Prioritizing Manual Test Cases in Traditional and Rapid Release Environments

Hadi Hemmati
Department of Computer Science
University of Manitoba
Canada
hemmati@cs.umanitoba.ca

Zhihan Fang
School of Software Engineering
Tongji University
China
ezhihan@gmail.com

Mika V. Mäntylä
Department of Computer Science & Engineering
Aalto University
Finland
mika.mantyla@aalto.fi

***Abstract*—Test suite prioritization is one of the most practically useful activities in testing, 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 suite prioritization techniques are based on code coverage, which requires access to source code. However, system tests are mainly black-box. Therefore, in this paper, we first examine the existing test suite prioritization techniques and modify them to be applicable on *manual black-box system testing*. We specifically study a *coverage-based*, a *diversity-based*, and a *risk driven* approach for test suite prioritization. Our empirical study on four older releases of Mozilla Firefox shows that non of the techniques are strongly dominating the others in all releases. However, when we study nine more recent releases of 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 65% effectiveness improvement) between the risk-driven approach and its alternatives. Our conclusion, based on one case study of 13 releases of an industrial system, is that test suites in rapid release environments, potentially, can be very effectively prioritized for execution, based on their historical riskiness; whereas the same conclusions do not hold in the traditional software development environments.**

I. 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 [2] and continuos integration [3]. 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 [2]. However, rerunning all tests on large scale systems is not possible, even if it is scheduled once a day as a nightly integration build [3]. 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 suite so that depending on the time constraints of the build, only the most important tests be executed.

Test suite 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 suite. Among them coverage-based heuristics (prioritizing tests with higher code coverage, *e.g.*, statement coverage) have been very popular [4]. 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 system-level testing, where the testers only have access to the system as a black-box. The situation is worse in manual system-level testing, where testing is mostly done through the system's graphical user interface, rather than calling source code methods in automated tests. Since such test scripts do not reveal even the APIs of the source code, which could be used for test prioritization.

In this paper, we first examine the existing heuristics that can potentially be applied on manual black-box system-level testing. The techniques under study fall under three heuristics: a) covering maximum topics of the test cases. Topics are extracted from the test cases by a text mining algorithm, b) diversifying test cases using a distance function applied on the textual data of the test cases, and c) clustering test cases based on their riskiness, which is determined based on whether they have been detecting faults in the previous releases or not. We then implement and exercise these techniques on four releases of Mozilla Firefox, which are developed by a traditional development approach, *i.e.*, yearly releases. The results show that non 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 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 65% more effective in terms of the APFD measure [4], which will be defined in Section IV). 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.

Th rest of this paper is organized as follows: the next section, Section II, provides a background and related work on test prioritization techniques and the rapid release. Section III

describes our proposed and implemented approaches for test prioritization and Section IV explains the design and results of the case study and finally, Section V concludes the paper.

II. 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.

A. Test suite 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 suite prioritization. Test suite reduction [5] usually removes redundant test cases from a test suite and test case selection [5] selects the most fault-revealing tests based on a given heuristic. However, test suite prioritization (TSP) [5] focuses on ranking all existing tests, without eliminating any test case. In other words, when a test suite is prioritized, one executes the test cases in the given order, until the testing budget is over. Thus a TSP's goal is to *optimize the overall fault detection rate of the executed test cases, for any given budget*.

There are several TSP techniques that one can apply on a given test suite [5]. Each technique may have access to different types of information and uses different heuristics to achieve the TSP's goal. In the rest of this section, we summarize the key TSP techniques, based on their inputs and heuristics.

1) Input resources for TSPs: The main input resources available for a TSP technique are as follows:

Change information: A TSP 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 TSP may then prioritize test cases that execute the affected parts over the other tests. The emphasis of these approaches is on change impact analysis [6], [7].

Historical fault detection information: TSPs may use fault detection information of each test case on the previous versions of the software, as a basis to identify its riskiness [8], [9]. 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 [10].

Dynamic and static coverage data: Another common resource that is being used in TSPs is the code coverage of each test case [4], [5]. 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. Therefore, the dynamic coverage data can only be used, if they are already available from the previous executions. Note that executing test cases to calculate the coverage is not an option for TSPs due to the nature of the problem (the limited budget). 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, TSP techniques should 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 [11].

Specification models: In model-based testing, TSPs 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 by paths in the state machine. A TSP, in this example, would prioritize test cases knowing which paths in the model are executed by which test cases [12], [13], [14].

Test scripts: Finally, there are a few TSP techniques that only look at the test scripts as a source of information. These TSPs are usually applicable in a wider range of domains, *e.g.*, black-box system testing, where the other mentioned sources of data may not be available. For instance, in [15] the authors derive a topic model from the test scripts, which approximates the features that each test case covers. Then the objective of the TSP is maximizing the topic coverage. There are also cases where the test scripts are taken as strings of words, without any extracted knowledge attached to them. The TSPs in this case may have different objective (diversifying tests), which we explain in the next subsection.

Availability of any of these resources is very context dependent. However, in general, testing type directly affects the input resources. For instance, in the case of white-box unit testing, TSPs usually have access to both test code and the source code [4], but in black-box testing the TSP only have access to the test code [11], [15]. Finally, the test level can also play a role in the availability of resources. For example, system-level test cases are black box and very high level. They are either test scripts that call different APIs or sometimes they are some step-by-step instructions for manual testers to follow on GUIs. Prioritizing system level tests, specially the later case is much harder due to the lack of resources to build our heuristics upon.

2) Objectives of TSPs: So far we reviewed some of the most common input sources that are available for TSPs. Given an input, a TSP uses a heuristic to optimize its objective. Two common heuristics from the literature are as follows:

Maximizing coverage: Since coverage info is one of the most used resources for TSPs, 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 [4] by a greedy algorithm. Maximizing coverage has also been done by clustering [16] or evolutionary search algorithms [17].

Diversifying test cases: More recently, researchers have also proposed another heuristic, diversity-based TSP, which tries to spread the testing budget evenly across different parts

of the code [18]. The hypothesis is that similar test cases detect the same faults and thus we should exercise more diverse test cases to detect more faults [19]. Diversification of test cases can be applied on different levels, *e.g.*, method calls, extracted topics of the test cases, and the text of the test scripts.

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

B. 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" [20]. Companies offering web-based services have taken this to the extreme, e.g., Amazon deploys software on every 11.6 seconds, on average [21]. However, the desire to increase speed in software development is not limited to companies operating in web services. According to our recent literature review [22], 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 [22]. 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" [23]. Similarly open source software development recognizes the importance of rapid releases with a well-known slogan from Raymond's book: [24] "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 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 project transition to rapid releases. This change has been studied in the past. Mantyla et al. [22] 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 niether produced a significant decline in the quality of the Firefox browser [25] nor did change the source code patch life cycle [26]. In this paper, we will look at the differences between rapid and traditional releases, in the context of test suite prioritization.

III. Black-box System-level test prioritization

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

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.

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

A. Topic coverage-based

As discussed in Section II, code coverage-based TSPs are the most common TSPs, in the literature and in practice. However, our black-box systems tests written in a natural language do not provide any information about the code coverage. Though the usual code coverage measures (e.g., statement coverage) can not be directly calculated, given our test cases, we have modified a TSP technique that provides us an alternative concept to cover. The concept is called *topic* and the TSP technique is based on *topic coverage*. The idea behind our topic coverage-based TSP is that *if you do not test a topic you won't find defects related to that topic*. In [15], the authors proposed a black-box topic-coverage-based TSP that uses a topic modeling technique called Latent Dirichlet Allocation (LDA) [27] 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 rank tests so that they cover more topics sooner. In this paper, we use a very similar approach but instead of applying the topic modelling algorithm on the linguistic data of the test scripts, which is not available in manual test cases written in a natural language, we apply it on the English text of the test instructions. To understand it 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 Firefox browser.

One way of summarizing the textual information in the test cases is using a topic modeling technique like LDA [27]. In general, LDA, works in two steps [15]: "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)" [15].

"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" [15].

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 cover more of uncovered topics. The exact algorithm for maximizing the topic coverage 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-coverage-based TSP is to rank the *n* test cases for execution, in a way that they cover as much topics as possible, sooner in the text execution phase. To do so, we define two variables: 1) a test suite coverage vector (TSCV), which represents the maximum coverage per topic, so far, for a given set of test cases and 2) a total topic coverage measure (TTC), which is the sum of all maximum topic coverages, in TSCV (SUM(TSCV[1]+TSCV[2]+...+TSCV[n])). The pseudo code for the topic coverage maximization algorithm is as follows:

```
Topic Coverage Maximization Algorithm

1) Unranked test cases = All tests
2) Ranked list= φ
3) Select the first test case (tc_i),
   randomly
4) TSCV = <p_i1, p_i2, p_i3,...p_im>
5) TTC = 1
6) while there is any unranked test case
   in the test suite
     a) for all unranked test cases
        (tc_x=<p_x1, p_x2, p_x3,...p_xm>)
          i) newTSCV(x)= <MAX(p_i1,p_x1),
             MAX(p_i2,p_x2), ...,
             MAX(p_im,p_xm) >
          ii)
             newTTC(x)=SUM(newTSCV(x)[1]+
             newTSCV(x)[2]+ ... +
             newTSCV(x)[m])
     b) add the test case (tc_x) with the
        maximum newTTC(x) to the ranked
        list
     c) remove tc_x from the unranked
        list and clear the newTTC(x) and
        newTSCV(x)
7) return the ranked test suite
```

Figure 2 summarizes the steps in our topic-coverage-based TSP. 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 IV).

B. Text diversity-based TSP

As we discussed in Section II, diversifying test cases is another common technique for TSP. For example, a TSP can maximize the diversity between test cases, where each test case is represented by a sequence of method calls (statically [15] or dynamically [16] extracted). In [18], 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* TSP. 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 [18], [28].

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 [18].

To maximize diversity between strings (test cases), Ledru et al. [18] 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 [18]. This algorithm has also been used by Thomas *et. al.*, in [15] as a baseline of comparison.

Our edition of text-diversity-based TSP uses the same algorithm as described in [18] and [15], but instead of applying it on the test scripts written in programming languages, we apply it on the English texts of systems-level test cases. More

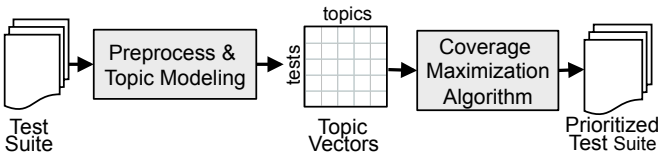

Fig. 2. Overview of our proposed topic-based TCP technique.

details about it will be discussed in the experiment design subsection in Section IV.

C. Risk-driven clustering

The last technique used in this study uses the historical fault detection information, described in Section II. This TSP requires access to the previous execution results of the test cases, which we had, in this case study. A common risk-driven TSP [8] only examines the last execution of the test cases. One can extend it to as many previous executions as possible [9]. 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 TSP 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 modified this approach and instead of having two clusters of previously failing vs. non-failing test cases, we create several clusters with different riskiness factor. 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 some failing tests (*FT(i)*) and some passing tests (*PT(i)*). Now assume we have a set of test cases (*TS(n+1)*) for release *n+1*. 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:

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

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

IV. Empirical study

In this section, we empirically evaluate different TSP 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 TSP techniques that we have adopted to the domain of black-box system level test prioritization, when the tests are written in natural languages.

A. Research objectives and questions

The objective of this research is to examine the effectiveness of well-known heuristics for test suite prioritization in a special context, where the type of information that is available for the TSP technique is very limited. In this context, not only the tests are black-box, which prevents a TSP 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 TSP 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 TSP heuristics are not directly applicable (or empirically evaluated) in this domain.

The TSP 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 all the test suites). However, the development paradigm is shifting more toward rapid releases and continuous deliveries [3]. 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 TSP technique in place (this can be even part of their build process).

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

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

RQ2: Does the relative effectiveness of the evaluated TSP 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 Mozilla Firefox, where the development strategy was Rapid release.

B. Subjects of the study

In this paper, we use system testing data from Mozilla Firefox web-browser project. We compare the system testing data of two release models. 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 use system testing data that we have used in our past work to study changes in the testing process [22]. 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"* [22]. 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.

C. Case study design

The experiments conducted in this study were exactly the same for RQ1 and RQ2, with the only difference on their subjects. Since the risk-driven TSP can be implemented in several ways, we have examined two basic ones and thus answered RQs by comparing five TSPs: a random TSP (as a baseline of comparison), topic-coverage, text-diversity, and two risk-driven TSPs. The first three TSPs 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.

1) Design decisions in TSPs: Next we explain the design decisions on the implantation of the TSPs that we have used in the experiments.

TABLE I. SYSTEMS UNDER TEST

Type	Release	Release date	No. of tests	No. of faults	Failure rate
TR	Firefox 3.0	12/2006	580	127	21.90%
	Firefox 3.5	7/2008	766	138	18.02%
	Firefox 3.6	8/2009	828	88	10.63%
	Firefox 4.0	2/2010	997	150	15.05%
RR	Firefox 5.0	4/2011	1055	6	0.57%
	Firefox 6.0	4/2011	1119	4	0.36%
	Firefox 7.0	5/2011	1111	4	0.36%
	Firefox 8.0	7/2011	1119	7	0.63%
	Firefox 9.0	8/2011	1114	4	0.36%
	Firefox 10.0	9/2011	1108	12	1.08%
	Firefox 11.0	11/2011	1121	3	0.27%
	Firefox 12.0	12/2011	1121	2	0.18%
	Firefox 13.0	2/2012	1189	4	0.34%

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

TopicCoverage: Covering maximum topics sooner by better ordering of test cases is the goal of a topic-coverage-based TSP. In Section II, we already have introduced the basic idea behind the approach and its general process, which consists of data preprocessing, topic extraction, and topic coverage 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 TSP 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 [29]. 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 our tool called *tcp.lda*, which is an open source tool for TSP using LDA [30]. 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 [15]. However, we did tune the other parameter of LDA (*i.e.*, K), which LDA is more sensitive to. K defines the number of topics to be extracted. Studies recommend anything between 5-500 [15]. For our small size corpuses, we tune K in the range of 5 to 50, using the data from the previous release. Basically, for each release, we run the LDA-based TSP on the previous release with K=5, 10, 15, ..., 45, and 50. Each TSP is executed 10 times and the K with the best results (*i.e.*, highest median APFD, which we introduce later in this section) will be chosen to be used in the next release. This way we had to exclude the very first release (Firefox 2.0) from our case study subjects and only use it as the training set for the second release. Table II shows the results of tuning for our case study. Given the variations in the results, we suggest tuning K, before applying a TopicCoverage TSP.

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 TopicCoverage TSP for TextDiversity, so that we only compare the effect of the TSP technique not the preprocessing. The only parameter left to set is the distance function. *tcp.lda* tool provides an option for text-diversity based prioritization, which has two built-in distance functions: Manhattan [15] and Euclidean [15] (they are selected in this tool due to their promising results in [18]). In our experiment, we tune the TextDiversity by running it with both distance

TABLE II. TUNING RESULTS FOR TOPICCOVERAGE AND TEXTDIVERSITY TSPS.

Type	Release	Distance function in TextDiversity	K in in TopicCoverage
TR	Firefox 3.0	Euclidean	5
	Firefox 3.5	Manhattan	5
	Firefox 3.6	Euclidean	10
	Firefox 4.0	Manhattan	15
RR	Firefox 5.0	Euclidean	25
	Firefox 6.0	Euclidean	15
	Firefox 7.0	Manhattan	10
	Firefox 8.0	Euclidean	5
	Firefox 9.0	Euclidean	50
	Firefox 10.0	Manhattan	5
	Firefox 11.0	Euclidean	10
	Firefox 12.0	Euclidean	10
	Firefox 13.0	Manhattan	15

functions 10 times on the previous release. For each release, we use the distance function that was more effective (*i.e.*, higher median APFD, to be defined later in this section), in the previous release.

RiskDriven: As we discussed in Section II, compared to TopicCoverage and TextDiversity, risk-driven TSPs have access to some extra knowledge about previous test execution results (pass or fail). Our version of RiskDriven TSP combines the results of all previous executions into the riskiness clusters. Therefore, a TSP 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 TSP 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 TopicCoverage, 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 TSP are RiskDrivenRandom and RiskDrivenDiversity (where we used the Random and TextDiversity TSPs, respectively, to rank the intra-cluster test cases).

2) TSP evaluation: We use the well-known APFD (Average Percentage of Fault-Detection) metric for assessing the effectiveness of a TSP, which is originally introduced by Rothermel et al. in [4]. APFD captures the average of the percentage of faults detected by a prioritized test suite. APFD is given by

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

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 TSP technique's effectiveness increases (i.e., more faults are detected with fewer test cases), the APFD metric approaches 100.

We run each of the five TSPs, on each of the four traditional and nine rapid releases of our case study, 100 times. To compare the APFD values of the different TSP techniques, we apply the non-parametric significant test, Mann-Whitney U test [31], 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 [31]. 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 TSPs, per release, we also report the distribution of APFDs for each TSP over the 100 runs, with a boxplot.

D. Case study results

In this section, we explain and discuss the results of the experiments under the two RQs.

*1) **RQ1 Results**:* To answer the question of "*What is the most effective TSP technique for black-box manual testing, in a traditional software development context?*", we look at the effectiveness of the five TSP techniques on the four traditional releases. Table III summarizes the APFD results as the median of 100 runs per TSP technique. The hypothesis is that RiskDriven approaches would outperform RandomTSP, TopicCoverage, and TextDiversity 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 TSPs are not by chance. The results are shaded cells in the Table III. Only two pairs comparisons (RiskDrivenDiversity vs. RiskDrivenRandom in version Firefox 3.5 and RiskDrivenDiversity vs. RandomTSP 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 paired comparisons of effect size (A measures) between RiskDrivenDiversity and all the other four TSPs. As we explained A measures less than 0.5 means that RiskDrivenDiversity is likely to perform worse than the compared with TSP, 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 TSPs' 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 TopicCoverage or TextDiversity is the better TSP. In fact, all TSPs including RiskDrivenDiversity and even RandomTSP 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 TopicCoverage, which, unfortunately, has a higher variance and is the worst in Firefox 3.6, which makes it unreliable.

*2) **RQ2 Results**:* In RQ2 ("*Does the relative effectiveness of the evaluated TSP techniques from RQ1 change, when the development moves toward rapid-releases?*"), we will analyze the same five TSPs but on nine more recent releases of Mozilla

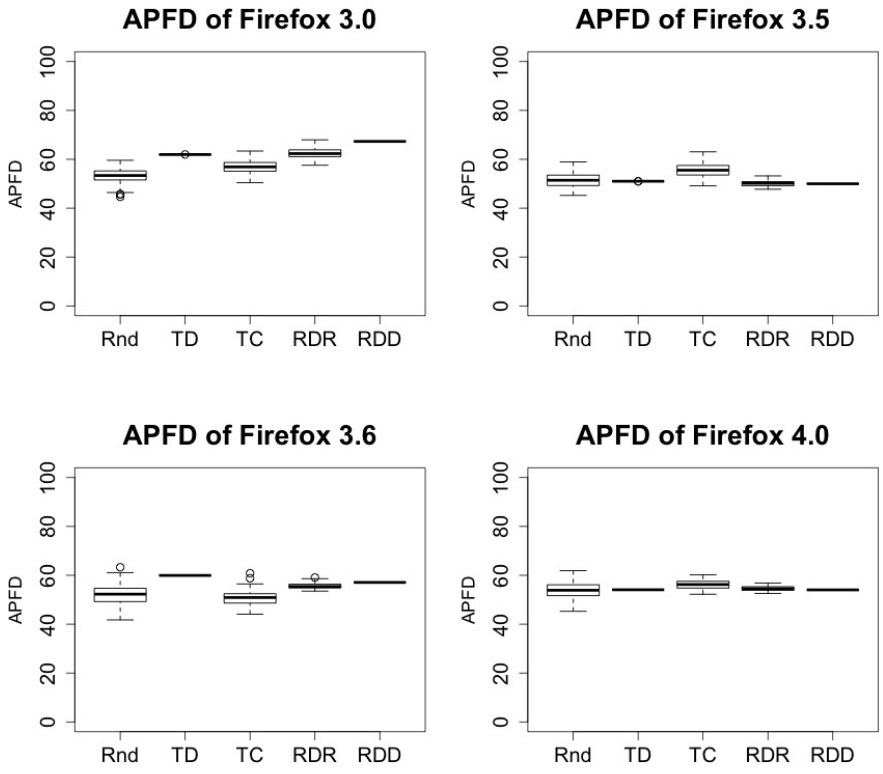

Fig. 3. Distribution of the APFDs of the five TSPs on four traditional releases, over 100 runs, as boxplots – (Abbreviations are the same as Table III)

TABLE III. MEDIAN APFDS (OVER 100 RUNS) OF THE FIVE TSPS AND THE EFFECT SIZES OF RISKDRIVENDIVERSITY VS. ALL OTHER FOUR TSPS, IN THE FOUR TRADITIONAL RELEASES – RND(RANDOMTSP), TD(TEXTDIVERSITY), TC(TOPICCOVERAGE), RDR(RISKDRIVENRANDOM), AND RDD(RISKDRIVENDIVERSITY

versions	Median APFD					Effect size of RDD vs.			
	Rnd	TD	TC	RDR	RDD	Rnd	TD	TC	RDR
Firefox 3.0	53.37	61.95	56.89	62.34	67.32	1.00	1.00	1.00	0.99
Firefox 3.5	51.43	51.02	55.54	50.18	49.99	0.35	0.00	0.03	0.44
Firefox 3.6	52.29	59.95	50.91	55.42	57.09	0.89	0.00	0.98	0.94
Firefox 4.0	53.88	54.08	56.23	54.52	54.03	0.54	0.00	0.14	0.31

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 TSP 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 TSPs in the rapid releases vs. traditional releases.

Table IV summarizes the APFD results as the median of 100 runs per TSP technique. The first observation from the table is that unlike RQ1, both RiskDriven approaches are by far more effective than the other three TSPs. On average, the median APFD of the best RiskDriven TSP improves the median APFD of the best of the other three TSPs by 65% (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 RandomTSP, TopicCoverage, 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.

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 in the results, which increases the reliability of the TSP 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 TSP 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

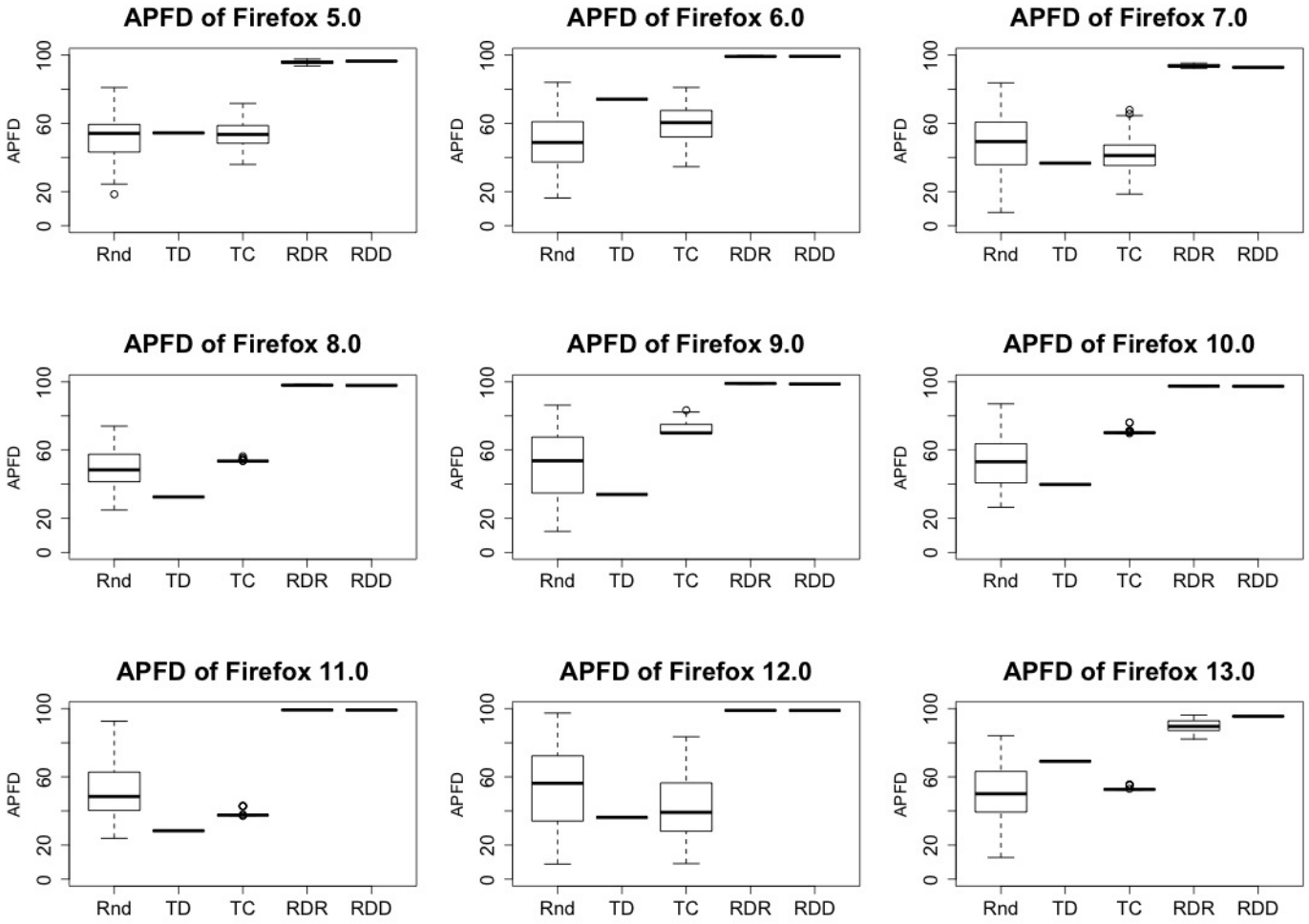

Fig. 4. Distribution of the APFDs of the five TSPs on nine rapid releases, over 100 runs, as boxplots – (Abbreviations are the same as Table III)

TABLE IV. MEDIAN APFDS (OVER 100 RUNS) OF THE FIVE TSPS AND THE EFFECT SIZES OF RISKDRIVENDIVERSITY VS. ALL OTHER FOUR TSPS, IN THE FOUR RAPID RELEASES – (ABBREVIATIONS ARE THE SAME AS TABLE III)

versions	Median APFD					Effect size of RDD vs.			
	Rnd	TD	TC	RDR	RDD	Rnd	TD	TC	RDR
Firefox 5.0	54.09	54.39	53.46	95.90	96.45	1.00	1.00	1.00	0.76
Firefox 6.0	48.73	74.08	60.40	99.15	99.21	1.00	1.00	1.00	0.57
Firefox 7.0	49.27	36.64	41.09	93.53	92.78	1.00	1.00	1.00	0.15
Firefox 8.0	48.39	32.53	53.53	98.02	97.88	1.00	1.00	1.00	0.31
Firefox 9.0	53.70	33.99	70.03	99.07	98.73	1.00	1.00	1.00	0.03
Firefox 10.0	53.16	39.85	70.17	97.48	97.38	1.00	1.00	1.00	0.21
Firefox 11.0	48.47	28.39	37.58	99.21	99.14	1.00	1.00	1.00	0.06
Firefox 12.0	56.23	36.24	39.19	98.92	98.89	1.00	1.00	1.00	0.33
Firefox 13.0	50.08	69.12	52.67	89.55	95.50	1.00	1.00	1.00	0.98

select RiskDrivenDiversity as the most effective and reliable TSP for the rapid release environments.

*3) **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 two other techniques (TopicCoverage and TextDiversity) have been tuned to reduce the threat of being biased toward a specific topic size or distance function. Regarding the conclusion validity, since all TSPs studied here are randomized, we have carefully studied the distribution of results using the TSPs 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 distributions by boxplots and discussing the results. Finally, with respect to external validity, we should emphasis that this paper reports a case study on 13 releases of Mozilla Firefox and it does not try to over-generalizes the findings. We, however, believe that replicating this study and applying TSPs, in general, on non-code-based test cases (such as manual system tests and acceptance tests) are required, before robust conclusions can be made.

V. 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 coverage-based, diversity-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, in the context of rapid release (versions 5.0 to 13.0 of Mozilla Firefox). The results of the risk-driven test prioritization approach for rapid releases 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 the risk-driven approach and compare them with the basic one, proposed in this paper.

REFERENCES

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[2] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[3] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[4] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

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

[6] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5. ACM, 2003, pp. 128–137.

[7] L. C. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on uml designs," in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 252–261.

[8] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[9] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 119–129.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 329–338.

[11] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," *IEEE Transactions on Software Engineering*, 2011.

[12] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 6, 2013.

[13] H. Hemmati, L. Briand, A. Arcuri, and S. Ali, "An enhanced test case selection approach for model-based testing: an industrial case study," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 267–276.

[14] Y. Chen, R. L. Probert, and D. P. Sims, "Specification-based regression test selection with risk analysis," in *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2002, p. 1.

[15] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[16] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009, pp. 201–212.

[17] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *Software Engineering, IEEE Transactions on*, vol. 33, no. 4, pp. 225–237, 2007.

[18] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2011.

[19] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 233–244.

[20] J. Bosch, "Driving innovation through software experiment systems," Keynote talk at International Symposium on Empirical Software Engineering and Measurement, 2011.

[21] J. Jenkins, "Velocity culture (the unmet challenge in ops)," Presentation at O'Reilly Velocity Conference, June 2011.

[22] M. V. Mantyla, B. Adams, F. Khomh, E. Engstrom, and K. Petersen, "On rapid releases and software testing: A case study and a semi-systematic literature review," *Empirical Software Engineering*, pp. 1–41, In press.

[23] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2007.

[24] E. S. Raymond, *The Cathedral and the Bazaar*, 1st ed., T. O'Reilly, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[25] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox," in *MSR*, 2012, pp. 179–188.

[26] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The secret life of patches: A firefox case study," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012, pp. 447–455.

[27] L. Bolelli, . Ertekin, and C. Giles, "Topic and trend detection in text collections using latent dirichlet allocation," *Advances in Information Retrieval*, pp. 776–780, 2009.

[28] H. Hemmati, A. Arcuri, and L. Briand, "Empirical investigation of the effects of test suite properties on similarity-based test case selection," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 327–336.

[29] A. Marcus, "Semantic driven program analysis," in *Proceedings of the 20th International Conference on Software Maintenance*, 2004, pp. 469–473.

[30] S. W. Thomas. (2014) Topic coverage test prioritization tool. [Online]. Available: https://github.com/stepthom/tcp.lda

[31] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.