SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2015; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Prioritizing Manual Test Cases in Rapid Release Environments

Hadi Hemmati[1]* Zhihan Fang[2], Mika V. Mäntylä[3], and Bram Adams[4]

[1]*University of Manitoba, Canada*
[2]*Rutgers, The State University of New Jersey, USA*
[3]*University of Oulu, Finland*
[4]*Polytechnique Montreal, Canada*

SUMMARY

Test case prioritization is one of the most vital testing activities, in practice, specially for large scale systems. The goal is ranking the existing test cases in a way that they detect faults as soon as possible, so that any partial execution of the test suite detects maximum number of defects for the given budget. Test prioritization becomes even more important when the test execution is time consuming, e.g., manual system tests vs. automated unit tests. Most existing test case prioritization techniques are based on code coverage, which requires access to source code. However, manual testing is mainly done in a black-box manner (manual testers do not have access to the source code). Therefore, in this paper, we first examine the existing test case prioritization techniques and modify them to be applicable on *manual black-box system testing*. We specifically study a *diversity-based* and a *risk driven* approach for test case prioritization. Our empirical study on four older releases of Desktop Firefox shows that none of the techniques are strongly dominating the others in all releases. However, when we study nine more recent releases of Desktop Firefox, where the development has been moved from a traditional to a more agile and rapid release environment, we see a very signifiant difference (on average 77% effectiveness improvement) between the risk-driven approach and its alternatives. The higher effectiveness of the risk-driven approach compared to alternatives also repeats, when we examine 28 new releases (following rapid release) of other Firefox projects – Mobile Firefox (on average 21% improvement) and Tablet Firefox (on average 22% improvement). Our conclusion, based on these case studies of 41 releases of Firefox Mozilla is that test cases in rapid release environments can be very effectively prioritized for execution, based on their historical riskiness; whereas the same conclusions do not necessarily hold in the traditional software development environments. Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Test case prioritization, Rapid release, Manual testing, Text mining, Risk, Historical data

1. INTRODUCTION

Testing has always been one of the main methods of software quality assurance in industry [1]. The emphasis on testing has been growing with the wide spread application of Agile methodologies and continuos integration [2]. Such methodologies and approaches suggest running all tests after each and every small change, which stops postponing the potential debugging and bug fixing activities to the end of the iteration/release. This also eases the debugging process due to the small modifications that should be investigated, per test failure. However, rerunning all tests on large scale systems is not possible, even if it is scheduled once a day as a nightly integration build [2]. So to follow continuos integration principles in large scale systems, we need to choose a subset of all test cases to be executed in each build; or ideally, prioritize the test cases so that depending on the time constraints of the build, only the most important tests be executed.

*Correspondence to: E-mail: hemmati@cs.umanitoba.ca

Test case prioritization is not a new concept in software testing. In the context of regression testing, researchers have proposed several techniques to effectively prioritize the existing test cases. Among them coverage-based heuristics (prioritizing tests with higher code coverage, *e.g.*, statement coverage) have been very popular [3]. The main assumption in such techniques is the availability of code coverage information (or accessibility of the source code to calculate such info). Unfortunately, that is unlikely in the manual system-level testing, where the testers (often non-technical people) only have access to the system as a black-box. These types of testing is mostly done through the system's graphical user interface, rather than calling source code methods in automated tests. Thus such test scripts do not reveal even the APIs of the source code, which could be used for test prioritization.

One potential solution is instrumenting the code by the development team to record the code coverage of the test cases, while being executed by the manual testers, and use this information in the later releases. This approach has three major issues: 1) the cost of instrumenting, recording, and maintaining all these coverage data, per release, if it is not already in place, 2) incompleteness of such coverage data for the new test cases (you only know the code coverage if you already have executed the exact test), and 3) inaccuracy of such coverage data on the new or modified source code. Another possible solution is using requirement-level coverage information [4]. Again, these types of approaches require additional information in terms of requirement documents that are not necessarily available for many commercial and open source systems.

Therefore, in this paper, we first propose three techniques based on the existing high-level ideas that can be applied, without any prerequisite, on manual black-box testing. The techniques under study fall under these three high-level ideas:

- **Text diversification:** where the heuristics is prioritizing test cases that are textually more diverse from the currently executed ones [5].
- **Topic modelling based prioritization:** where the idea is using topic modelling to abstract textual representation of test cases into vectors of topic memberships. Then diversify the prioritized tests with respect to their topics [6].
- **Risk-driven test prioritization:** where the high-level idea is prioritizing the risky test cases in each release. We define the test case riskiness as their number of failures in the past. [7, 8]

We then implement and exercise these techniques on four releases of Mozilla Desktop Firefox, which are developed by a traditional development approach, *i.e.*, yearly releases. The results show that none of the proposed techniques are significantly dominating the others, in all four releases. In addition, they are not much more effective than a simple random prioritization.

We then study the same techniques on nine more recent releases of Mozilla Desktop Firefox, where the development team has switched to a rapid and frequent release development methodology. The results of the study show that, interestingly, this time one approach, our risk-driven approach, is by far and consistently dominating the others (on average 77% more effective in terms of the APFD measure [3], which will be defined in Section 4).

Finally, we apply the same techniques on 28 releases of Mobile and Tablet Firefox projects (all following rapid release) and we observe the same type of results with respect to the effectiveness of the risk-driven approaches. The strong results of risk-driven approach not only suggest a potential tool for developers/testers in the rapid release community to prioritize their tests, but also promote rapid release over traditional development, due to the lack of an effective alternative in the traditional development context.

Note that this paper is an extended version of our published conference paper [9]. The main difference between the two is the extra studies on the Mobile and Tablet Firefox projects (RQ3), which are conducted in the journal version to examine the generalizability of the results to other projects than the original study. In addition, a modified risk-driven approach is also introduced and examined in the journal version. Finally, some modifications and corrections are applied on the parameter tuning and the topic-based prioritization algorithm, based on the conference version's reviews.

DOI: 10.1002/stvr

2. BACKGROUND AND RELATED WORK

In this section, we shortly review different test prioritization techniques, explain the related work, and introduce rapid release and compare it with the traditional software development.

2.1. Test case prioritization

One of the challenges of software testing is optimizing the order of test case execution in a test suite, so that they detect maximum number of faults for any given testing budget. The testing budget is always limited and thus not enough for exercising the massive test suites of large scale systems. Three typical solutions that are studied in the literature are test suite reduction, test case selection, and test case prioritization. Test suite reduction [10] usually removes redundant test cases from a test suite and test case selection [10] selects the most fault-revealing tests based on a given heuristic. However, test case prioritization (TCP) [10] focuses on ranking all existing tests, without eliminating any test case. In other words, when test cases are prioritized, one executes the test cases in the given order, until the testing budget is over. Thus a TCP's goal is to *optimize the overall fault detection rate of the executed test cases, for any given budget*.

There are several TCP techniques available [10], where each technique may have access to different types of information and uses different heuristics to achieve the TCP's goal. In the rest of this section, we summarize the key TCP techniques, based on their inputs and heuristics.

2.1.1. Input resources for TCPs The main input resources available for a TCP technique are as follows:

Change information: A TCP in the context of regression testing usually analyzes the source and test code before and after each change and identifies (directly and indirectly) affected parts of the code. The TCP may then prioritize test cases that execute the affected parts over the other tests. The emphasis of this type of approaches is on change impact analysis [11, 12].

Historical fault detection information: TCPs may use fault detection information of each test case on the previous versions of the software, as a basis to identify its riskiness [13, 7]. The high-level heuristic is that the test cases that failed in the past (detected faults), are riskier and should be ranked higher. A variation of this high level idea may assign higher rank to more severe faults and their corresponding test cases [14]. In a recent study by Elbaum *et. al.* [8], previous faults have been used as a basis of prioritization in the context of continuous integration at Google.

Dynamic and static coverage data: Another common resource that is being used in TCPs is the code coverage of each test case [3, 10]. The coverage can be of any sort, *e.g.*, statement, branch, and method coverage. Such coverage can be obtained by dynamically analyzing the program execution or by statically analyzing the test and source code. The dynamic analysis is more accurate, but it requires a real execution of the test cases. Note that executing test cases to calculate the coverage is not an option for TCPs due to the nature of the problem (the limited budget). Therefore, the dynamic coverage data can only be used, if they are already available from the previous executions. But this data may not be perfect since for instance the coverage data for the new test cases is not available from previous test suite executions. Also, the code change from a release to another may reduce the accuracy of such coverage data, even for the old test cases running on the modified source code. In addition, in many scenarios, instrumenting the code for dynamic analysis and keeping all the coverage data from the previous versions are not practical.

In the absence of execution information, coverage-based TCP techniques rank the test cases solely based on the static analysis of the test cases and/or the source code. For example, one can calculate method coverage of test cases, by extracting the sequences of method calls in the source code for the given test cases, by static analysis [15]. Of course, the availability of test scripts that contain such information, *e.g.*, method calls, is a prerequisite here, which does not hold in some cases like manual test cases written in natural languages.

Specification models or requirement documents: In model-based testing, TCPs have access to the specification models of the software under test. Test cases in this context are generated from the model. For example, a typical scenario is specifying the software by a state machine and test cases

DOI: 10.1002/stvr

by paths in the state machine. A TCP, in this example, would prioritize test cases, knowing which paths in the model are executed by which test cases [16, 17, 18]. There are also some studies that prioritize test cases based on software requirement artifacts [4, 19, 20]. However, all the approaches in this category require extra information about the software that is not commonly available, when prioritizing test cases.

Test scripts: Finally, there are a few TCP techniques that only look at the test scripts as a source of information. These TCPs are usually applicable in a wider range of domains, where the other mentioned sources of data may not be available. For instance, in [6] the authors derive a topic model from the test scripts, which approximates the features that each test case covers. There are also cases where the test scripts are taken as strings of words, without any extracted knowledge attached to them. Given such data per test case, TCPs in this category may target different objectives such as diversifying test cases or maximizing their coverage, which we explain in the next subsection.

Note that availability of these resources is very context dependent. However, in general, the testing type heavily influences the available input resources. For instance, in the case of white-box unit testing, TCPs typically have access to both test and source codes [3], but in black-box testing the TCP only have access to the test code and potentially specification or requirements information. In the context of this paper, the only available data is the test cases written in natural language and the historical fault detection information.

2.1.2. TCP Objectives So far we reviewed some of the most common input sources that are available for TCP. Given an input, a TCP uses a heuristic to optimize its objective on that input. Two common objectives from the literature are as follows:

Maximizing coverage: Since coverage info is one of the most used resources for TCP, heuristics based on coverage are also well-studied. Given the coverage of the test cases, one common heuristic is assigning higher rank to test cases that examine uncovered parts of the code. For example, a common objective is maximizing (additional) coverage [3] by a greedy algorithm. Maximizing coverage has also been done by clustering [21] or evolutionary search algorithms [22].

Diversifying test cases: More recently, researchers have also proposed another heuristic, diversity-based TCP, which tries to spread the testing budget evenly across different parts of the code [5]. The hypothesis is that similar test cases detect the same faults and thus we should exercise a diverse set of test cases to detect more faults [23]. Diversification of test cases can be applied on different levels, *e.g.*, method calls [16], extracted topics of the test cases [6], and the text of the test scripts [5].

In Section 3, we will cover three most relevant TCP techniques to our case study, in more details.

2.2. Rapid Releases

Speed in delivering software has become vitally important in software development. Some even claim that increasing it should be the top priority of the software development: "Increasing speed trumps any other improvement software R&D can provide to the company" [24]. Companies offering web-based services have taken this to the extreme, e.g., Amazon deploys software on every 11.6 seconds, on average [25]. However, the desire to increase speed in software development is not limited to companies operating in web services. According to our recent literature review [26], rapid releases are practiced in multiple domains including automotive, finance, telecom, and even in high reliability domains like space and energy. Obviously, companies operating in those domains do not deploy as frequently as Amazon, yet they are deploying faster than they use to.

Rapid releases originate from several sources [26]. Agile software development highlights the importance of rapid releases, as one of its principles states "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale" [27]. Similarly open source software development recognizes the importance of rapid releases with a well-known slogan from Raymond's book: [28] "Release early. Release often". The change to rapid releases may also be motivated by declining market share. This happened in Mozilla Firefox browser project as it was losing market share to Google Chrome who already utilized the rapid

DOI: 10.1002/stvr

release model. Therefore, Mozilla Firefox changed its release model from traditional, annual release, to rapid releases where a new version is released once every six weeks.

In this paper, we use data from the Mozilla Firefox projects transition to rapid releases. This change has been studied in the past. Mantyla et al. [26] found that switch to rapid releases makes testing easier because the scope is narrower. On the other hand, testing in the rapid releases becomes even more deadline oriented. The increased speed in testing also made it more difficult to attract larger volunteer tester community and forced Firefox project to use more sub-contractors. However, these large changes to the testing process neither produced a significant decline in the quality of the Firefox browser [29] nor did change the source code patch life cycle [30]. In this paper, we will look at the differences between rapid and traditional releases, in the context of test case prioritization.

3. BLACK-BOX MANUAL TEST PRIORITIZATION

In this paper, we are interested in the TCP problem in the context of manual system-level black-box testing. This type of tests can also be used for acceptance testing. These tests usually explain the steps in a natural language (*e.g.*, instructions in English) and no information from the code or APIs are available to extract. This limits the number of applicable TCPs. In this paper, we have modified and analyzed three most relevant existing TCP techniques, to our context.

3.1. Text diversity-based TCP

As we discussed in Section 2.1, diversifying test cases is a common technique for TCP. For example, a TCP can maximize the diversity between test cases, where each test case is represented by a sequence of method calls (statically [6] or dynamically [21] extracted). In [5], the authors applied a diversity-based approach directly on the test script without extracting their method calls. We call this approach a *text diversity-based* TCP. The text diversity-based technique treats test cases as single, continuous strings of words. The technique uses common string distance metrics, such as the Hamming distance, on pairs of test cases to determine their dissimilarity. The intuition is that if two test cases are textually similar, they will likely exercise the same portion of the source code and therefore detect the same faults [5, 31].

To measure the distance between two strings (i.e., test cases), Ledru et al. consider several distance metrics, including Euclidean, Manhattan, Levenshtein, and Hamming. The authors find that the Manhattan distance has the best average performance for fault detection [5].

To maximize diversity between strings (test cases), Ledru et al. [5] use a greedy algorithm that always prioritizes the test case which is furthest from the set of already-prioritized test cases. To do so, they define a distance measure between a single test case and a set of test cases. For a test case T_i, the set of already-prioritized test cases PS, and a distance function $d(T_i, T_j)$ which returns the distance between T_i and T_j, the authors define the distance between T_i and PS to be:

$$\text{AllDist}(T_i, PS, d) = \min\{d(T_i, T_j) \mid T_j \in PS, j \neq i\}. \quad (1)$$

The authors choose the min operator because it assigns high distance values to test cases which are most different from all other test cases. The greedy algorithm iteratively computes the AllDist metric for each un-prioritized test case, giving high priority to the test with the highest AllDist value at each iteration [5]. This algorithm has also been used by Thomas *et. al.*, in [6] as a baseline of comparison.

Our edition of text-diversity-based TCP uses the same algorithm as described in [5] and [6], but instead of applying it on the test scripts written in programming languages, we apply it on the English texts of manual test cases. More details about it will be discussed in the experiment design subsection in Section 4.

DOI: 10.1002/stvr

Testcase ID #: 10766

Summary: [Bookmarks: Tags] Add a tag to a bookmark - 'typed text' via bookmark edit dialog

Product: Firefox

Branch: 4.0 Branch

Steps to Perform:
1. Ensure you have already created at least one bookmark.
2. Navigate the browser to one of your previously bookmarked pages.
3. Click on the "Star" in the right end of the location bar.
4. In the resulting dialog, type a unique word in the **Tag:** field, then click "Done"
5. Click on the "Star" again.

Expected Results:
1. Ensure that the tag is present in the dialog
2. Also ensure the tag is present in the bookmark in the Library by navigating to that same bookmark there.
3. While in the Library, open the "Tag" folder in the folder tree view on the left. Ensure the tag is present there.

Figure 1. A typical manual system-level test case in Mozilla Firefox project.

3.2. *Topic diversity-based TCP*

Following the taxonomy of Section 2.1, a topic diversity-based approach uses test scripts as inputs and applies diversification as objective. The test scripts in this case are not the typical programming scripts but are rather instructions for manual testers. As explained before, a typical diversification objective would maximize the test cases' diversity in terms of covering different parts of the source code. However, given the inputs in our context, we have to redefine diversification objective. To do so we have adopted an information-retrieval based TCP that works on test scripts. The TCP uses a concept from text mining domain, called *topic modelling*.

The idea behind a topic modelling-based TCP is that *if you do not test a topic you won't find defects related to that topic*. In [6], the authors proposed a black-box topic-based TCP that uses a topic modelling technique called Latent Dirichlet Allocation (LDA) [32] to approximate business concerns of the software under test. They applied LDA on linguistic data in the test scripts (identifier names, comments, and string literals) and extracted the topics for each test cases. The goal was to diversify test cases with respect to their topic memberships. In this paper, we use the same approach but instead of applying it on the linguistic data of the test scripts (e.g., JUnit tests that we don't have), we apply it on the textual content of our manual test cases. To understand more, let us explain the technique in details.

Though very limited, but the textual instructions in the manual system-level tests contain information about the features being tested. For example, Figure 1, is a sample test case from our case study, which is testing a bookmarking feature of the Desktop Firefox browser.

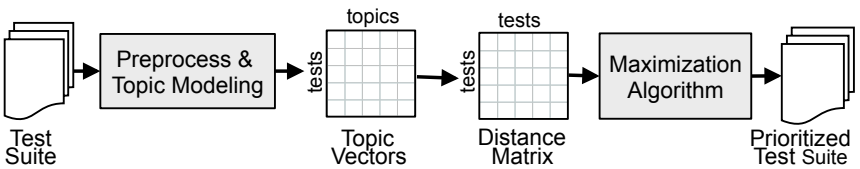

Figure 2. Overview of a topic-based TCP technique [6].

DOI: 10.1002/stvr

One way of summarizing the textual information in the test cases is using a topic modeling technique like LDA [32]. In general, LDA, works in two steps [6]: "The first step analyzes the statistical co-occurrences of the words in the documents and creates a set of topics". For instance, in the previous example, LDA extracts a set of words containing "bookmark", "tag", "folder", etc. that are mostly occur together when the test case is about testing a feature of bookmarking. "Each topic is then defined as a probability distribution over the unique words in the corpus (set of documents). Words with high probability in a given topic tend to co-occur frequently. (Note that the distance between any two words in the document is not important, only whether they both occur in a given document). The second step of topic modeling assigns topic membership vectors to each original document. A topic membership vector indicates the proportion of words in a document that come from each topic. The inferred topics are defined as probability distributions over the corpus vocabulary, and documents are defined as probability distributions over topics. The number of topics to be created, K, is a parameter to the model" [6].

In our paper, we apply LDA on the textual description of test cases (including the summary text) and its expected results (see Figure 1). Therefore, each document is a test case and each inferred topic is a collection of words that co-occur in several test case descriptions. The goal is to prioritize test cases that are more dissimilar to the already prioritized test cases. The exact algorithm is as follows:

Assume we have n test cases and m topics (extracted from those test cases). Let us define a test case tc_i with a topic membership vector of $<p_{i1}, p_{i2}, p_{i3},...p_{im}>$, where p_{ij} is the proportion of words in tc_i that belong to $topic_j$ (where SUM($p_{i1}, p_{i2}, p_{i3},...p_{im}$) = 1) . The goal of our topic-diversity-based TCP is to rank the n test cases for execution, in a way that, always, the next test case to run among all options, is the test case that makes the set of executed tests most diverse with respect to topics that have been covered.

To do so, we calculate the distance between all pairs of test cases (given a distance function) and record that in a matrix called *Distance Matrix*. Then we prioritize test cases by maximizing their average distances to already prioritized test cases, using any of several distance maximization algorithms (e.g., greedy, clustering, or genetic algorithms).

Figure 2 summarizes the steps in our topic-diversity-based TCP. Note that there are still some details about the process that we have not explained yet, such as pre-processing of the test cases and tuning the LDA parameters, which we will discuss about, in the experiment design section (Section 4).

3.3. Risk-driven clustering

The last technique used in this study uses the historical fault detection information, described in Section 2.1. This TCP requires access to the previous execution results of the test cases, which we had, in this case study. A common risk-driven TCP [13] only examines the last execution of the test cases. One can extend it to as many previous executions as possible [7, 8]. If some tests were failed before, we have to make sure that we run those tests with the current version (if they are still applicable). This technique can be combined with other TCP techniques, as well. For example, one can prioritize the previously failed test cases using a coverage-based approach to provide a full ordering of the test cases. In our paper, we use a variation of this generic approach that creates several clusters with different riskiness factors similar to the work in [8], which was published at the same time as the conference version of this paper.

In the original version of our algorithm, the highest risk is assigned to the tests that failed in the immediate version before the current version. The next riskiest cluster are tests that are not failed in the previous version but failed in the two versions before the current version, and so on. After the failing test cases, we assign high priority to test cases that exist in the previous version, but they were not failing (again the tests from more recent versions have higher priority). Finally, we append the new tests.

To be more precise, assume we have n releases, where each release (i) has several failing ($FT(i)$) and passing test cases ($PT(i)$). Now assume we have a set of test cases ($TS(n+1)$) for release $n+1$.

DOI: 10.1002/stvr

The test cases of (*TS(n+1)*) will be clustered (the riskiest cluster is C_1 and the least risky cluster is C_{2n+1}) based on their riskiness as follows:

$\forall\ tc_x \in TS(n+1)$
$C_1 = \{\ tc_x \text{ — } tc_x \in FT(n)\ \}$
$C_2 = \{\ tc_x \text{ — } tc_x \notin C1 \text{ AND } tc_x \in FT(n\text{-}1)\ \}$
$C_3 = \{\ tc_x \text{ — } tc_x \notin \cup(C_1, C_2) \text{ AND } tc_x \in FT(n\text{-}2)\ \}$
...
$C_n = \{\ tc_x \text{ — } tc_x \notin \cup(C_1,...,C_{n-1}) \text{ AND } tc_x \in FT(1)\ \}$
$C_{n+1} = \{\ tc_x \text{ — } tc_x \notin \cup(C_1,...,C_n) \text{ AND } tc_x \in PT(n)\ \}$
...
$C_{2n} = \{\ tc_x \text{ — } tc_x \notin \cup(C_1,...,C_{2n-1}) \text{ AND } tc_x \in PT(1)\ \}$
$C_{2n+1} = \{TS(n+1) - \cup(C_1..C_{2n})\}$

In another version of our approach we assign the highest priority to the new test's cluster and then start ranking the old failing tests (see more details in Section 4).

Finally, we need a method to rank the test cases within each cluster, to provide a full ordering of test cases in our TCP. This can be done by using any other TCP technique for the test cases of a cluster. For example, one can just randomly order them or use a coverage-based or diversity-based TCP. In (Section 4), we explain these details in our experiment design.

4. EMPIRICAL STUDY

In this section, we empirically evaluate different TCP techniques both in the context of traditional software development and rapid release environments. We explain our research objectives, questions, design, and results on a set of experiments with three TCP techniques that we have adopted to the domain of black-box system level test prioritization, when the tests are written in natural languages.

4.1. Research objectives and questions

The objective of this research is to examine the effectiveness of well-known heuristics for test case prioritization in a special context, where the type of information that is available for the TCP technique is very limited. In this context, not only the tests are black-box, which prevents a TCP technique to have access to the source code, but they are also written in natural languages. Having tests in natural languages limits the ability to extract an accurate model of the test execution from the test case. For example, one type of heuristic that a TCP technique could use, if the tests were automated scripts with a proper test driver code in a programming/scripting language, is to maximize API/method coverage. This is because of the fact that using such test scripts one can model each test case with a sequence of API/method calls, even in a black-box system-level testing. Such models can then be used for test prioritization. However, there are several situations that the test cases (specially system-level ones) are not automated test scripts. These tests, which are designed by test designers, are mainly aimed for testing the system through its GUIs, by manual testers. The regression test suites of this types of tests can grow to a degree that the limited manual testing resources of the company can not handle them all, in a timely fashion. So on one hand, we have large test suites to prioritize, and on the other hand, the common TCP heuristics are not directly applicable (or empirically evaluated) in this domain.

The TCP problem could be less critical, if we would follow a traditional development approach and spend a large amount of time at the end of each release/iteration for testing (which could be used to retest the entire test suites). However, the development paradigm is shifting more toward rapid releases and continuous deliveries [2]. In such an environment, rerunning the entire test suite before each release might not be an option, due to the time constraints, specially if the tests are manual test cases. Therefore, it is very critical, for the success of the development team with a rapid release strategy, to have an effective TCP technique in place (this can be even part of their build process).

DOI: 10.1002/stvr

To achieve this objective (finding an effective TCP for manual black-box tests), we have conducted an experiment to answer the following research questions:

RQ1: What is the most effective TCP technique for black-box manual testing, in a traditional software development context? In this research question, we compare the three TCP techniques introduced in Section 2 (topic-diversity, text-diversity, and risk-driven). We examine the techniques on four old releases of Mozilla's Desktop Firefox, where the development strategy was not rapid release.

RQ2: Does the relative effectiveness of the evaluated TCP techniques from RQ1 change, when the development moves toward rapid-releases? To answer this question, we repeat the RQ1 experiment with nine more recent releases of Desktop Firefox, where the development strategy was Rapid release.

RQ3: Is the effectiveness of TCP approaches consistent across projects, following rapid release? To answer this question, we apply the same TCPs on 28 releases of Mobile and Tablet Firefox projects. We also examine the effect of modifying the risk-driven TCP, when the new tests cluster has become the highest priority cluster.

4.2. Subjects of the study

In this paper, we use system testing data from Mozilla Firefox web-browser projects (Desktop, Mobile, and Tablet). We compare the system testing data of two release models for Desktop Firefox. The traditional release model (TR) was used until March 2011. The rapid releases (RR) started from version 5.0 and has been practiced ever since. We partly use system testing data that we have used in our past work to study changes in the testing process [26]. Our past work does not include using this data to study the usefulness of test-suite prioritization algorithms, which is the focus of this paper.

****

We collected the data from Litmus system, which, as explained by a Mozilla QA engineer, is used for regression testing at Mozilla: *"We use it primarily to test past regressions . . . and as an entry point for community involvement in release testing"* [26]. It consists of written natural language test case description as exemplified in Figure 1.

We web crawled the Litmus system to get the test cases and execution results from the full functional test suites of versions 2.0 to 13.0 of the Mozilla Firefox project. The data collection found 1,547 unique test cases for a total of 312,502 test case executions across 6 years of testing (06/2006–06/2012), performed on 2,009 software builds, 22 operating system versions and 78 locales. Our dataset ends to Firefox release 13.0 as after that Firefox started to use another system testing service from which we have no data. Table I shows the statistics about the 13 (four traditional and nine rapid) releases under study.

****

4.3. Case study design

The experiments conducted in this study were quite similar for RQ1, RQ2, and RQ3, with the only difference on their subjects. Since the risk-driven TCP explained in Section 3.3 can be implemented in several ways, we have examined two variants of it (differing in the way intra-cluster entries are ordered) and answered RQs by comparing five TCPs: a random TCP (as a baseline of comparison), topic-diversity, text-diversity, and the two risk-driven TCPs. The first three TCPs can be applied on any test suite without any extra information, but the two risk-driven approaches require some extra information, *i.e.*, historical execution results. In RQ3, we have an extra study where we take the best risk-driven approach and modify the priority of the cluster of new test cases in the algorithm and compare the results.

4.3.1. Design decisions in TCPs Next we explain the design decisions on the implantation of the TCPs that we have used in the experiments.

RandomTCP: Random ordering of test cases is often used as a baseline of comparison for a TCP to set the minimum acceptance bar. RandomTCP does not have any special setting.

DOI: 10.1002/stvr

Table I. Systems under test

Type	Firefox Release	Release date	No. of tests	No. of faults	Failure rate
TR	3.0	12/2006	580	127	21.90%
	3.5	7/2008	766	138	18.02%
	3.6	8/2009	828	88	10.63%
	4.0	2/2010	997	150	15.05%
RR	5.0	4/2011	1055	6	0.57%
	6.0	4/2011	1119	4	0.36%
	7.0	5/2011	1111	4	0.36%
	8.0	7/2011	1119	7	0.63%
	9.0	8/2011	1114	4	0.36%
	10.0	9/2011	1108	12	1.08%
	11.0	11/2011	1121	3	0.27%
	12.0	12/2011	1121	2	0.18%
	13.0	2/2012	1189	4	0.34%
Mobile	16.0	6/2012	364	79	21.70%
	17.0	8/2012	367	72	19.62%
	18.0	8/2012	366	87	23.77%
	19.0	10/2012	388	76	19.59%
	20.0	11/2012	381	95	24.93%
	21.0	1/2013	389	72	18.51%
	22.0	2/2013	423	79	18.68%
	23.0	4/2013	426	76	17.84%
	24.0	5/2013	476	82	17.23%
	25.0	7/2013	514	83	16.15%
	26.0	8/2013	343	50	14.58%
	27.0	9/2013	434	84	19.35%
	28.0	11/2013	380	28	7.37%
	29.0	12/2013	293	21	7.17%
Tablet	16.0	7/2012	368	77	20.92%
	17.0	8/2012	372	77	20.70%
	18.0	9/2012	377	98	25.99%
	19.0	10/2012	385	76	19.74%
	20.0	11/2012	391	62	15.86%
	21.0	1/2013	390	67	17.18%
	22.0	2/2013	396	74	18.69%
	23.0	4/2013	446	88	19.73%
	24.0	5/2013	448	42	9.375%
	25.0	7/2013	490	82	16.73%
	26.0	8/2013	333	50	15.06%
	27.0	9/2013	372	61	16.40%
	28.0	11/2013	345	39	11.30%
	29.0	12/2013	319	31	9.72%

TextDiversity: This approach can be applied on the test cases without any preprocessing. However, to avoid confounding factors in our experimentation, we apply the very same preprocessing as the TopicDiversity (explained in the next subsection) for TextDiversity, so that we only compare the effect of the TCP technique not the preprocessing. The only parameter left to set is the distance function. Our tool, which is an open source TopicDiversity/textDiversity tool, called *tcp.lda*, provides an option for text-diversity based prioritization, which has two built-in distance functions: Manhattan [6] and Euclidean [6] (they are selected in this tool due to their promising

DOI: 10.1002/stvr

results in [5]). In our experiment, we use Manhattan distance which showed the best results in [5] for TextDiversity.

TopicDiversity: Diversifying the topic memberships within the prioritized test cases is the goal of a topic-diversity-based TCP. In Section 3, we already have introduced the basic idea behind the approach and its general process, which consists of data preprocessing, topic extraction, and distance maximization. The data preprocessing is partially context-dependant. In our context, we deal with black-box test cases of a web-browser. Our test cases usually include a URL, but the main objective of the test is not verifying the correct loading of a specific page, but rather verifying a functionality of the browser, on any given website. Therefore we exclude all URLs. This avoids URLs to become part of the topics. It worth mentioning that, in general, not always the test cases are URL-independant. However, our TCP approaches do not examine the input data (URLs in this case) and only focuses on the test design.

The rest of preprocessing is quite standard in text-mining [33]. We first remove special characters (*e.g.*, "&", "!", "+") and numbers. Next, we split names based on camel case and underscore naming schemes, for example turning "identifierName" into the words "identifier" and "name". Next, we stem each word into its base form, for example turning both "names" and "naming" into "nam". Finally, we remove common English-language stop words, such as "the", "it", and "is". These steps help the topic modeling technique (LDA) to operate on a cleaner dataset and create more meaningful topics.

For the topic extraction step, we use *tcp.lda*, which uses LDA [34] for topic modelling. We have used the default values for internal LDA parameters (iteration=200, alpha=0.1, and beta=0.1), where LDA was shown to be not very sensitive to their changes, in the analysis that we conducted in [6]. However, we did a sensitivity analysis on the other parameter of LDA (*i.e.*, K), which LDA used to be more sensitive to. K defines the number of topics to be extracted. Studies recommend anything between 5-500 [6]. For our small size corpuses, we use a default K=15 and compare the results if we would have used K=5, 10, 25, and 50, instead. Each TCP is executed 10 times and the median APFD, which we introduce later in this section, has been reported in Table II and discussed in Section 4.4.1. The choice of distance function in topicDiversity is also studied by using Manhattan as the default and Euclidean as an alternative, which again has been reported in Table II and discussed in Section 4.4.1.

RiskDriven: As we discussed in Section 3, compare to TopicDiversity and TextDiversity, risk-driven TCPs have access to some extra knowledge about previous test execution results (pass or fail). Our version of RiskDriven TCP combines the results of all previous executions into the riskiness clusters. Therefore, a TCP should make sure that the tests from the riskier clusters are ranked higher. However, within each cluster there can be several test cases with the same riskiness factor. Thus we need to have a method for ranking the intra-cluster test cases. To do so, one may use any applicable TCP technique within a cluster. Note that we still have the restriction of being black-box, in place; so the options are limited. We also could not use TopicDiversity, in this case, because of the very small sizes of some clusters, which makes the topic extractions meaningless. The two approaches that we use in this experiments for RiskDriven TCP are RiskDrivenRandom and RiskDrivenDiversity (where we used the Random and TextDiversity TCPs, respectively, to rank the intra-cluster test cases). In RQ3, we also use a third risk-driven TCP approach where we take RiskDrivenDiversity but change the priority of the new tests cluster from the lowest to the highest. The reason is that it as much as it makes sense to rerun the very last failing tests first, it also makes sense to run the new tests first. So a comparison between these two approaches is interesting.

4.3.2. TCP evaluation We use the well-known APFD (Average Percentage of Fault-Detection) metric for assessing the effectiveness of a TCP, which is originally introduced by Rothermel et al. in [3]. APFD captures the average of the percentage of faults detected by a prioritized test suite. APFD is given by:

$$APFD = 100 * \left(1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}\right), \quad (2)$$

DOI: 10.1002/stvr

Table II. Sensitivity analysis of distance function and the number of topics (K) in TopicDiversity. Defaults are Manhattan and K=15. +/- means increase/decrease in the median APFD when changing the default setting to the given setting.

Type	Release	Euclidean	K in topicDiversity			
			5	10	25	50
TR	Firefox Desktop3.0	1.29	1.70	-0.20	-0.67	-1.06
	Firefox Desktop3.5	-0.86	-2.73	-4.36	-2.19	-2.95
	Firefox Desktop3.6	-0.35	2.80	0.22	-0.36	-0.50
	Firefox Desktop4.0	-1.68	-1.00	0.21	-0.53	-0.73
RR	Firefox Desktop5.0	1.78	-3.34	-0.48	3.03	0.75
	Firefox Desktop6.0	-9.48	-11.32	-10.41	-7.94	-12.09
	Firefox Desktop7.0	-0.79	-3.56	-7.79	-1.05	-2.94
	Firefox Desktop8.0	1.72	-1.91	2.80	-0.72	-1.05
	Firefox Desktop9.0	6.10	16.02	15.92	6.32	9.52
	Firefox Desktop10.0	3.19	5.08	4.92	-2.21	0.60
	Firefox Desktop11.0	12.83	5.72	9.35	7.99	6.07
	Firefox Desktop12.0	0.22	-3.33	0.90	-8.16	-3.52
	Firefox Desktop13.0	-0.28	-1.68	4.43	-3.06	2.63
	Firefox Mobile16	2.21	0.67	2.1	-2.91	-2.95
	Firefox Mobile17	-0.89	-3.5	-1.59	-1.55	-0.28
	Firefox Mobile18	0.17	0.04	0.76	-1.39	0.52
	Firefox Mobile19	-0.23	-0.73	0.38	2.41	-0.55
	Firefox Mobile20	1.06	1.25	1.75	-3.53	-2.21
	Firefox Mobile21	-2.3	-2.45	-2.97	-2.98	-0.19
	Firefox Mobile22	-1.07	3.6	3.88	-2.1	-0.75
	Firefox Mobile23	1.51	6.66	2.15	-1.86	-1.56
	Firefox Mobile24	0.2	1.5	-1.42	-0.93	-0.11
	Firefox Mobile25	0.73	-0.71	3.48	-0.06	-0.3
	Firefox Mobile26	3.5	-7.76	-1.69	0.27	1.68
	Firefox Mobile27	1.15	0.46	2.2	3.2	3.8
	Firefox Mobile28	0.55	-1.29	2.51	-0.24	-2.09
	Firefox Mobile29	6.88	9.71	-2.04	-1.87	-1.4
	Firefox Tablet16	-2.21	-0.62	-0.05	-2.49	-5.36
	Firefox Tablet17	0.17	-0.74	0.19	-1.73	0.17
	Firefox Tablet18	0.87	-1.87	0.73	2.05	0.5
	Firefox Tablet19	0.49	-1.65	1.02	1.14	0.49
	Firefox Tablet20	-0.7	2.8	2.28	-0.89	-0.86
	Firefox Tablet21	1.47	1.51	0.94	-1.76	1.47
	Firefox Tablet22	0.21	-0.63	-0.22	0.46	-4.53
	Firefox Tablet23	0.98	0.28	-1.23	4.85	0.98
	Firefox Tablet24	-1.08	-0.04	-0.79	0.13	-1.96
	Firefox Tablet25	0.84	-3.57	-0.16	-0.81	0.84
	Firefox Tablet26	0.73	5.19	2.88	-0.42	3.35
	Firefox Tablet27	-1.88	0.98	0.77	-0.34	-1.88
	Firefox Tablet28	-2.03	3.07	0.44	0.61	3.27
	Firefox Tablet29	-0.24	9.9	0.92	-3.52	-0.24

where n denotes the number of test cases, m is the number of faults, and TF_i is the number of tests which must be executed before fault i is detected. As a TCP technique's effectiveness increases (i.e., more faults are detected with fewer test cases), the APFD metric approaches 100.

DOI: 10.1002/stvr

We run each TCPs, on every single release of our case study, 100 times. To compare the APFD values of the different TCP techniques, we apply the non-parametric significant test, Mann-Whitney U test [35], to determine if the difference between the APFD results are statistically significant (p-value below 0.01).

The significant test tells us that the differences are not by chance, but it does not tell us how much one technique outperforms another. To do so, we use a non-parametric *effect size*, Vargha-Delaney A measure [35]. The A measure indicates the probability that one technique will achieve better performance (i.e., higher APFD) than another technique. When the A measure is 0.5, the two techniques are equal. When the A measure is above 0.5, the first techniques outperforms the other, and vice versa. The closer A measure to 0 or 1.0 the higher the differences between the two techniques.

To show the practical differences between the TCPs, per release, we also report the distribution of APFDs for each TCP over the 100 runs, with a boxplot.

4.4. Case study results

In this section, we explain and discuss the results of the experiments under the three RQs. But first let us explain the results of the sensitivity analysis that we ran on some of the parameters of the TopicDiversity based approach.

4.4.1. ***TopicDiversity sensitivity analysis*** Table II details the results based on the design that was explained earlier. The goal of this analysis is to see how sensitive is our TopicDiversity-based approach with respect to changes on the number of topics (K) and the distance function (used for diversification). To do so, we look at the final results in terms of median APFDs over 10 runs of the technique with the given parameter set ups. Looking at the distance function column shows that changing Manhattan to Euclidean improves the median APFD, 25 times in the total of 41 releases. However, if we count the number of times that this improvement is actually practically significant (let's say at least greater than 5% difference), then there is only 3 releases where this change would actually makes the results better. In a similar manner, if we count the number of releases where switching the distance function makes the results more than 5% worse (<-5.0), we will be left with only 1 release. In short, replacing Manhattan with Euclidean does not affect the result of our study, much. Therefore, to be comparable with the earlier publications in this domain, we just use Manhattan in our diversity-based approaches.

Another important parameter in topic modeling is the number of topics that should be predefined (K). We used a range of K between 5 to 50 (5, 10, 15, 25, 50) for our small size corpus. Then we chose the middle value (15) as the default K. The rest of the experiment is similar to the distance function sensitivity analysis, *i.e.,* each column in Table II shows the median APFD differences when K has been changed to 5, 10, 25, and 50.

Looking only at the cases where the APFD differences are >+5.0 or <-5.0 reveals that in (7, 2, 2, and 2) cases APFD is greater when using k=5, 10, 25, or 50 , respectively, rather than k=15 and in (2, 2, 2, and 2) cases APFD is lower, after that change to the number of topics. What this analysis basically says is that the effect of input parameters, in TopicDiversity, are generally not that significant so that it changes the overall discussion of relative differences between the effectiveness of the TCP algorithms that we will see in the RQ results.

4.4.2. ***RQ1 Results*** To answer the question of "*What is the most effective TCP technique for black-box manual testing, in a traditional software development context?*", we look at the effectiveness of the five TCP techniques on the four traditional releases. Table III summarizes the APFD results as the median of 100 runs per TCP technique. The hypothesis is that RiskDriven approaches would outperform RandomTCP, TextDiversity, and TopicDiversity because of the extra knowledge that they have about the previous execution results, specially, RiskDrivenDiversity, since it uses the heuristics from both camps. However, Table III shows that RiskDrivenDiversity is not an obvious dominator. Therefore, we ran a statistical significant test (Mann-Whitney U test) to first make sure the differences between RiskDrivenDiversity and the other TCPs are not by chance. The

DOI: 10.1002/stvr

Table III. Median APFDs (over 100 runs) of the five TCPs and the effect sizes of RiskDrivenDiversity vs. all other four TCPs, in the four traditional releases of Desktop Firefox – Rnd(RandomTCP), TxD(TextDiversity), TpD(TopicDiversity), RDR(RiskDrivenRandom), and RDD(RiskDrivenDiversity

versions	Median APFD					Effect size of RDD vs.			
	Rnd	TxD	TpD	RRD	RDD	Rnd	TxD	TpD	RRD
Firefox 3.0	53.37	61.95	57.25	62.34	67.32	1.00	1.00	1.00	0.99
Firefox 3.5	51.43	51.02	55.52	50.18	49.99	0.35	0.00	0.00	0.44
Firefox 3.6	52.29	59.95	53.06	55.42	57.09	0.89	0.00	0.83	0.94
Firefox 4.0	53.88	54.08	55.82	54.52	54.03	0.54	0.00	0.22	0.31

results are shaded cells in the Table III. Only two pairs of comparisons (RiskDrivenDiversity vs. RiskDrivenRandom in version Firefox 3.5 and RiskDrivenDiversity vs. RandomTCP in version Firefox 4.0) are not significant. Knowing that the differences are not by chance, we finally look at the effect size measure. Table III also shows the pair comparisons of effect size (A measures) between RiskDrivenDiversity and all the other four TCPs. As we explained, A measures less than 0.5 means that RiskDrivenDiversity is likely to perform worse than the compared with TCP, which is not uncommon based on Table III (all the other four techniques at least once outperform RiskDrivenDiversity).

To summarize all of these statistical comparisons, from a practical point of view, Figure 3 shows the distribution of the five TCPs' APFD, as boxplots. The most clear message that the figure conveys is that there is no common pattern between the four releases. Sometimes RiskDriven approaches perform better and sometimes TopicDiversity or TextDiversity is the better TCP. In fact, all TCPs including RiskDrivenDiversity and even RandomTCP perform quite in the same range. Therefore, RQ1 does not have an easy answer. The only technique that outperforms all others in more than one release out of four, is TopicDiversity, which, unfortunately, has a high variance and is the worst among not Random TCPs in Firefox 3.6, which makes it unreliable.

So the best answer to RQ1 is that there is no best TCP among those that we have tried; from risk-driven approaches to information retrieval based techniques. The ineffectiveness of these TCPs may be partly due to the nature of our problem (prioritizing manual test cases) or due to the characteristics of these four releases (traditional long release cycles). To study this further we look at RQ2.

4.4.3. ***RQ2 Results*** In RQ2 ("*Does the relative effectiveness of the evaluated TCP techniques from RQ1 change, when the development moves toward rapid-releases?*"), we will analyze the same five TCPs but on nine more recent releases of Desktop Firefox, where the development environment follows rapid release policies. The fact that the releases are more often, which results in more test executions, is even more interesting from the perspective of TCP techniques. The larger test suites and the more test executions per unit of time, the higher need for more effective test prioritization techniques. Therefore, we specifically analyze the TCPs in the rapid releases vs. traditional releases.

Table IV summarizes the APFD results as the median of 100 runs per TCP technique. The first observation from the table is that unlike RQ1, both RiskDriven approaches are by far more effective than the other three TCPs. On average, the median APFD of the best RiskDriven TCP improves the median APFD of the best of the other three TCPs by 77% (it ranges between 34% to 105% in the nine releases). The differences are also statistically significant and the effect size measure is always 1.0, when comparing, e.g., RiskDrivenDiversity with the RandomTCP, TopicDiversity, and TextDiversity.

One plausible explanation is that in rapid release the modifications on each release are very limited, which makes the number of faults per release very small, compared to the traditional releases, as it is seen in Table I. Therefore, in traditional release many of the defects are from completely new parts of the code that the older test cases can not detect them. However, the older test suites in rapid release are still quite good, for the next release, due to the limited change in the code. As we said, this is just a hypothesis and further research is required to validate it.

DOI: 10.1002/stvr

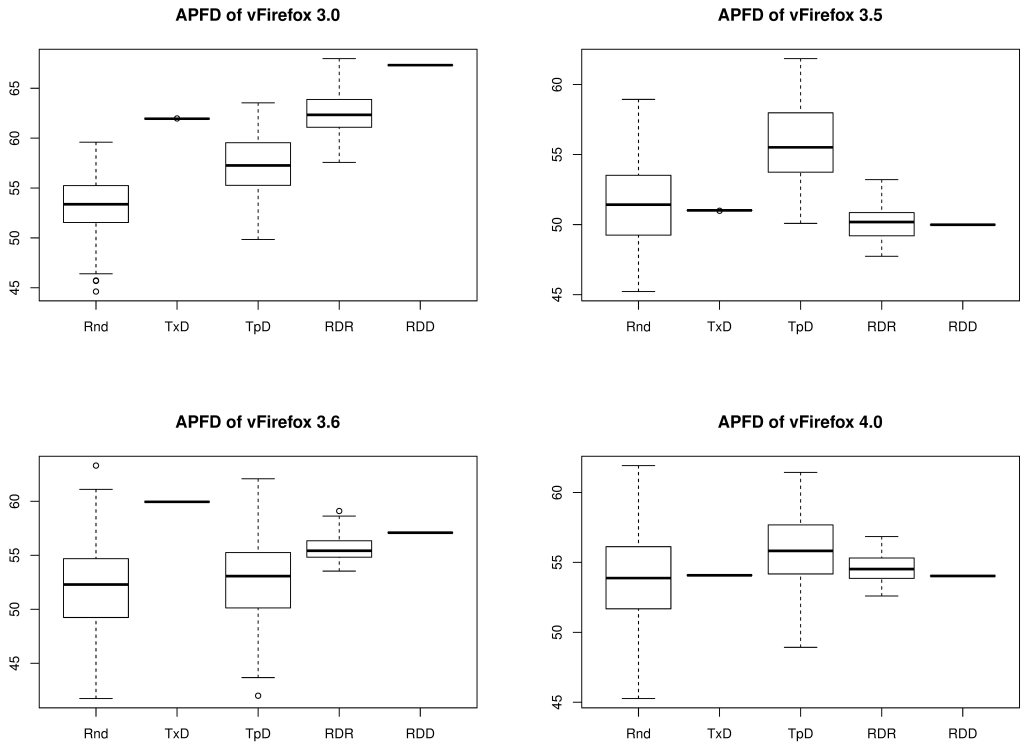

Figure 3. Distribution of the APFDs of the five TCPs on four traditional releases of Desktop Firefox, over 100 runs, as boxplots.

To study RiskDriven approaches, in more details, we look at the distribution of the results in the boxplots shown in Figure 4. Looking at Figure 4, we can see that RiskDriven approaches not only show higher effectiveness in terms of APFD, but also less variance (only when comparing with Rnd and TpD) in the results, which increases the reliability of the TCP technique to be used in other releases and potentially other systems.

The second observation is the close to 100% APFDs that the RiskDriven approaches constantly show in the nine releases. These features make them a perfect candidate for TCP in rapid release environments. Finally, to choose one among the two, as the best, we should go for RiskDrivenDiversity. Though the Table IV suggests otherwise, looking at the Figure 4 reveals that the poor effect size does not practically matter in most cases (e.g., in Firefox 7.0), since the actual difference is not practically significant. However, RiskDrivenDiversity shows less variance compared to RiskDrivenRandom, which can be a deciding factor. Therefore, though both RiskDrivenDiversity and RiskDrivenRandom are highly effective and very close in terms of median APFD, we select RiskDrivenDiversity as the most effective and reliable TCP for the rapid release environments.

4.4.4. ***RQ3 Results*** After identifying RiskDrivenDiversity as the best TCP for rapid release environments, in RQ2, in RQ3, we would like to know whether such high performance is consistent across other projects. Therefore, we run the same experiment as RQ1 and 2 on two new projects from Mozilla Firefox (Mobile Firefox and Tablet Firefox), 14 releases each. Table V shows the APFD results along with the effect size measures for the Mobile Firefox project. Again the differences between RDD and RRD is minor so we focus on RDD vs. the other three TCPs. Overall, among the 14 releases, in 12 releases RDD performs better than any of the Rnd, TxD, and TpD. Only in two

DOI: 10.1002/stvr

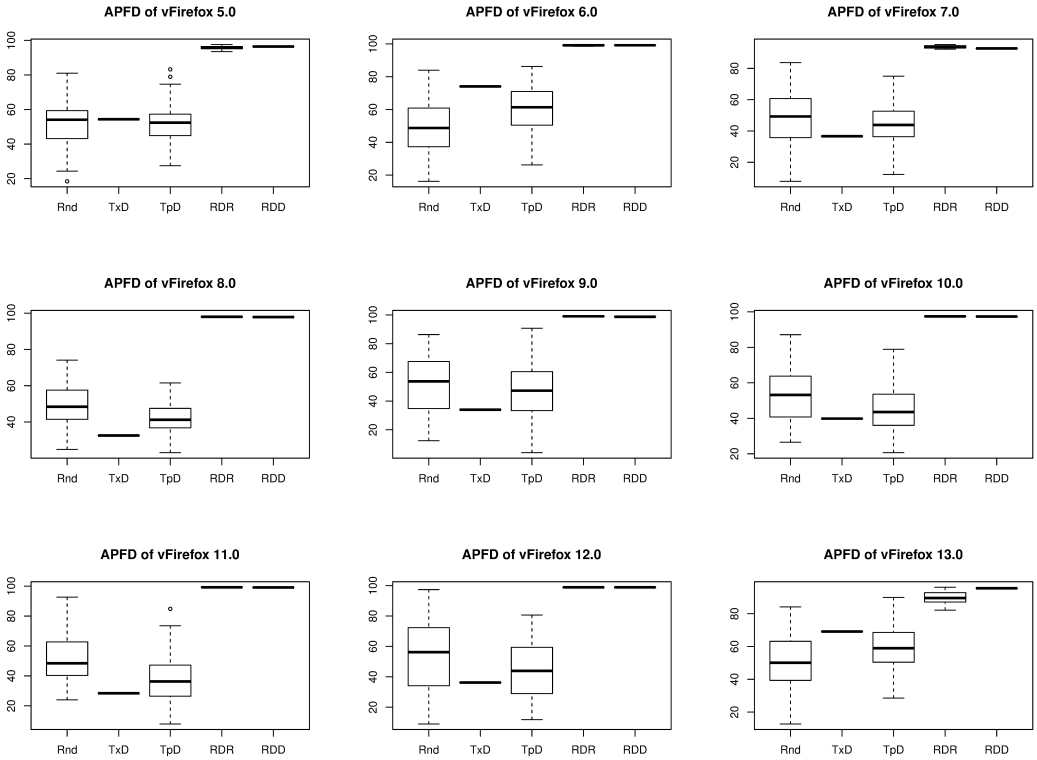

Figure 4. Distribution of the APFDs of the five TCPs on nine rapid releases of Desktop Firefox, over 100 runs, as boxplots.

Table IV. Median APFDs (over 100 runs) of the five TCPs and the effect sizes of RiskDrivenDiversity vs. all other four TCPs, in the nine Rapid releases of Desktop Firefox.

versions	Median APFD					Effect size of RDD vs.			
	Rnd	TxD	TpD	RRD	RDD	Rnd	TxD	TpD	RRD
Firefox 5.0	54.09	54.39	52.39	95.90	96.45	1.00	1.00	1.00	0.76
Firefox 6.0	48.73	74.08	61.38	99.15	99.21	1.00	1.00	1.00	0.57
Firefox 7.0	49.27	36.64	43.85	93.53	92.78	1.00	1.00	1.00	0.15
Firefox 8.0	48.39	32.53	41.19	98.02	97.88	1.00	1.00	1.00	0.31
Firefox 9.0	53.70	33.99	47.24	99.07	98.73	1.00	1.00	1.00	0.03
Firefox 10.0	53.16	39.85	43.56	97.48	97.38	1.00	1.00	1.00	0.21
Firefox 11.0	48.47	28.39	36.38	99.21	99.14	1.00	1.00	1.00	0.06
Firefox 12.0	56.23	36.24	43.92	98.92	98.89	1.00	1.00	1.00	0.33
Firefox 13.0	50.08	69.12	58.92	89.55	95.50	1.00	1.00	1.00	0.98

releases, V19 and V28, TextDiversity outperforms RDD. However, the APFD difference between them in V28 is really minor (0.65%) and TxD shows an slightly more variance.

Looking at Figure 5 we also observe that the RDD's APFD is not as close to maximum as it was in the rapid releases of the Desktop version. One potential reason is that the mobile project is newer and less stable than the Desktop project. Therefore, as we see in Table I, the failure rate is higher in the mobile compared to rapid Desktop releases. What a high failure rate may mean is that the failing tests are not just a few from the most recent releases. But they are spread across all clusters and thus they are not all prioritized up front by RDD. Nevertheless, RDD still provides a very high APFD (median 78% across all the 14 releases). Therefore, with quite high confidence, we can say that

DOI: 10.1002/stvr

Table V. Median APFDs (over 100 runs) of the five TCPs and the effect sizes of RiskDrivenDiversity vs. all other four TCPs, in the 14 Rapid releases of Mobile Firefox.

versions	Median APFD					Effect size of RDD vs.			
	Rnd	TxD	TpD	RRD	RDD	Rnd	TxD	TpD	RRD
Mobile Firefox v16	53.83	60.73	56.34	76.88	77.66	1.00	1.00	1.00	0.72
Mobile Firefox v17	55.25	56.94	55.84	79.37	79.01	1.00	1.00	1.00	0.34
Mobile Firefox v18	55.46	59.51	56.16	78.08	78.65	1.00	1.00	1.00	0.73
Mobile Firefox v19	55.24	54.93	55.80	47.61	47.29	0.01	0.00	1.00	0.44
Mobile Firefox v20	58.45	60.93	58.13	80.29	80.82	1.00	1.00	1.00	0.69
Mobile Firefox v21	54.98	57.03	57.88	84.14	84.99	1.00	1.00	1.00	0.86
Mobile Firefox v22	56.77	65.62	59.49	77.40	77.60	1.00	1.00	1.00	0.60
Mobile Firefox v23	59.85	58.61	53.55	64.65	65.22	0.94	1.00	1.00	0.81
Mobile Firefox v24	55.02	57.56	54.78	80.31	80.57	1 .00	1.00	1.00	0.67
Mobile Firefox v25	56.47	62.39	60.59	81.55	82.46	1.00	1.00	1.00	0.86
Mobile Firefox v26	52.58	53.34	58.16	64.93	65.72	0.98	1.00	0.95	0.66
Mobile Firefox v27	56.05	46.33	54.16	58.76	57.80	0.67	1.00	0.87	0.21
Mobile Firefox v28	52.80	56.34	52.72	51.40	55.69	0.65	0.05	0.71	0.98
Mobile Firefox v29	55.16	58.99	56.43	59.07	60.62	0.77	1.00	0.78	0.72

RDD is the most effective TCP among the five studied techniques in these Mobile Firefox releases as well.

We also repeat the same analysis on the Tablet Firefox project. Table VI and Figure 6 show the same type of results for the 14 versions of Tablet Firefox project. The results are quite consistent with the Mobile Firefox. In three releases (v27, v28, and v29) out of 14, RDD is outperformed by TxD or TpD, but only in one releases (v27) the difference is higher that 5%. The only down side of RDD is that it is not effective in the last three releases of tablet. However, the median APFD in tablet releases is again 78%, which is quite high and inline with the Mobile Firefox results.

- **Improving RDD**

To improve the effectiveness of RDD, specially in cases like Mobile Firefox v19, we have modified RDD so that it prioritizes the new test cases first. We call this approach New Risk Driven Diversity (NRDD). The change basically assumes that the new tests are more effective in finding bugs than the old tests. This effectiveness partly depends on the number of old vs. new tests as well. If there are many new tests and a few old tests, it worths running the old tests first and vice versa. We run NRDD on all the 41 releases 100 times and compare the APFDs with those of RDD. The results which are shown in Figures 7, 8, 9, 10, clearly show that there is no best approach that works best for al releases. In cases like v19 of Firefox Mobile putting the new clusters first helps a lot but there are several other cases where the effect is vice versa. However, no matter what strategy one chooses for ranking the new tests cluster, the overall results across all releases are as good as it can be. In other words, if we have no inside information about the number and historical effectiveness of the new tests it does not really matter whether we rank them first or last.

4.4.5. ***Threat to the validity*** In terms of internal and construct validity, we have reduced the potential threats by building our system upon existing tools (tcp.lda) and measures (APFD). The only algorithm that we build from scratch is RiskDriven, which has carefully explained in the paper and is easy to implement, with minimum tuning required. The other technique (TopicDiversity) have been analyied with respect to its major parameters, to reduce the threat of being biased toward a specific topic size or distance function. Regarding the conclusion validity, since all TCPs studied here are randomized (sources of randomness are e.g., the initial selection, topic extraction procedure, random selections in case of ties, etc.), we have carefully studied the distribution of results using the TCPs by 100 time running each technique and reporting statistical significance tests and effect sizes. In addition, we have looked at the practical differences between results by plotting the entire

DOI: 10.1002/stvr

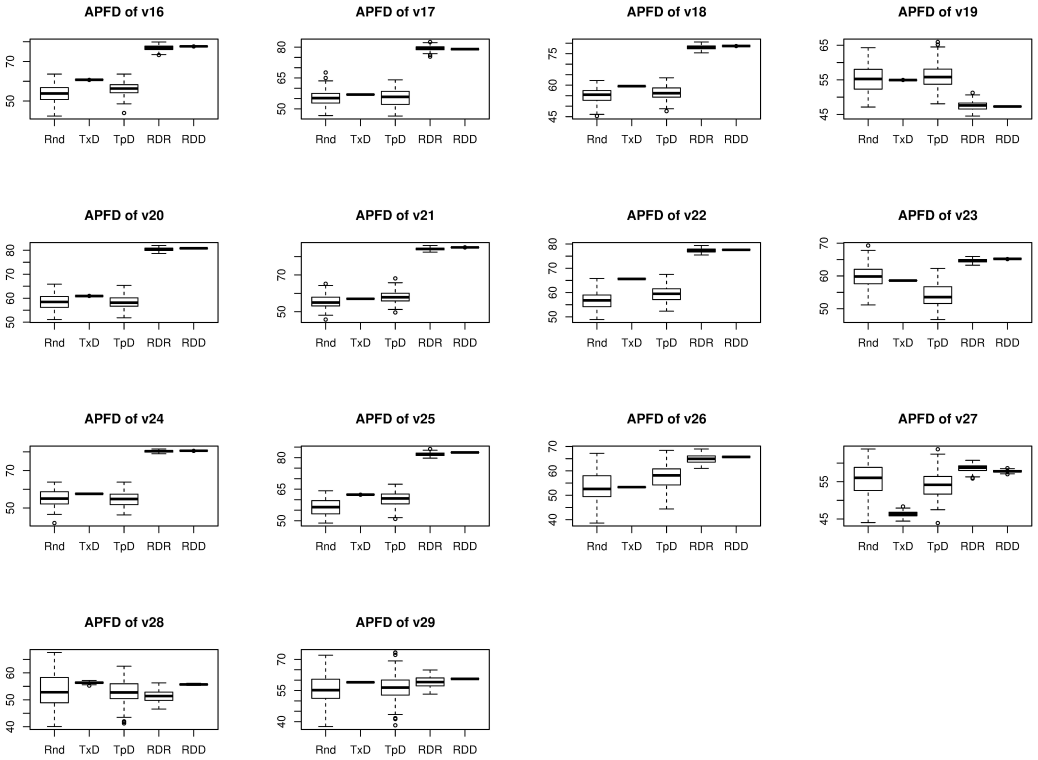

Figure 5. Distribution of the APFDs of the five TCPs on 14 rapid releases of Mobile Firefox, over 100 runs, as boxplots.

Table VI. Median APFDs (over 100 runs) of the five TCPs and the effect sizes of RiskDrivenDiversity vs. all other four TCPs, in the 14 Rapid releases of Tablet Firefox.

versions	Median APFD					Effect size of RDD vs.			
	Rnd	TxD	TpD	RRD	RDD	Rnd	TxD	TpD	RRD
Tablet Firefox 16	55.32	59.61	58.61	76.54	77.50	1.00	1.00	1.00	0.77
Tablet Firefox 17	54.81	60.49	57.05	80.35	82.07	1.00	1.00	1.00	0.98
Tablet Firefox 18	54.87	55.68	57.18	77.50	76.13	1.00	1.00	1.00	0.10
Tablet Firefox 19	54.80	52.90	55.32	58.46	59.05	0.88	1.00	0.87	0.66
Tablet Firefox 20	55.46	60.92	56.80	81.57	83.09	1.00	1.00	1.00	0.93
Tablet Firefox 21	56.25	64.33	57.26	84.49	85.48	1.00	1.00	1.00	0.94
Tablet Firefox 22	55.07	63.63	54.79	77.10	78.08	1.00	1.00	1.00	0.92
Tablet Firefox 23	56.95	64.25	57.44	78.08	78.77	1.00	1.00	1.00	0.71
Tablet Firefox 24	52.64	55.90	56.56	85.85	85.56	1.00	1.00	1.00	0.39
Tablet Firefox 25	55.72	54.24	56.95	83.46	83.88	1.00	1.00	1.00	0.78
Tablet Firefox 26	53.93	45.44	55.64	62.97	61.40	0.96	1.00	0.92	0.12
Tablet Firefox 27	53.56	53.99	50.20	48.03	47.52	0.09	0.00	0.87	0.36
Tablet Firefox 28	50.98	44.28	52.42	49.76	47.56	0.22	1.00	0.11	0.18
Tablet Firefox 29	51.03	56.35	53.68	51.51	52.68	0.63	0.00	0.44	0.67

distributions by boxplots and discussing the results. Finally, with respect to external validity, we should emphasis that this paper reports a case study on 41 releases of Mozilla Firefox. We do not know how much the results are specific to Mozilla Firefox and believe that replicating this study on other projects in the similar contexts is required, before robust conclusions can be made.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

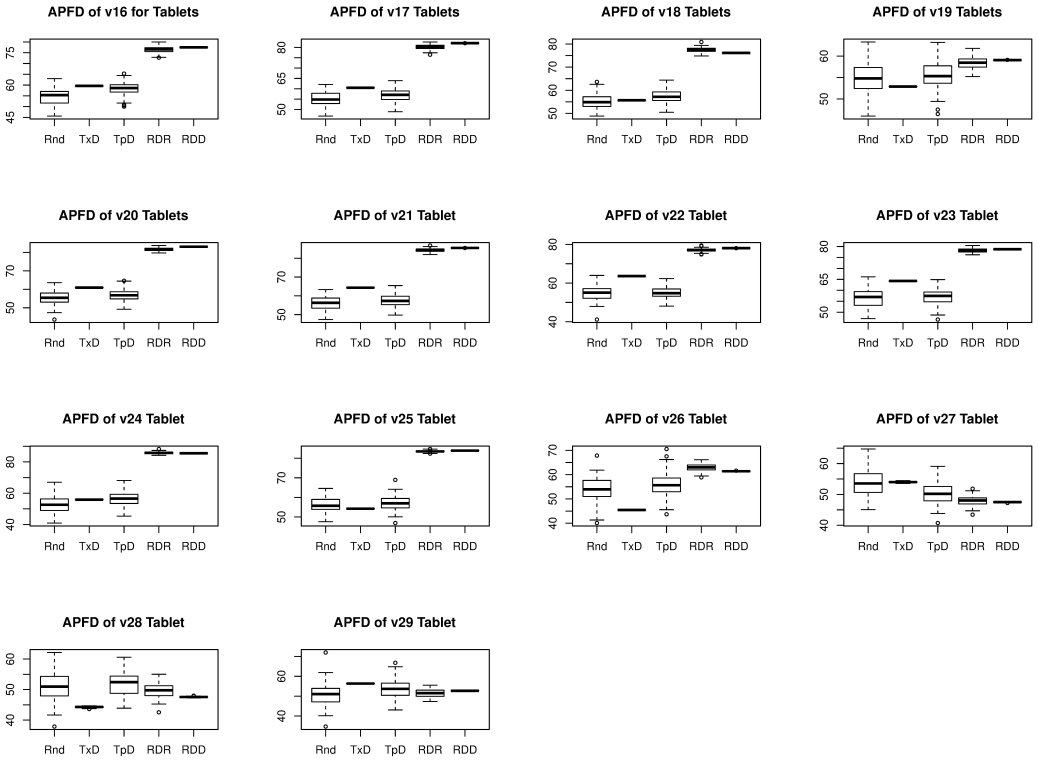

Figure 6. Distribution of the APFDs of the five TCPs on 14 rapid releases of Tablet Firefox, over 100 runs, as boxplots.

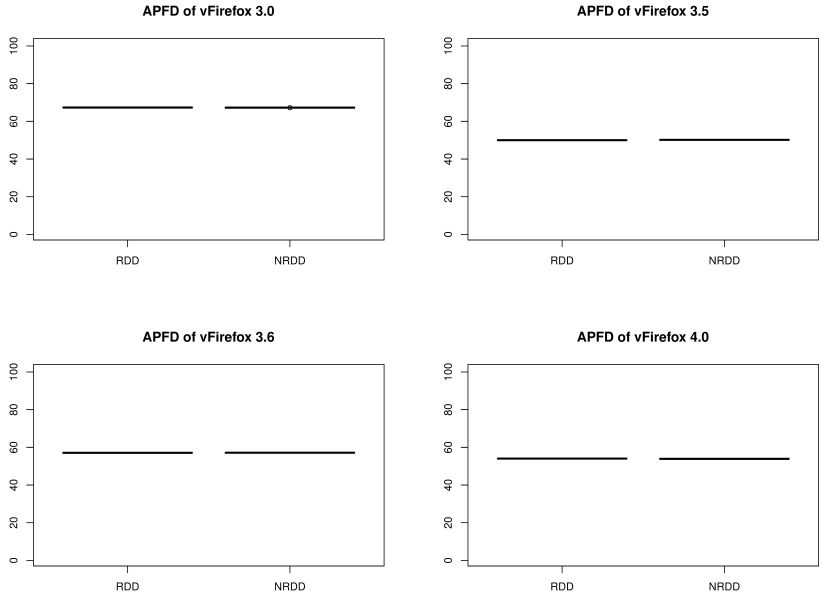

Figure 7. Distribution of the APFDs of RDD vs NRDD on 4 traditional releases of Desktop Firefox, over 100 runs, as boxplots.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

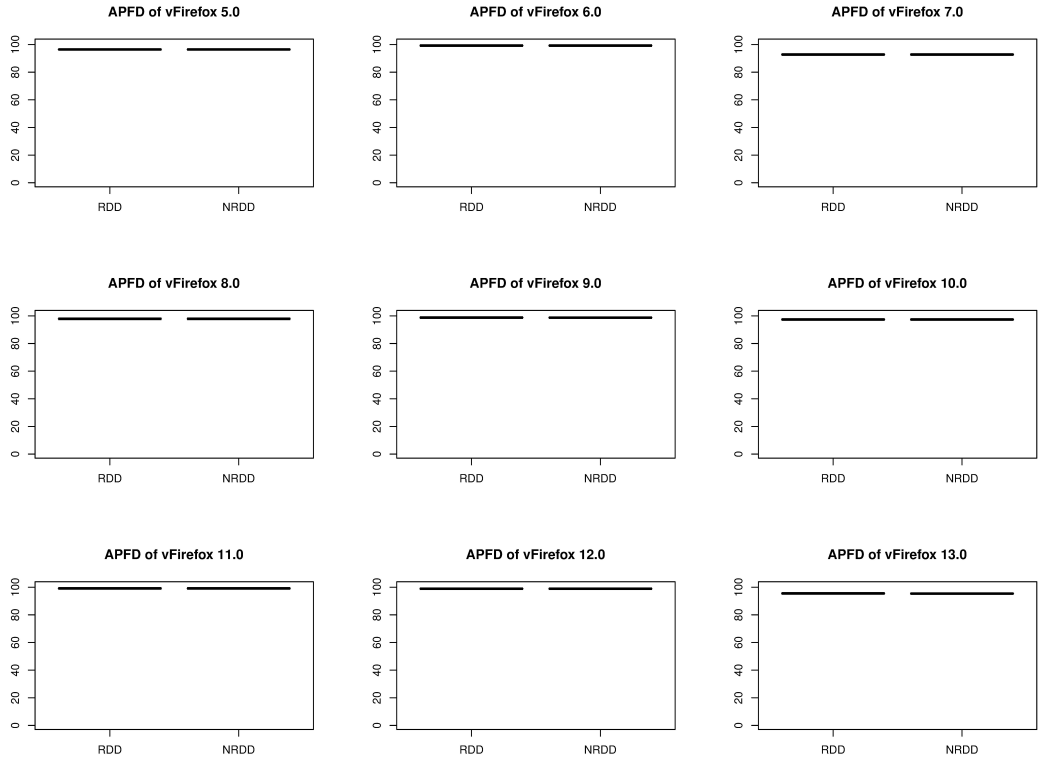

Figure 8. Distribution of the APFDs of RDD vs NRDD on 9 rapid releases of Desktop Firefox, over 100 runs, as boxplots.

5. ACKNOWLEDGEMENT

Zhihan Fang was supported by Mitacs Globalink program at University of Manitoba. Mika V. Mäntylä collected the data when he was at Lund University and he was at Aalto University at the time of conducting the study.

6. CONCLUSION

Continuos delivery and rapid release are becoming very common in software industry due to several reasons such as faster time-to-market and frequent user feedback. Keeping the high quality in this fast paced environment requires a lot of testing before release. However, the massive test suites of large scale systems are infeasible to be fully re-tested after every single change. Therefore, in the context of regression testing, it is crucial to identify effective test prioritization techniques that maximize the fault detection power of test cases, for the given testing budget.

In this paper, we targeted a specific type of testing, manual black-box system testing, which is a common testing in practice. The challenge is that such test cases usually written in a natural language and explain the steps to take in GUIs. Therefore, unlike usual unit/integration tests, they do not reveal useful information for a typical test prioritization technique. Thus we proposed three prioritization techniques and compared their effectiveness in the context of system testing at Mozilla Firefox.

Our experiments showed that none of the topic-based, text-based, and risk-driven approaches are highly dominating the others, in the context of traditional software development (version 2.0 to 4.0 of Mozilla Firefox). However, the risk-driven approach is by far more effective than the others,

DOI: 10.1002/stvr

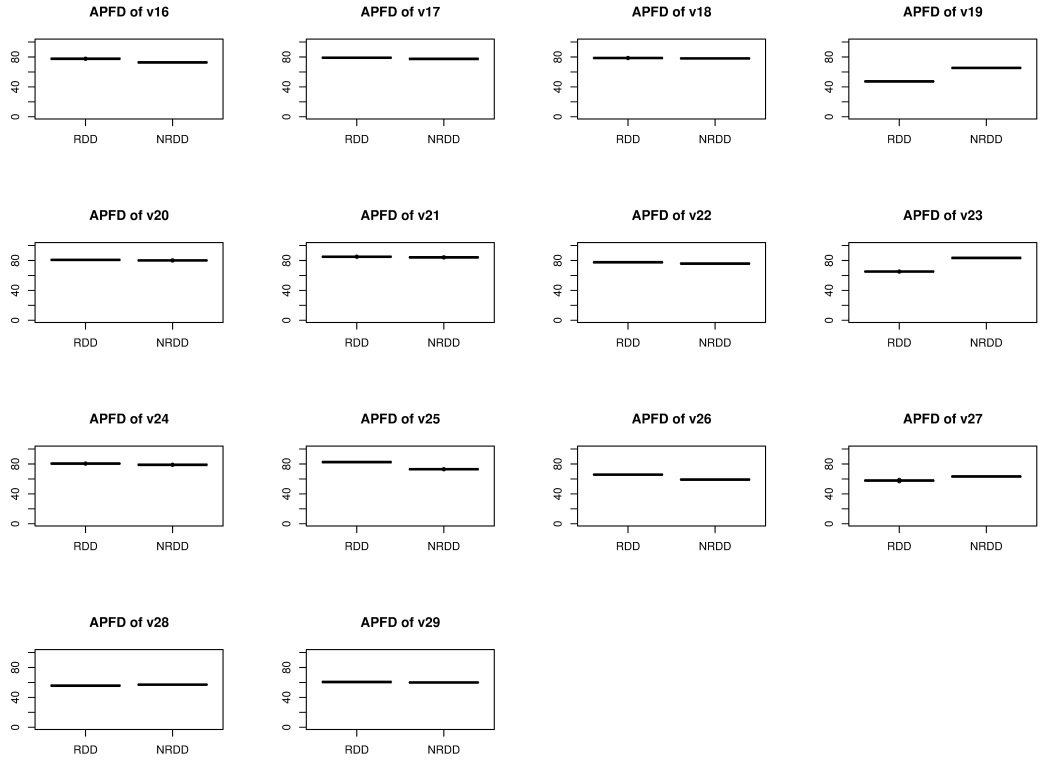

Figure 9. Distribution of the APFDs of RDD vs NRDD on 14 rapid releases of Mobile Firefox, over 100 runs, as boxplots.

in the context of rapid release (versions 5.0 to 13.0 of Mozilla Firefox and the Mobile and Tablet versions). The results of the risk-driven test prioritization approach for rapid releases, in many cases, are also very close to the optimum values, which makes the findings very interesting. In the future, we plan to replicate the study on other software systems and examine the rationales behind the better results of risk-driven approach, in more details. We also plan to propose different variations of text-mining-based approach (TopicDiversity) and compare them with the basic one, proposed in this paper.

REFERENCES

1. Bertolino A. Software testing research: Achievements, challenges, dreams. *2007 Future of Software Engineering*, IEEE Computer Society, 2007; 85–103.
2. Humble J, Farley D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st edn., Addison-Wesley Professional, 2010.
3. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on* 2001; **27**(10):929–948.
4. Arafeen M, Do H. Test case prioritization using requirements-based clustering. *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, 2013; 312–321.
5. Ledru Y, Petrenko A, Boroday S, Mandran N. Prioritizing test cases with string distances. *Automated Software Engineering* 2011; **19**(1):65–95.
6. Thomas SW, Hemmati H, Hassan AE, Blostein D. Static test case prioritization using topic models. *Empirical Software Engineering* 2014; **19**(1):182–212.
7. Kim JM, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, IEEE, 2002; 119–129.
8. Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

DOI: 10.1002/stvr

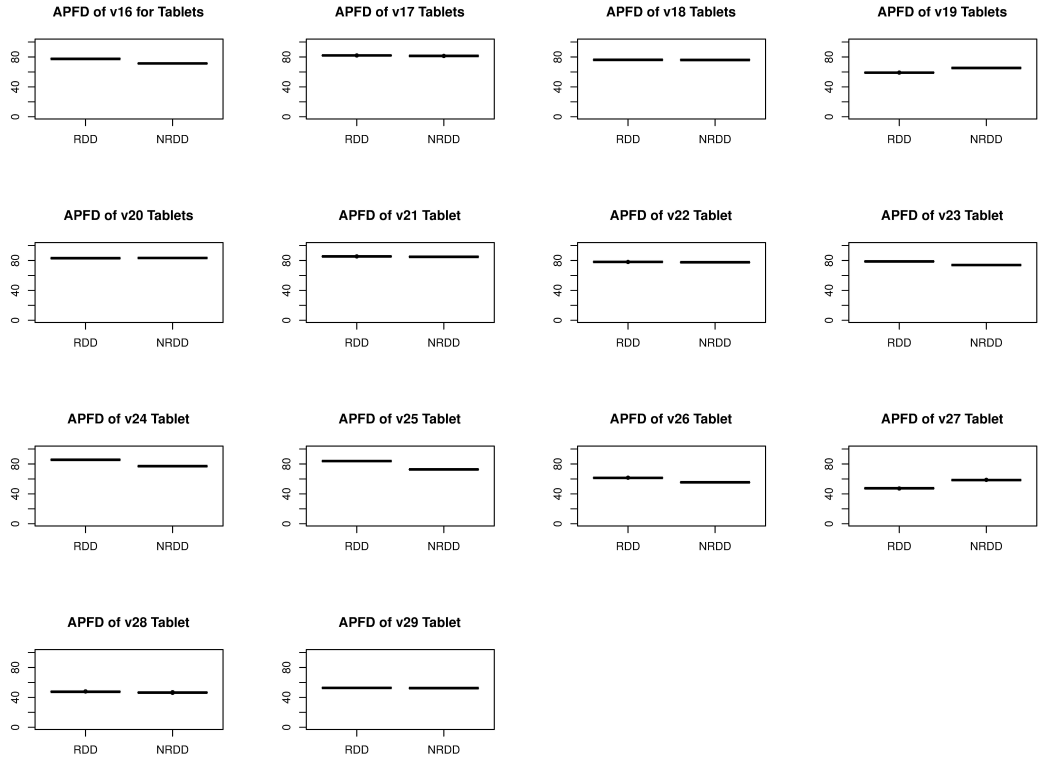

Figure 10. Distribution of the APFDs of RDD vs NRDD on 14 rapid releases of Tablet Firefox, over 100 runs, as boxplots.

9. Hemmati H, Fang Z, Mantyla MV. Prioritizing manual test cases in traditional and rapid release environments. *International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2015; 1–10.
10. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Software,Testing, Verification, and Reliability* 2012; **22**(2):67–120, doi:10.1002/stv.430.
11. Orso A, Apiwattanapong T, Harrold MJ. Leveraging field data for impact analysis and regression testing. *ACM SIGSOFT Software Engineering Notes*, vol. 28, ACM, 2003; 128–137.
12. Briand LC, Labiche Y, Soccar G. Automating impact analysis and regression test selection based on uml designs. *Software Maintenance, 2002. Proceedings. International Conference on*, IEEE, 2002; 252–261.
13. Onoma AK, Tsai WT, Poonawala M, Suganuma H. Regression testing in an industrial environment. *Communications of the ACM* 1998; **41**(5):81–86.
14. Elbaum S, Malishevsky A, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, 2001; 329–338.
15. Mei H, Hao D, Zhang L, Zhang L, Zhou J, Rothermel G. A static approach to prioritizing JUnit test cases. *IEEE Transactions on Software Engineering* 2011; .
16. Hemmati H, Arcuri A, Briand L. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2013; **22**(1):6.
17. Hemmati H, Briand L, Arcuri A, Ali S. An enhanced test case selection approach for model-based testing: an industrial case study. *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ACM, 2010; 267–276.
18. Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 2002; 1.
19. Krishnamoorthi R, Mary SSA. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology* 2009; **51**(4):799 – 808.
20. Srikanth H, Williams L, Osborne J. System test case prioritization of new and regression test cases. *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005.
21. Yoo S, Harman M, Tonella P, Susi A. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009; 201–212.
22. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *Software Engineering, IEEE Transactions on* 2007; **33**(4):225–237.

DOI: 10.1002/stvr

23. Jiang B, Zhang Z, Chan WK, Tse T. Adaptive random test case prioritization. *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, IEEE, 2009; 233–244.
24. Bosch J. Driving innovation through software experiment systems. Keynote talk at International Symposium on Empirical Software Engineering and Measurement 2011.
25. Jenkins J. Velocity culture (the unmet challenge in ops). Presentation at O'Reilly Velocity Conference June 2011.
26. Mantyla MV, Adams B, Khomh F, Engstrom E, Petersen K. On rapid releases and software testing: A case study and a semi-systematic literature review. *Empirical Software Engineering* In press; :1–41.
27. Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, *et al.*. Manifesto for agile software development 2007.
28. Raymond ES. *The Cathedral and the Bazaar*. 1st edn., O'Reilly & Associates, Inc.: Sebastopol, CA, USA, 1999.
29. Khomh F, Dhaliwal T, Zou Y, Adams B. Do faster releases improve software quality? an empirical case study of mozilla firefox. *MSR*, 2012; 179–188.
30. Baysal O, Kononenko O, Holmes R, Godfrey MW. The secret life of patches: A firefox case study. *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012; 447–455.
31. Hemmati H, Arcuri A, Briand L. Empirical investigation of the effects of test suite properties on similarity-based test case selection. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, IEEE, 2011; 327–336.
32. Bolelli L, Ertekin , Giles C. Topic and trend detection in text collections using latent dirichlet allocation. *Advances in Information Retrieval* 2009; :776–780.
33. Marcus A. Semantic driven program analysis. *Proceedings of the 20th International Conference on Software Maintenance*, 2004; 469–473.
34. Thomas SW. Topic coverage test prioritization tool 2014. URL https://github.com/stepthom/tcp.lda.
35. Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. *Software Engineering (ICSE), 2011 33rd International Conference on*, IEEE, 2011; 1–10.

DOI: 10.1002/stvr