

The number of selected tests We plot, in Figure 6, the number of tests selected by manual RTS against the number of tests selected by automated RTS (i.e., FaultTracer) for each test session. A quick look may reveal that there is a substantial difference between manual and automated RTS, which we further analyze.

Figure 7 shows the distribution, across test sessions, of the number of tests selected by manual and automated RTS. We show the distribution for two cases: with ("w/") and without ("w/o") single-test sessions. It can be seen that the median is much lower for the automated tool in both cases. This implies that the developer is imprecise (i.e., selects more than necessary). Further, if single-test sessions are included, we can observe that the arithmetic mean (shown as a star) is lower for manual than automated RTS. How-

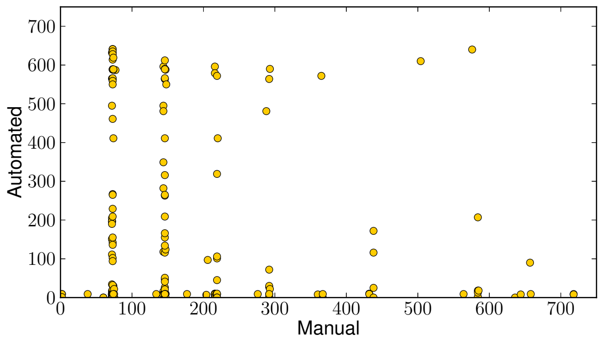

Figure 6: Relationship of the number of tests selected in each test session by manual and automated RTS for $\mathcal{P}_{14}$

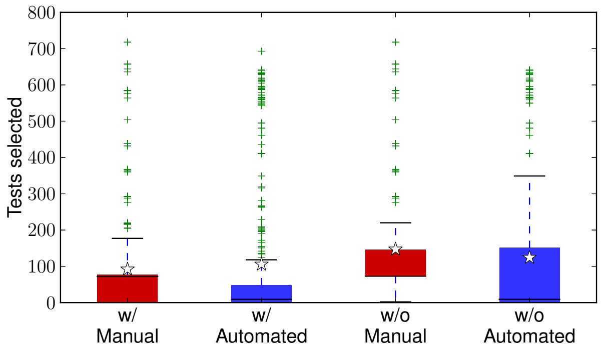

Figure 7: Distribution of selected tests for $\mathcal{P}_{14}$ with ("w/") and without ("w/o") single-test sessions

ever, when single-test sessions are excluded, we can see the opposite. This indicates, as expected, that developer focuses on very few tests while debugging and ignores the other affected tests. Finally, when single-test sessions are excluded from the manually selected tests, we found that many test sessions contain the number of tests equal to the median. Our closer inspection shows this to be due to the lack of support for arbitrary selection in Eclipse, which forced the developer to run all tests from one class.

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

Figure 6 shows the relationship between the numbers of tests selected by both approaches. The Spearman's and Pearson's correlation coefficients are 0.18 (p-value below 0.01) and 0.00 (p-value is 0.98), respectively. These values indicate a rather low, almost non-existent, correlation.

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

Correlation with code changes In Section 2.2, we found that for *all* projects in our study there is low correlation between code change ratio and manual RTS. We revisit that correlation in more detail for the $\mathcal{P}_{14}$ project. To further compare manual and automated RTS, we evaluate whether either of these selection approaches correlates better with code changes. Effectively, we re-check our intuition that the developer is more likely to select fewer tests after smaller code changes. We measured the Pearson's and Spearman's correlation coefficients for both manual and automated RTS. The values for Spearman's coefficients are 0.22 (p-value below 0.01) and 0.01 (p-value is 0.93) for manual and automated RTS, respectively. The values for Pearson's coefficients are 0.08 (p-value is 0.10) and -0.02 (p-value is 0.77) for manual and automated RTS, respectively. While the correlation is low in all cases, the slightly higher values of correlation coefficients for manual RTS may indicate that (compared to automated RTS) the developer indeed selects fewer tests after smaller changes and more tests after larger changes, as it becomes harder to reason which tests are affected by larger changes. The plot in Figure 8 visualizes the relationship, for each test session, between code change ratio and the number of selected tests for both manual and automated RTS. We can observe that manual RTS is less likely to select many tests for small changes (e.g., fewer red dots than blue dots are close to the x-axis around the 600 mark). In the end, the size of semantic effect of a change (as measured by the number of affected tests) is not easy to predict from the size of the syntactic change (as measured by the number of AST nodes changed).

Performance We finally compare manual and automated RTS based on the time taken to select the tests. Figure 9 shows the distribution of selection time (first boxplot), as defined in Section 2.1, and analysis time (second boxplot) incurred by `FaultTracer`. We can observe that the developer is faster than the automated RTS tool in selecting which tests to run (the p-value for the Mann-Whitney U test is

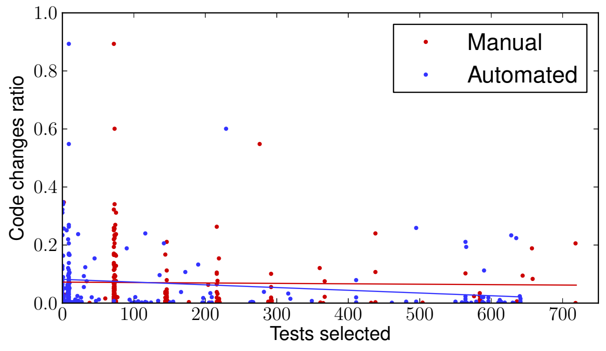

Figure 8: Relationship of manual and automated RTS with relative size of code changes for $\mathcal{P}_{14}$

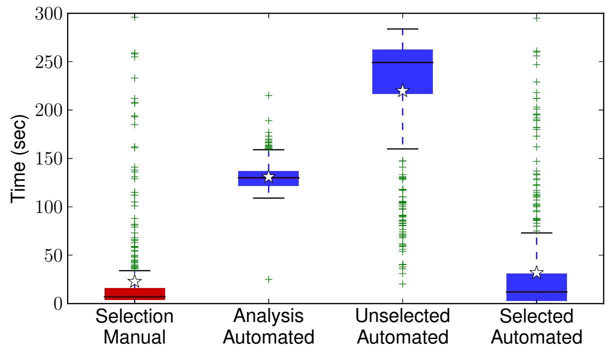

Figure 9: Distribution for $\mathcal{P}_{14}$ of estimated manual selection time (first boxplot), automated analysis time (second boxplot), execution time of the tests unselected by `FaultTracer` (third boxplot), and execution time of the selected tests by `FaultTracer` (fourth boxplot)

below 0.01). For comparison, we also show the distribution of estimated execution time for tests that are unselected by `FaultTracer` (third boxplot) and actual execution time for tests selected by `FaultTracer` (fourth boxplot). We ran all our experiments on a 3.40 GHz Intel Xeon E3-1240 V2 machine with 16GB of RAM, running Ubuntu Linux 12.04.4 LTS and Oracle Java 64-Bit Server version 1.6.0_45.

One can observe that `FaultTracer` analysis took substantial time. Although the analysis time (130.94 ± 13.77 seconds) is, on average, less than the time saved by not running unselected tests (219.86 ± 68.88 seconds), it is important to note that one may also want to take into account time to collect necessary coverage information to enable change impact analysis; if time taken for analysis plus overhead for collecting coverage plus running selected tests is longer than time taken for running all the tests, then test selection provides no benefit. This raises the question whether a fine-grained technique, such as the one implemented in `FaultTracer` [44], can be optimized to bring benefits to smaller projects. One of our planned future directions is to explore which granularity level of automated RTS techniques is appropriate for most projects. Further, we believe that research studies on automated RTS should provide more information about their complexity (e.g., time to implement the technique) and efficiency (e.g., analysis time, time to collect coverage, etc.). Previous research focused mostly on the number of selected tests (i.e., safety and precision), which is not sufficient for proper comparison and discovering a test selection technique that works in practice.

4. DISCUSSION

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

Test Selection Granularity We mentioned earlier that systems such as TAP at Google [37, 38] and Scout at Microsoft [16,17,34] are successfully used for test selection/prioritization. However, these systems are used as part of the gated check-in [9] infrastructure (i.e., all affected regression tests are executed before a commit is accepted into the central repository). In other words, they are not used (and are not applicable) on developers' machines where developers commonly work on few modules at a time (and run tests locally). Even developers at either of these companies, let alone many developers who do not develop code at the scale of Google or Microsoft, would benefit from an improved fine-grained test selection. This provides motivation for research on finding the best balance between analysis time, implementation complexity, and benefits obtained from test selection. Improved fine-grained test selection would be more widely applicable and could be used in addition to coarse-grained test selection systems.

Experience with IDE-integrated automated RTS We experimented with Visual Studio 2010, the only tool (to the best of our knowledge) that integrates automated RTS with an IDE. We did this to see if such a tool would perform better than manual RTS in terms of safety and precision. Specifically, the Test Impact Analysis (TIA) tool in Visual Studio 2010 [36] was designed to help reduce testing effort by focusing on tests that are likely affected by code changes made since the previous run of the tests. We think this is an excellent step towards improved RTS in developer environments and that similar tools should be developed for other IDEs. We successfully installed TIA and ran it on several simple examples we wrote and on an actual open-source project. However, we found a number of shortcomings with TIA. Most importantly, the tool is *unsafe*: any change not related to a method body is ignored (e.g., field values, annotations, etc.). Also, changes like adding a method, removing a method, or overriding a method remain undetected [28]. Furthermore, TIA does not address any of the issues commonly faced by selection techniques [1,3,5,7,14,15,31,40,42], such as library updates, reflection, external resources, etc. Our opinion is that a safe but imprecise tool would be more appreciated by developers.

Potential improvement of IDEs Across all projects, we observed that developers commonly select tests during debugging. Thus, one common way by which an IDE might help is to offer two separate modes of running tests, a *regular mode* and a *test selection mode*. In the regular mode, the developer may choose to re-run, after a series of code changes, one or more previously failing tests (while ignoring other affected tests). Once the test passes, the developer may run in the test selection mode to check for regressions. Notice that the test selection runs would be separated by a series of regular runs. Consider two test selection runs, A and B (Figure 10). In A, some tests were selected to be run and failed. Developer then performs (regular) runs a_1, a_2, ... a_n, until the previously failing test passes. The test selection run B is then executed to ensure that there are no regressions due to code changes, since A. Note that the analysis performed before running B should consider the difference since A and not just the difference between a_n and B; otherwise, tests affected by the changes between A and a_n would not be accounted for. As a simple optimization step, the tool could exclude the tests affected between A and B that were already run after the change that was affecting them.

5. THREATS TO VALIDITY

External: Developers, Projects, and Tools The results of our study may not generalize to projects outside of the scope of our study. To mitigate this threat, we used 17 projects that cover various domains and 14 developers with different levels of programming experience. Further, these projects vary significantly in size, number of developers, and number of tests. Regarding the comparison of manual and

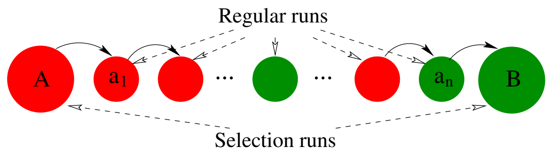

Figure 10: An example of a common pattern when developer alternates selection and regular runs

automated RTS, we used the largest project for which we could reconstruct the entire state for many test sessions.

We used `FaultTracer`, a research prototype, to perform automated RTS. Other tools [1, 3, 5, 7, 14, 15, 31, 40, 42] that implement different test selection techniques could have led to different results. We chose `FaultTracer` because it implements a safe and precise test selection technique. To the best of our knowledge, no other publicly available tool for test selection exists (except the proprietary tools that work at coarse-granularity level, which would not be applicable to any of the projects used in our study). Our experience with VisualStudio 2010 demonstrated that the implemented approach is unsafe, thus inappropriate for our study.

Finally, the patterns of test selection could differ in other languages but Java. We leave the investigation of how manual RTS is performed in other languages for future work.

Internal: Implementation Correctness We extracted data relevant to manual RTS from the study participants' recordings. To extract the data, we wrote analyzers on top of the infrastructure that we used in our prior research studies on refactorings [21, 23, 39]. Further, new analyzers and scripts were tested and reviewed by at least two authors.

Construct: IDEs and Metrics Because `CodingTracker` is implemented as an Eclipse plugin, all developers in our study used Eclipse IDE. Therefore, our study results may not hold for other IDEs. However, because Eclipse is the most popular IDE for Java [19], our results hold for a significant portion of Java developers. We leave the replication of our study using other popular IDEs (both for Java and other languages) for future work.

6. RELATED WORK

We complement existing work on automated RTS [1,3,5,7, 14,15,31,40,42] by investigating actual RTS practices of developers. Over the last three decades, many automated RTS techniques were proposed, which have mostly focused on improving the precision and safety of test selection [24]. One key difference among these techniques is the granularity level at which they detect affected tests (e.g., statement, control-flow edge, method, etc.). Recent work [10, 16, 34, 37, 38] has also reported some success with automated RTS in large software projects. Unfortunately, after decades of research, no practical RTS tool is available for widespread adoption by developers. The need for a practical tool is becoming even more urgent with the widespread adoption of agile development, which heavily relies on (regression) testing [26, 35]. In particular, although we did not account for Test Driven Development (TDD) in our study, the practical impact of such testing-oriented software development methodologies on RTS is still largely unknown. Therefore, our study aims to shed light on common manual RTS practices.

Our study is different in scope, emphasis and approach from that of Greiler et al. [13], who recently conducted a study of testing practices among developers. We did not limit our scope to a specific class of software (they focus on testing component-based software). Their emphasis is on answering important questions about the testing practices that are (not) adopted by organizations and *why*. On the other hand, we focus on *how* developers perform RTS. Finally, their approach utilizes interviews and surveys, but we analyzed data collected from developers in real time.

Concerning the empirical study of RTS techniques, the closest work to ours is the use of field study data by Orso et al. [25]. They collected usage profile data from *users* of deployed software for tuning their Gamma approach for RTS and impact analysis. We study data collected from *developers* to gain insight on improving manual RTS. Other empirical studies have been conducted, mainly to evaluate proposed automated RTS techniques [27, 30, 33, 41]. Our study is not technique-specific; rather, we investigate common manual RTS practices.

Although our work is based on data that has been used previously, this is the first use of the data for studying how developers perform testing. The data was previously used to show that VCS commit data is imprecise and incomplete for studying software evolution [23], for comparing manual and automated refactorings [21], and for mining fine-grained code change patterns [22].

7. CONCLUSIONS AND FUTURE WORK

In this paper, we provided evidence on the pervasiveness of manual RTS and compared manual and automated RTS. The results show that, while developers commonly perform manual RTS, they need better support to perform manual RTS while working on their typically-sized projects. In particular, all but two of the 14 developers in our three-month user study performed ad-hoc manual RTS in 59.19% of test sessions. The practice of manual RTS occurred regardless of project properties like project size, test execution time, and the size of code changes made before test runs. Further, we found that manual RTS is most commonly performed during debugging. By comparing manual and automated RTS, we also showed how they differ in terms of safety, precision, and performance. We found that manual RTS, compared to an automated RTS tool, selected to run more tests than necessary in 73% of the 450 test sessions that we evaluated, and in 74% of the test sessions selected fewer tests than were affected by the most recent code changes.

Future Work There are three main thrusts in our planned future work. First, we want to work on finding a better balance between automated RTS granularity level and implementation complexity, so that we can deliver a widely used tool for automated RTS. Second, we plan to create RTS techniques that incorporate the knowledge of common debugging scenarios to improve developers' debugging experience. Finally, we plan to investigate how developers manually perform other activities related to regression testing, such as test-suite reduction and test prioritization.

Acknowledgments We thank Lamyaa Eloussi, Alex Gyori, Rohan Sharma, and August Shi for helping with the experiments, Lingming Zhang for providing `FaultTracer`, and Aleksandar Milicevic for providing feedback on an early draft of this paper. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1012759 and CCF-1439957, and by Boeing under Grant No. ITI RPS #28.

8. REFERENCES

[1] T. Ball. On the limit of control flow analysis for regression test selection. In *ISSTA*, 1998.

[2] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *TOSEM*, 2001.

[3] L. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *IST*, 2009.

[4] CodingTracker. http://codingtracker.web.engr.illinois.edu/.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 1978.

[6] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *SQJ*, 2004.

[7] E. Engström and P. Runeson. A qualitative survey of regression testing practices. In *PROFES*, 2010.

[8] K. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *NTC*, 1981.

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

[10] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov. Regression test selection for distributed software histories. In *CAV*, 2014.

[11] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *TOSEM*, 1975.

[12] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *ICSE*, 1998.

[13] M. Greiler, A. van Deursen, and M. Storey. Test confessions: A study of testing practices for plug-in systems. In *ICSE*, 2012.

[14] 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 *OOPSLA*, 2001.

[15] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *ICSM*, 1988.

[16] J. Hartmann. Applying selective revalidation techniques at Microsoft. In *PNSQC*, 2007.

[17] J. Hartmann. 30 years of regression testing: Past, present and future. In *PNSQC*, 2012.

[18] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 1998.

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

[20] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, 2014. to appear.

[21] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *ECOOP*, 2013.

[22] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *ICSE*, 2014.

[23] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP*, 2012.

[24] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications*, 1998.

[25] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *FSE*, 2003.

[26] D. Parsons, T. Susnjak, and M. Lange. Influences on regression testing strategies in agile software development environments. *SQJ*, 2013.

[27] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *ISSTA*, 2008.

[28] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*, 2004.

[29] G. Rothermel and M. Harrold. Selecting regression tests for object-oriented software. In *ICSM*, 1994.

[30] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *TOSEM*, 1998.

[31] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 1997.

[32] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, 1999.

[33] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, 2004.

[34] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, 2002.

[35] D. Talby, A. Keren, O. Hazzan, and Y. Dubinsky. Agile software testing in a large-scale project. *Software*, 2006.

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

[37] Testing at the speed and scale of Google, 2011. http://goo.gl/OKqBk.

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

[39] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, 2012.

[40] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *ICSM*, 2005.

[41] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, 1997.

[42] S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *COMPSAC*, 1987.

[43] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 2012.

[44] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, 2011.