Prioritizing Browser Environments for Web Application Test Execution

Jung-Hyun Kwon
School of Computing, KAIST
South Korea
junghyun.kwon@kaist.ac.kr

In-Young Ko
School of Computing, KAIST
South Korea
iko@kaist.ac.kr

Gregg Rothermel
University of Nebraska–Lincoln
USA
grother@cse.unl.edu

ABSTRACT

When testing client-side web applications, it is important to consider different web-browser environments. Different properties of these environments such as web-browser types and underlying platforms may cause a web application to exhibit different types of failures. As web applications evolve, they must be regression tested across these different environments. Because there are many environments to consider this process can be expensive, resulting in delayed feedback about failures in applications. In this work, we propose six techniques for providing a developer with faster feedback on failures when regression testing web applications across different web-browser environments. Our techniques draw on methods used in test case prioritization; however, in our case we prioritize web-browser environments, based on information on recent and frequent failures. We evaluated our approach using four non-trivial and popular open-source web applications. Our results show that our techniques outperform two baseline methods, namely, no ordering and random ordering, in terms of the cost-effectiveness. The improvement rates ranged from -12.24% to 39.05% for no ordering, and from -0.04% to 45.85% for random ordering.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Empirical software validation**; **Maintaining software**;

KEYWORDS

Web application testing, Regression testing, Browser environments

ACM Reference Format:
Jung-Hyun Kwon, In-Young Ko, and Gregg Rothermel. 2018. Prioritizing Browser Environments for Web Application Test Execution. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180244

1 INTRODUCTION

Modern, client-side web applications are becoming increasingly more complex and fault prone [34, 40]. Faults in these applications may manifest themselves differently in various combinations of web browsers, their versions, and underlying operating systems. One reason this occurs is that some browser combinations may not support certain features that a client-side web application requires.

To prevent such configuration-specific faults from affecting the usage of a web application, appropriate testing is needed. This is particularly true as a web application evolves, because new or changed code can cause unexpected faults in existing, previously-tested functionalities. For this reason, techniques for regression testing web applications have been actively studied. Such techniques include approaches for automatically generating test cases or oracles from previous versions of web applications [20, 34–37, 45, 51], repairing test cases that are invalid after web application updates [21, 25], selecting subsets of or prioritizing regression test suites [49, 59], augmenting test suites [33], among others. None of this work, however, has considered the additional problems that arise when regression testing must be performed across different web-browser environments (henceforth referred to as browser environments), such as environments in which different web browsers and operating systems are utilized.

When a developer modifies a web application that is meant to function in different browser environments, regression testing that web application can require a large amount of time and computing resources. To date, more than a hundred different web browsers exist [57]. Each web browser can have many different versions in the field simultaneously, and each can be run on various types of operating systems on a wide range of different computing devices. This creates problems of scale for regression testing of web applications [44]. Extended regression testing times delay the feedback that can be provided to developers on failing combinations of web browsers, and therefore, delay the debugging of failures and the release of new versions.

In this work, to improve the process of regression testing client-side web applications, and in particular to reduce delays in providing feedback about failures, we propose six different techniques for *prioritizing browser environments.*[1] Our techniques make use of information about the *failure history* associated with browser environments in prior applications of regression testing. Two of our techniques utilize recent failure information, including types of web browsers, browser versions, and of operating systems, that is stored in a cache; test execution environments are scheduled according to the cached information. Browser environments with recent failure information are given higher priority than others, and will be tested early. Another two techniques are based on commonly

[1] Our techniques should not be confused with existing approaches for prioritizing *test cases*; we are focusing instead on browser environments under the assumption that existing test cases will be run on each environment on which they are applicable.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180244

failing browser environments, and we propose a failure-frequency-based approach that gives greater priority to browser environments under which regression testing failed many times in prior builds, and machine-learning-based approaches that automatically learn the pattern of failures and predicts browser environments on which a web application test is likely to fail. The last two techniques consider information on both recent failing and commonly failing environments. The techniques first schedule browser environments based on cached information, and for browser environments with equal priority, apply a failure-frequency-based or machine-learning-based approach to order them.

Our prioritization techniques have several advantages. First, our techniques are applicable to modern web applications. Modern web applications are written in a mixture of languages such as HTML, JavaScript, and CSS; therefore, obtaining code coverage for test cases, which is usually required by traditional test case prioritization methods, is challenging and expensive. Our techniques, however, do not require code coverage information. Second, our techniques are especially effective for supporting Continuous Integration (CI) practices. In CI environments, there is a short time interval between runs of regression tests. Developers frequently check their code in to the mainline codebase, and regression tests relevant to that code need to be performed in applicable browser environments. Techniques for calculating code coverage cannot keep up with the pace of change that occurs in such processes.

We empirically studied our techniques on four non-trivial, popular open source web applications. Our results show that our techniques can be cost-effective. Our approaches generally detect more failures faster than two baseline approaches, in which browser environments are not prioritized or are randomly ordered. We also compared our six prioritization techniques, and analyzed which techniques are more cost-effective than the other techniques for each experiment object. This analysis can suggest which prioritization techniques a developer should apply first in their web application testing. In addition, we addressed some practical issues about using our techniques in industry, such as prioritization time overhead and parallel test execution.

The main contributions of this work are as follows:

- This is the first work to investigate the process of prioritizing browser environments for web applications.
- We propose four novel techniques for prioritizing web browser environments for regression testing, that rely on test results from prior regression testing sessions.
- We report the results of an empirical evaluation of our techniques using non-trivial, real-world web applications; our results demonstrate the cost-effectiveness of our techniques.
- We make our implementations and data publicly available [9].

2 MOTIVATION

To further motivate this work, we present the results of an initial investigation into whether web-browser-environment scheduling affects the cost-effectiveness of regression testing for web applications. For this analysis, we considered Mootools [38], a popular open-source web application that manipulates Document Object Model (DOM) elements and handles events. To obtain web-browser environments, and to measure the time required to test the application in each environment, we retrieved previous build histories of Mootools from Travis CI [39], where test results for Mootools are stored. The build histories we used began on March 16, 2014 and ended on February 9, 2015. We excluded build histories that were not directly relevant to this investigation. For instance, some build histories do not contain test results due to compilation failures, and some build histories do not provide information on failed web-browser environments. After excluding such build histories, ten remained.

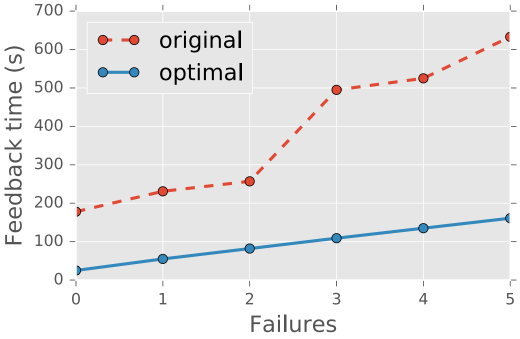


Figure 1: The effect of prioritizing web-browser environments for web applications

For each build, we counted the number of web-browser environments tested. The median, minimum, and maximum numbers across the builds were 23.5, 22.0, and 24.0, respectively. Each web-browser environment consists of multiple properties; these include browser type and version, operating system and version, and build type ("default" and "nocompat").[2] There are five browser types, but by varying other properties of the web browser environments, the number of environments is increased to 24.

One common strategy for reducing regression testing time across execution environments is to run test suites under each environment concurrently. Popular cloud-based testing services such as Sauce Labs [50] and Browser Stack [10] support such approaches, and Mootools uses Sauce Labs. Although test suites can be run concurrently in such cases, the total testing time required may vary with the number of environments to be tested. If the number of testing environments is greater than the number of resources available for parallelization, a substantial amount of time can still be required to detect faults that can be uncovered only in specific environments. Increasing the number of resources available can address this problem, but at additional costs. If there are many web applications to manage, it becomes more important to use computing resources in an efficient manner by preventing each web application from consuming excessive resources. For a similar reason, existing studies on test case prioritization emphasize the importance of scheduling test cases in a concurrent environment to improve the cost-effectiveness of regression testing [17].

We compared two different schedules of web-browser environments on Mootools: optimal and original, and measured whether the optimal schedule improved the cost-effectiveness of regression testing. The original ordering involves using the test schedule that

[2] The terms "default" and "nocompat" indicate whether the application is compatible or not with older versions.

is recorded in the build history of Mootools, whereas the optimal schedule places all web-browser environments that were observed to fail ahead of those that were not observed to fail. We then extracted the execution time for each of the browser environments ("job times" in Sauce Labs) from the build history. Next, we calculated the times at which web-browser environments that contained at least one failing test cases were reported. We then compared the calculated times ("feedback times") reported for the optimal and original schedules.

Across all ten builds, the optimal schedule provided feedback 315.12 seconds faster on average than the original schedule. Figure 1 provides a line plot showing the feedback times obtained for each schedule when browser environments used in build ID 200 were scheduled. The optimal schedule is presented as a solid line, and the unordered schedule as a dashed line. Across 24 browser environments, five failed. The X-axis denotes each failed browser environment, and the Y-axis denotes the feedback time associated with each failed browser environment. The optimal schedule provided feedback 153 to 472 seconds faster than the original schedule (68% to 86% faster). This shows that techniques for scheduling web browser environments do have the potential to improve feedback.

3 TECHNIQUES FOR PRIORITIZING WEB-BROWSER ENVIRONMENTS

In this section, we present techniques for prioritizing web-browser environments for use when regression testing web applications. We modify the formal definition of the test case prioritization problem [46] to formally define the problem of prioritizing web-browser environments as follows:

Definition 1: Problem of prioritizing test-execution environments:
Given: E, a list of web-browser environments, PE, the set of permutations of E, and f, a function from PE to the real numbers.
Problem: Find $E' \in PE$ s.t. $(\forall E'')(E'' \in PE)(E'' \neq E')$ $[f(E') \geq f(E'')]$

In Definition 1, PE refers to all possible orderings of E, and f represents an objective function used to quantify a goal of prioritization. In this work, our goal is to increase the rate of fault detection of the test suites for a web application under a certain ordering of web-browser environments. Different objective functions are defined and used for different prioritization techniques.

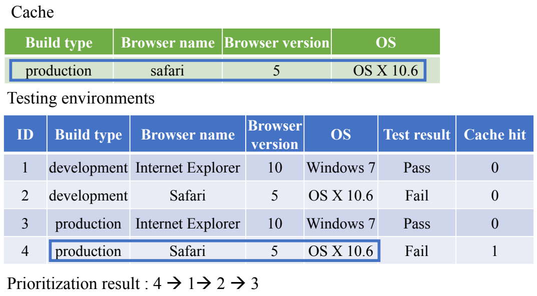
In this work we consider eight different techniques for ordering web-browser environments overall. The first two techniques represent cases in which no heuristic is applied; these serve as baseline orderings to compare results against, as follows:

M_1: **No prioritization.** The web-browser environments are not prioritized; they retain the ordering utilized by developers initially.

M_2: **Random prioritization.** The web-browser environments are ordered randomly.

The six other techniques we present can be categorized into three classes: techniques that consider *recently failing* web-browser environments, techniques that consider *frequently failing* environments, and techniques that consider both. We describe these next.

M_3: **Exact matching-based prioritization.** This technique considers recently failed web-browser environments first. The technique is based on the assumption that recently failed web-browser environments may fail again on future builds of a web application.



Cache

Build type	Browser name	Browser version	OS
production	safari	5	OS X 10.6

Testing environments

ID	Build type	Browser name	Browser version	OS	Test result	Cache hit
1	development	Internet Explorer	10	Windows 7	Pass	0
2	development	Safari	5	OS X 10.6	Fail	0
3	production	Internet Explorer	10	Windows 7	Pass	0
4	production	Safari	5	OS X 10.6	Fail	1

Prioritization result : 4 → 1→ 2 → 3

Figure 2: Example of matching browser environments

As such, the technique is based on prior work that considers recent failure information [17, 30]. To utilize recent failure information in prioritizing web-browser environments we use a cache. A cache lists the environments for which previous builds failed. If web-browser environments in the current build are contained in the cache, a higher priority is assigned to those environments.

We formally define a cache-hit as follows:

$$\left.\begin{aligned} &TE = \{C_1, C_2, ..., C_n\}\\ &Build_i \subset TE\\ &Cache_{it} \subset TE\\ &C_m = (x_1, x_2, ..., x_k)\\ &Cache\_hit_i = Build_i \cap Cache_{it} \end{aligned}\right\} \quad (1)$$

Here, TE is a set of web-browser environments that a developer is considering for regression testing. $Build_i$ represents a subset of TE at build i. $Cache_{it}$ refers to the subset of TE that exhibited failures when moving from build $i-1-t$ to build $i-1$. t is a threshold that specifies the build range, and a browser environment in the range can be cached. If $i-1-t$ is less than 1, the value is set to 1. Each browser environment (C_m) in TE is a vector containing properties of browser environments. For example, Figure 2 illustrates how the cache-hit occurs. A web-browser environment in the figure consists of four properties: Build type, Browser name, Browser version and OS. Each property in the vector takes a value from a set of possible values. $Cache\_hit_i$ refers to the intersection of $Build_i$ and $Cache_{it}$. In Figure 2, a single browser environment hits the cache. The cache-hit browser environment receives a higher priority than the other environments. The browser environments are scheduled based on priority score in a descending order. If the priorities of two browser environments are the same, they can be scheduled in the original order or randomly. In Figure 2, the priority of the last browser environment is 1 and the priorities of the other environments are 0; therefore, the scheduling result becomes "4, 1, 2, 3".

An advantage of the cache-based approach is that it is computationally light-weight, and therefore is suitable for fast development environments such as CI environments. A disadvantage of using the cache-based approach, however, is that the performance of the approach could be poor if a web application does not satisfy our assumption that a recently failed browser environment is likely to fail again in near future. A second potential disadvantage is that the approach may not be cost-effective in situations in which testing of the previous build fails on most browser environments, but testing in the next build fails on few of the environments. Because there is no additional priority considered among cache-hit browser environments, the cost-effectiveness of this approach may suffer if the failed environments are not detected before most of the cache-hit browser environments have been tested. A third potential disadvantage is that it is possible that only some properties in a browser environment are relevant to predicting failed environments in the next build, and the cache-based approach does not consider such cases. Instead, it requires all properties of a browser environment to be matched.

M_4: **Similarity matching-based prioritization.** Our second technique attempts to address the second and third disadvantages of the exact matching technique. Instead of assigning the same priorities to cache-hit web-browser environments, this technique calculates the similarity between each environment in the build under test and each environment in the cache. The priority of a browser environment in the build under test is the average similarity, which is calculated by summing up the similarity values between that environment and each environment in the cache, divided by the number of cached browser environments. Then, the browser environments are scheduled based on their similarity scores in a descending order. To measure similarity in this context, we use the Jaccard similarity coefficient [58]. This assigns a floating number between 0 and 1 to a browser environment. In comparison to the exact matching technique, this reduces the number of cases in which browser environments have the same priority. In addition, this assigns a positive priority to browser environments that do not exactly match those in the cache. For example, even if a browser version in a browser environment does not exist in the cache, the environment can receive a positive priority if other features such as the browser name and operating system are in the cache.

We use Equation 2 to measure the similarity between a browser environment in a build under test and an environment in the cache. The notations are the same as those used in Equation 1. Sim_{ij} represents the similarity score for the jth browser environment in $Build_i$. In addition, for each vector in $Cache_{it}$, the Jaccard similarity coefficient is calculated by finding the norm of the intersection between a vector from $Cache_{it}$, and a vector from $Build_i$ divided by the norm of their union. The Jaccard similarity coefficients are then averaged.

$$\left. Sim_{ij} = \frac{1}{|Cache_{it}|} \sum_{C_k \in Cache_{it}} \frac{|Build_{ij} \cap C_k|}{|Build_{ij} \cup C_k|} \right\} \tag{2}$$

For the browser environments shown in Figure 2, using our approach, the similarity scores calculated for the environments are $0/8 = 0, 3/5 = 0.6, 1/7 = 0.14$, and $4/4 = 1$, respectively. Note that when using the exact matching method, browser environments 2 and 3 have priority 0. When using the similarity matching method, however, they have priority values 0.6 and 0.14, respectively, and therefore the prioritization result becomes "4, 2, 3, 1". If this added level of distinction is useful, it may render the similarity-matching technique more cost-effective than the exact matching technique.

M_5: **Failure-frequency-based prioritization.** One way in which to consider frequently failed web-browser environments is to count the number of previous failures for each browser environment. This method is based on the assumption that frequently failed web-browser environments are likely to fail again in future builds. If testing against a web application fails more often in a certain web-browser environment than in other environments, this method can be effective for predicting environments that will fail in future builds. Thus, this method gives higher priority to browser environments that have greater numbers of previous failures. To schedule browser environments using this method, we need to have a key-value data structure in which the key is a browser-environment vector, C_i, and the value is the number of previous builds in which the environment failed.

One issue regarding this technique is that using all the properties of a browser environment to count failure frequency may decrease the cost effectiveness of the browser-environment scheduling. For example, when a browser environment consists of the browser name, browser version, and OS name, using a subset of those properties such as the browser name and OS name as the key and the number of the failed browser environments containing the key as the value can result in higher cost effectiveness. It is difficult, however, for a developer to manually determine which subset of the properties might be effective to render scheduling more cost effective. As the number of properties in a browser environment increases, this issue may become more important.

M_6: **Machine-learning-based prioritization.** To address the problems of the failure frequency method, a machine learning (ML) method may be useful. An ML method can automatically learn a pattern of failures from the previous build history, and provide a failure probability for browser environments in subsequent builds. Here, we use Bernoulli Naive Bayes classifier, which calculates failure probabilities based on Bayes' theorem [4]. The properties of the browser environments in this study are discrete data, and the Bernoulli Naive Bayes classifier is suitable for such data.

A classifier is built by learning test history from previous builds in which some but not all browser environments failed. The classifier returns the probability of test failures under a given browser environment. The classifier adaptively updates its learning model once the testing of a build is finished and has failed on some but not all browser environments. The priority score for each browser environment is equal to the output of the ML model, the probability of test failures under the environments. The browser environments are then sorted based on priority scores in descending order.

Equation 3 shows how to calculate the probability of test failures under a given browser environment. D refers to a set of browser environments and k refers to the number of possible browser environments. T is a set of test results, which are either "Pass" or "Fail". $P(Fail|D_i)$ is the probability of test failures under a given browser environment, D_i. Using Bayes' theorem, the posterior, $P(Fail|D_i)$, can be calculated by a likelihood, $P(D_i|Fail)$, times the prior, $P(Fail)$, divided by the evidence, $P(D_i)$. $P(Fail)$ or $P(True)$ can be obtained from either a uniform distribution or a domain expert, $P(D_i|Fail)$ or $P(D_i|True)$ can be calculated from the test results from the previous builds, and $P(D_i)$ can be calculated from the prior results and likelihood. Note that this technique orders browser environments based on the posterior probability, so a threshold of the posterior probability is not needed.

$$\left. \begin{array}{l} D = \{D_1, D_2, ..., D_k\} \\ T = \{Pass, Fail\} \\ P(D_i) = \sum_{j \in T} P(D_i|T_j) \times P(T_j) \\ P(Fail|D_i) = \frac{P(D_i|Fail) \times P(Fail)}{P(D_i)} \end{array} \right\} \tag{3}$$

One potential advantage of the ML-based method is that it provides more sophisticated scheduling than the cache-based or failure-frequency-based methods. Thus, each property of a browser environment can be used in an adaptive manner to predict failure-prone environments. A second advantage is that the approach is suitable for fast development environments, because a developer does not need to create a new learning model whenever a new build is conducted. Instead, they need only update the existing model based on the new testing results that are generated from the current build. A potential disadvantage of the ML-based approach is that it assumes there exist frequently failed browser environments or properties; if this assumption is not satisfied, the performance of the approach may be compromised.

M_7 and M_8: **Hybrid prioritization.** Techniques may perform differently across applications, so it could be difficult for a developer to decide which technique to use. It is possible to combine techniques to reduce these differences. The hybrid prioritization technique uses two prioritization criteria, so browser environments are ordered based on the first criterion, and if more than one environment has the same priority, the hybrid prioritization uses the second criterion to order those environments.

We chose the exact matching-based prioritization technique as the first criterion because the priority score of the exact matching-based technique is either zero or one, so it is more likely for the technique to have more browser environments with the same priority than the other techniques. The second criterion can be either the failure-frequency-based or ML-based techniques. We define M_7 as a prioritization technique that combines the exact matching-based technique with the failure-frequency-based technique, and we define M_8 as a prioritization technique that combines the exact matching-based technique with the ML-based technique. One advantage of the hybrid prioritization techniques is that they can consider both recently failed browser environments and frequently failed browser environments.

Table 1 shows the code, mnemonic and description of the prioritization techniques that we introduced in this section.

Table 1: Prioritization Techniques

Code	Mnemonic	Description
M_1	untreated	no ordering
M_2	random	random ordering
M_3	exact-matching	exact match-based
M_4	similar-matching	similar match-based
M_5	fail-freq	failure frequency-based
M_6	ML	machine learning-based
M_7	$M_3 + M_5$	exact-matching plus fail-frequency
M_8	$M_3 + M_6$	exact-matching plus ML

4 EVALUATION

We conducted an empirical study to investigate the effectiveness of the foregoing techniques.

4.1 Objects of Study

For this study, we selected four web applications. These applications were all obtained from the "popular package" section of bower.io [8], which is a repository of client-side JavaScript packages. Table 2 provides details on the study objects as of October 26, 2016, including lines of code, time period of collected build history, number of builds, number of failed builds, and the median number of web browser environments (obtained via processes described below). Backbone [1] is a JavaScript framework that supports the use of Model-View-Controller patterns in web application development. Bootstrap [5] is an HTML, CSS and JavaScript framework for developing responsive and mobile web applications. Lodash [28] is a JavaScript utility library that provides various functions for improving modularity and performance, among others. Underscore [56] is a JavaScript utility library that provides help functions for functional programming. We excluded "Mootools," which was used for our initial investigation described in Section 2, because it has too few builds available to support statistical analysis.

Table 2: Objects of Study and Build Information

	Lines of Code	Build Period	Builds	Failed Builds	Median BEs
Backbone	1214	15/2/20-16/10/27	3840	105	23
Bootstrap	2384	14/1/31-16/11/24	11219	58	12
Lodash	5893	13/11/05-16/10/10	520	227	99
Underscore	1683	15/2/20-16/10/27	1546	50	23

We measured lines of code using CLOC [11], which counts the lines of code excluding blank and comment lines. We collected all previous build information for each object from Travis CI, a cloud-based continuous integration service [54]. For each build, unit testing is conducted in a testing environment. QUnit [43] is the unit testing framework that is used for the objects. A cloud-based cross-browser testing service, Sauce Labs, is used to conduct regression testing of the objects in each testing environment. Sauce Labs considers testing in a testing environment to be a job, so the number of jobs is equal to the number of web-browser environments. For all objects except Lodash, the job IDs are recorded in build logs obtained from Travis CI. Given job IDs, we were able to obtain test results for each testing environment using the Sauce Labs REST API. In Lodash, passed job IDs are not recorded in build logs, but property names of the web-browser environments and unit testing results are recorded in the logs. In Table 2, the "Build Period" and "Builds" columns show the period and number of builds on which regression testing is performed for each object.

Prioritization techniques are applied to browser environments following the occurrence of a failed build. In this study, we exclude builds that have passed in regression testing for all browser environments, and builds that have failed in regression testing for all browser environments. The "Failed Builds" column in Table 2 indicates the number of builds that we actually utilized.

The number of browser environments considered in a given build can change across builds, because some browser environments may be removed and some new ones (e.g., using new web browsers or browser versions) may be added. The "Median BE" column in Table 2 indicates the median number of browser environments that are tested across all builds for our objects. Lodash has the largest number of browser environments because the number of properties in its browser environments (five) is larger than that for the other objects (all of which have three). Specifically, Lodash adds module names with information on relevant modules, and a build type such as "development" or "production".

4.2 Variables and Measures

4.2.1 Independent Variable. Our independent variable involves prioritization techniques. We evaluated eight techniques; these correspond to the two baseline techniques (M_1 and M_2) and the six heuristics (M_3 – M_8) described in Section 3. The ordering in which browser environments were tested in practice for each web application was obtained from the Travis CI log data; we regard this ordering as the original schedule.

4.2.2 Dependent Variable. To measure the effectiveness of prioritization techniques at improving the rate of failure detection, we used the Average Percentage Faults Detected (APFD$_C$) metric; this metric is typically used to measure the cost-effectiveness of prioritizing test cases [15] when those test cases have different costs (execution times). We used this metric because we can regard a browser environment as a test case; therefore, scheduling browser environments will produce effects similar to those seen when prioritizing test cases.

The equation for APFD$_C$ is as follows:

$$APFD_C = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=C_i}^{n} t_j - \frac{1}{2}t_{C_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

Here, n is the number of browser environments and m is the number of failures. C_i refers to the position, in the list of browser environments considered, of the browser environment that detects the ith failure. t_j denotes the cost of performing regression testing on a C_j, and f_i refers to the severity of ith failure. The more effective a prioritization technique is, the closer the APFD$_C$ value is to 100%.

From the log data for each application we were able to determine when regression testing on each browser environment began and ended; this yields a measure of *job time*. In this study, for all apps other than Lodash, For Lodash, job times for builds are not available, so we set t_j to 1, which implies treating all builds as being equally costly. We were unable to assess the severity of failures, however, and therefore we set f_i to 1 for each failure.

4.3 Study Procedure

We simulate regression testing on different browser environments and with different orders of environments using the log data for Travis CI and Sauce Labs. The procedure was as follows:

(1) Build the cache, data structure for failure frequency, and ML model based on the test results of the first-failed build.
(2) Prioritize the browser environments in the current build based on the priority produced by each technique.
   (a) If more than one browser environment has the same priority, order them randomly.
(3) Measure the APFD$_C$ value of the result of the regression testing session on the current build.
(4) Repeat (2) and (3) 30 times; average the APFD$_C$ values.
(5) Update the cache, data structure for failure frequency, and ML model based on the test result from the current build when regression testing on that build fails on some but not all browser environments.
(6) Repeat Steps (2) – (5) for the next failed build.

We performed the foregoing process for each of the eight prioritization techniques considered, with two exceptions. For random ordering, Step (2) is performed via simple randomization using a different ordering on each of the 30 runs, and for no ordering, Steps (2)-(a) and (4) are skipped. We performed 30 runs for techniques other than no ordering (Step (4)) because we used a random order whenever there is a tie (in Step (2)-(a)), which renders techniques non deterministic. We could have used no ordering instead of random ordering; however, there is no evidence that the original order is meaningful.

We adaptively update the ML model online, so training data consists of the properties of the failed browser environments in previous builds and their test results, and test data is the properties of the browser environments under which tests are performed.

4.4 Threats to Validity

External validity: The web applications that we study are widely used and popular open source projects, regression tested across more than 10 browser environments. Their code sizes range from 1214 to 5893 lines, which places them as small and mid-size applications. As such, the applications cannot represent all classes of web applications, especially large-scale web applications such as on-line shopping and game applications. However, we were unable to find other web applications that publicly provide test failure data relative to cross-browser testing.

Internal validity: We have implemented the recent-failure-based methods and failure-frequency-based method by ourselves, and therefore there is a possibility that our implementations do not work as intended. To reduce this threat, we conducted a manual verification process with a small number of builds to determine whether results in those cases were correct. Another threat to internal validity is that we conducted our runs on a cloud server, and this may allow jobs to be interrupted, causing measures of time to differ. However, Sauce Labs creates a new virtual machine having the same computing resources for each job, and each job performs regression testing on an application with each browser environment. In this context, a regression test run cannot be interrupted by another regression test run within the cloud server.

Construct validity: We used APFD$_C$ to measure prioritization technique cost-effectiveness. We assume that the severity of each failure is the same, but in cases in which it differs cost-effectiveness results could differ. We consider every detection of a failure when calculating APFD$_C$, even when a failure may have previously been detected; this is necessary, however, because it is difficult to accurately assess (in this experiment, or in practice) whether a given failure is "the same" as a previously encountered failure. We do not account for potential changes in APFD$_C$ values that may occur if faults are fixed during a testing process. We use job time to represent the cost of regression testing within a testing environment. Job time includes execution time of test suites and time for generating logs and metadata which help a developer to debug failures.

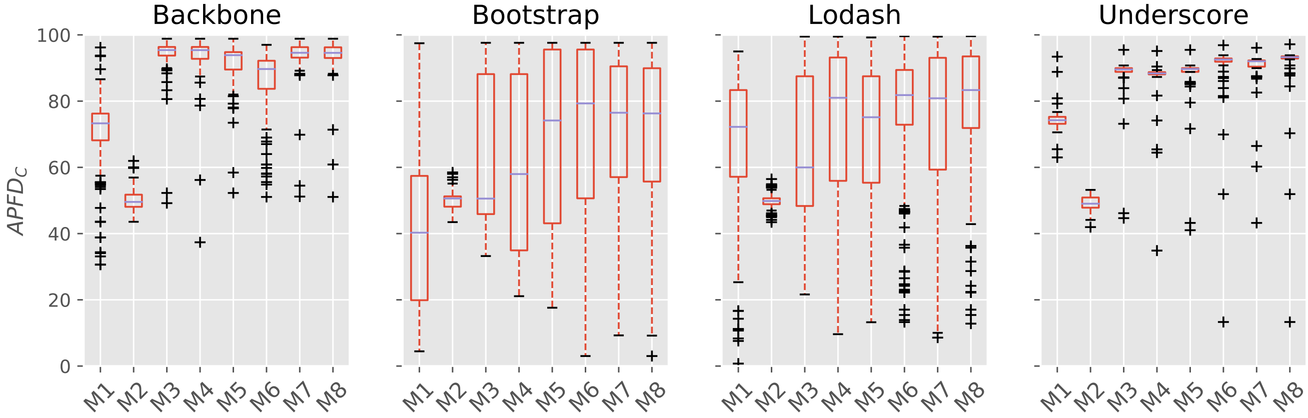


Figure 3: APFD$_C$ for each web application and prioritization technique

We consider job time alone as our measure for testing cost, but in practice, testing cost may include additional factors such as time spent by humans interpreting results.

Table 3: Pairwise Tests on Technique Pairs

Backbone		
Grouping	Mean	Technique
A	93.80	M3
A	93.61	M8
A	93.52	M7
A	93.45	M4
B	91.22	M5
C	85.24	M6
D	69.48	M1
E	50.10	M2

Bootstrap		
Grouping	Mean	Technique
A	68.35	M7
A	68.20	M6
A	67.78	M8
A	66.63	M5
A	62.78	M3
A	59.18	M4
B	50.05	M2
B	42.77	M1

Lodash		
Grouping	Mean	Technique
A	79.07	M8
A, B	76.15	M6
A, B	75.10	M7
B	72.05	M4
C	70.91	M5
D	67.02	M3
D	66.77	M1
E	49.84	M2

Underscore		
Grouping	Mean	Technique
A	89.41	M8
B	88.48	M6
B	88.23	M7
C	86.21	M3
C	86.02	M4
C	85.91	M5
D	73.07	M1
E	49.11	M2

4.5 Results and Analysis

Figure 3 provides data on the APFD$_C$ values obtained using the prioritization techniques and baseline techniques we considered, per web application. The Y-axes indicate APFD$_C$ values, and the X-axes denote techniques using their abbreviated names. For each application, for each technique, the box represents the distribution of APFD$_C$ values obtained across all the runs involving that technique and application.

Inspection of the boxplots suggests that for all four applications, the four basic prioritization techniques outperformed the baseline approaches (in terms of median performance) in almost all cases. The exceptions involve M3 compared with M2 on Bootstrap, and M3 compared to M1 on Lodash. Across all techniques and web applications, the median improvement rates ranged from -12.24% to 39.05% for no ordering, and from -0.04% to 45.85% for random ordering.

The techniques that achieved the highest median APFD$_C$ value are the exact matching-based prioritization technique (M3) on Backbone, and the ML-based technique (M6) on Bootstrap, and the hybrid technique using the ML-based technique (M8) on the other applications. The lowest median APFD$_C$ values among the heuristics occurred when using the ML-based technique (M6) for Backbone, the exact matching-based prioritization technique (M3) for Bootstrap and Lodash, and the similarity matching-based prioritization technique (M4) for Underscore. For both Backbone and Underscore, all the heuristics tended to achieve high APFD$_C$ values (more than 88% median APFD$_C$). For Bootstrap and Lodash, the exact matching-based prioritization technique (M3) displays the lowest median APFD$_C$ among the heuristics, but after combining it with the techniques that consider frequently failed browser environments (M7, M8), the median APFD$_C$ values increased by 25.82% and 22.12% on average, respectively.

To determine whether the APFD$_C$ differences we observed are statistically significant, we performed a one-way ANOVA test using Python SciPy [52]. Our null hypothesis was: H_0: all the prioritization techniques have the same APFD$_C$ means, and because results vary somewhat widely across web applications, we applied the test to the results on a per-application basis. The ANOVA test shows that the p-value for each application was less than 0.001, so we were able to reject the null hypothesis for each of the experiment objects. The results of this ANOVA test indicate that at least one of the prioritization techniques produced statistically significantly different mean APFD$_C$ values.

To determine which pairs of techniques differ, we applied pairwise tests. Because the rate of Type 1 errors (incorrectly rejecting the null hypothesis) increases when conducting two-sample t-tests multiple times, we corrected the significance level using Bonferroni correction [14]. After the pairwise testing, we grouped the techniques that are not statistically significant into the same group.

Table 3 shows the relationships between techniques obtained by the foregoing process, per program. Techniques are listed in terms of descending order of APFD$_C$ means, and shared grouping letters indicate cases in which techniques do not statistically significantly

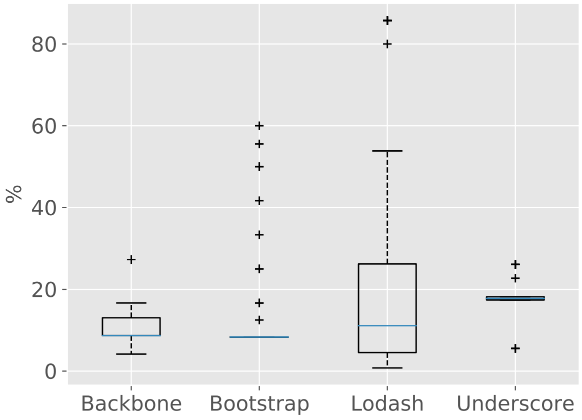


Figure 4: Distribution of ratio of failed environments to all environments in a build

differ. For all objects, the grouping of random ordering (M2) differs from that of all prioritization heuristics. For all objects except Lodash, the grouping of no ordering (M1) differs from that of all prioritization heuristics. (The exact match-based technique (M3) in Lodash belongs to the same group as no ordering.)

Based on the result of the pairwise testing, if these results generalize, we would suggest that techniques belonging to the first grouping (A) be the first choices for use in practice; however, these differ across web applications and practitioners would need to use results from initial runs to select techniques to use longer term. On Backbone, techniques that consider recently failed browser environments (M3 and M4) and hybrid techniques (M7 and M8) are such techniques. For Bootstrap, any prioritization heuristics can be used . For Lodash, the ML-based (M6) and hybrid techniques are such techniques, and for Underscore, the hybrid approach using the ML technique (M8) is the only such technique. That said, across all the objects, the hybrid approach using the ML technique always belongs to the first group, so this technique might be the one to consider first when a new web application must be tested across many browser environments.

5 DISCUSSION

We now present the results of additional analyses of our results. We analyze the ratio of failed browser environments and details about test cases and causes of failures. We also consider several other issues such as parallel test execution and prioritization time.

Ratio of failed browser environments in a build. We analyzed the ratio of failed browser environments to all browser environments tested for each build of each experiment object. Figure 4 shows the result. We found that the ratio of failed builds to total builds was less than 20% in all cases (9.64% for Backbone, 13.41% for Bootstrap, 19.01% for Lodash and 18.12% for Underscore). This result shows that there is room for prioritization of browser environments to provide faster feedback to developers.

Details on test cases and causes of failures. To identify the causes of test failures in browser environments and determine whether failures are environment-specific, we analyzed the number of failed unit test cases for each failed browser environment and the overlap among failed test cases between environments. In this analysis, we focus on test case failures, not faults. This is because it is difficult and error prone in many cases to trace failures to faults, in part because developers seldom explicitly report environment failures in issue reports.

From the test logs for Backbone and Underscore, we extracted the number of total and failed test cases, and the names of failed test cases, for each browser environment. For Bootstrap, failed test cases are not described in the logs, but we were able to obtain information about six builds from unit test reports in the Sauce Labs website. For Lodash, the number of total test cases is not in the logs, so we extracted the number including other information on 160 builds from the unit test reports.

The number of failed test cases is small for all objects. The percentage of failed test cases to total test cases ranged from 0.10% to 1.14%. The average number of total and failed test cases were 399.67 and 2.07 for Backbone, 341.96 and 3.89 for Bootstrap, 3173.70 and 3.32 for Lodash, and 211.49 and 1.47 for Underscore.

To calculate overlap among failed test cases between environments, we calculated Jaccard similarity coefficients between pairs of browser environments, and averaged the coefficients of all the pairs. A coefficient is calculated by the size of the intersection of the two failed test-case name sets divided by the size of their union. For Backbone, Bootstrap and Underscore, the overlaps are low (8%, 0% and 4%, respectively). Meanwhile, for Lodash, the overlap is high (90%). It is possible that faults in Lodash manifest themselves in more browser environments than faults in other objects.

We were able to find several environment-specific failure causes by reading conversations among developers and commit messages. Table 4 provides examples. Four causes involved implementation; in these cases browsers did not fully implement certain features. One cause involved a test case: even when an application does not contain environment-specific failures, its test code can contain an environment-specific fault. Two other causes involved mobile devices. Mobile browsers have different levels of feature support than desktop browsers, and tend to perform worse than browsers on desktop devices. Finally, there were flaky failures.

Reduced feedback time using the proposed techniques. To determine why our prioritization techniques' APFD_C values are higher than those for the baseline techniques, we investigated how feedback time is actually reduced for each failure. The boxplot on the left side of Figure 5 shows the reduction in feedback time for failures for each of the experiment objects other than Lodash, when using the ML-based technique. (Lodash is omitted because we do not have time information for it). The ML-based technique provides feedback for each failure between 0.86 and 1.19 minutes faster than when no ordering is used, and between 0.93 and 3.12 minutes faster than when random ordering is used.

For Lodash, we estimated job time as the job time for accessible builds (nine builds), and measured reduction in feedback time based on that. We estimated the job time for a browser environment as the average job time for the browser environment with the same module name and browser type. As a result, the reduction in feedback time was estimated at 10.77 minutes. Given that Lodash's median number of browser environments is 99, we expect that the larger the number of browser environments, the greater the reduction in feedback time that can be achieved.

Table 4: Causes of Failures

Cause	Cause detail
A feature is not fully implemented	IE11 does not implement string description for `Map`, `Set` and `WeakMap` objects [55]
A feature that is not implemented is used in test code	A test case uses `Array.prototype.indexOf` function but the function is not implemented in IE8 [2]
A feature is implemented differently	`hasOwnProperty` is a non-enumerable property in IE8 while it is enumerable in other browsers [3]
A feature is not implemented in a mobile browser	`TypedArray.prototype.subarray()` is supported from Safari on iOS 4.2 [26]
Device performance is different	Performance of Safari on iOS 8 is slower than desktop browser, so the browser cannot finish rendering an updated page in a given time in test code [6]
Flaky failure	iOS 7.1 flakiness [7]

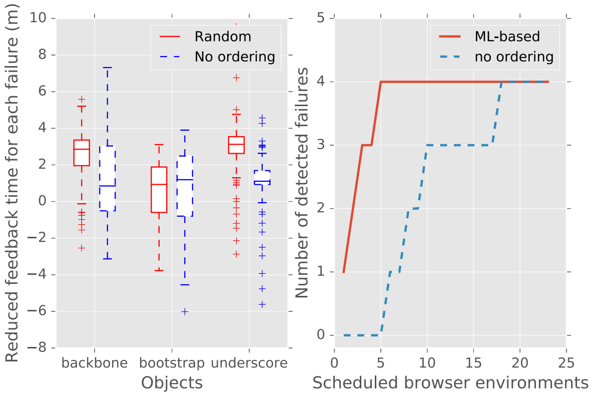


Figure 5: Left: Reduced feedback time for failures. Right: an example showing different feedback times in `Underscore`

The right side of Figure 5 presents an example showing how faster feedback is achieved on failures for one particular build of `Underscore`. The ML-based technique detected all four failed browser environments by testing the application on the first five browser environments, whereas the use of no ordering did not detect all failures.

Prioritization time analysis. Each prioritization technique uses computation time to prioritize browser environments. Recently-failed-based techniques spend time comparing browser environments in a build with cached browser environments, and the ML-based technique spends time learning its prediction model and predicting the probability of failure on a browser environment. Table 5 shows the time required to apply each technique other than the hybrid techniques to browser environments. Each technique was able to prioritize browser environments in less than 0.03 seconds for each of the applications. Naturally, the more sophisticated similarity-based matching and ML-based techniques required more time than the less sophisticated exact matching-based and failure frequency-based techniques, but the times were still small. Where hybrid techniques are concerned, the time required to prioritize browser environments can be estimated as the sum of the time taken to run the two techniques the hybrid is composed of.

Table 5: Runtime Costs of Prioritization Techniques

	`Backbone`	`Bootstrap`	`Lodash`	`Underscore`
M3	0.007	0.007	0.007	0.007
M4	0.023	0.023	0.023	0.023
M5	0.006	0.006	0.006	0.006
M6	0.012	0.012	0.012	0.012

Parallel test execution. Test cases can be run in parallel in different browser environments, and prioritizing browser environments is still necessary. In fact, Sauce Labs allows developers to set priorities for browser environments. When Sauce Labs runs out of available virtual machines or a concurrency limit given to a developer is exceeded, browser environments with higher priority take precedence over browser environments with lower priority.

To quantify the benefits of scheduling test environments in the presence of parallel test execution, we modified the existing APFD equation [16] as follows:

$$APFD_p = 1 - \frac{\lceil \frac{C_1}{p} \rceil + \lceil \frac{C_2}{p} \rceil + \ldots + \lceil \frac{C_m}{p} \rceil}{nm} + \frac{1}{2n}$$

The variables used here are the same as those used earlier, with the exception of p. Here, p represents the number of parallel executions. When $p = 1$, the modified equation becomes the same as the original equation. When we assume that the testing time under each browser environment is the same, all the C_m values are reduced by p times because p sessions can be run in parallel, and test failures can be detected p times faster.

We compare one of the hybrid approaches (M8) to no ordering and random ordering in a parallel environment. Figure 6 shows the APFD_p results as the number of parallel executions increases. Each green line with squares indicates our approach, each blue line (with points denoted by stars) indicates no ordering, each red line (with points denoted by plus signs) indicates random ordering, and each green line (with points denoted by squares) indicates our hybrid approach. For all the approaches, the APFD_p values increase as the degree of concurrency increases. In addition, the APFD_p values for both approaches converge as the degree of concurrency increases. Although there is a difference in degree, our approach outperforms the baseline approach even in the presence of parallel executions.

Exact matching-based prioritization technique in `Lodash`. M3 (Exact-matching) has an APFD_C value that is 12 percentage points lower than M1 (no ordering). There are two possible reasons for this result. First, the number of properties for `Lodash` browser environments is more than that for other applications, so it can be more difficult to obtain exact matchings for `Lodash`. Second, for `Lodash`, the browser version in the list of environments was updated more often than in the other applications. According to Lodash’s maintainer, their builds consider current and previous versions of most browsers [27]. Consequently, M3 often fails to assign higher priority values to environments that partially match previously failed environments. However, other techniques assign higher priority scores regarding the partial matching.

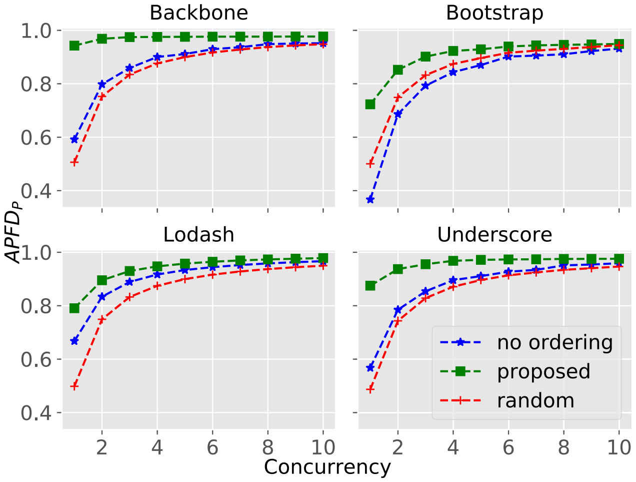


Figure 6: $APFD_p$ according to concurrency

6 RELATED WORK

Detecting regression failures under multiple web-browser environments is closely related to detecting cross-browser incompatibility (XBI) problems. Many researchers have attempted to automatically detect cross-browser incompatibility problems [13, 29, 32, 47, 53] or match compatible features across different web-browser environments [48]. To date, however, this work has not specifically considered regression testing. In addition, existing research considers XBIs at the DOM level to ensure consistent appearance of HTML pages, but does not consider XBIs at the unit-function level. It is important to check XBIs at the unit-function level because unit testing of functions can isolate root causes of errors more easily than other types of tests; moreover, for open source web applications, functions are usually called by various other web applications.

Testing of web-browser environments is also similar to the testing of configurable software. Configurable software is software that can be customized by users through selections of options; such software must be tested on various configurations. Cohen et al. [12] present a study in which a single version of a configurable web browser (Firefox) is tested. This study quantifies the effects of changing configurations on fault detection effectiveness and the code coverage achieved by test suites. Qu et al. [42] present a regression testing technique that prioritizes configurations of a configurable software system, and show that scheduling configurations can result in earlier fault detection. These studies, however, do not consider faults in client-side web applications. In addition, these approaches require code coverage information, which can be expensive to obtain in a rapid development environment.

The issues for testing of Software Product Lines (SPLs) are also closely related to issues for testing web applications across browser environments. To reduce the costs of testing SPLs, researchers have considered techniques that select subsets of variation-point combinations. One approach utilizes combinatorial test designs that select only pair-wise combinations, thereby reducing the number of combinations [41]. A second approach extends combinatorial test designs to apply a specific testing method to SPL models, such as a feature model or orthogonal variability model [31]. Similar to our work, these approaches attempt to reduce testing costs in situations where many combinations of variation points are present. Our approach, however, focuses on scheduling browser environments.

Our prioritization techniques are based on considering recently or frequently failing testing environments. This idea is closely related to a defect prediction work by Kim et al. [24] that considers recent faults. Kim et al. utilize a cache to store fault-prone software entities, locations of fixed faults, and other locations including the locations of recently added or changed code. They show that when 10% of the source code files are in a cache, 73%–95% of faults occur in these files, thus demonstrating the usefulness of cached information in fault finding. Engstrom et al. [18] also study the use of cached information in regression testing. Their approach, however, focuses on regression test selection rather than test-case prioritization, and they do not consider the XBI problem. Some defect prediction techniques use machine learning algorithms; these build machine learning classifiers that automatically learn patterns in buggy files or APIs, and predict defective files or APIs [19, 23]. This work does not, however, consider regression testing.

Several regression testing techniques have made use of previous build history information [17, 22, 30]. These approaches are based on the idea that some test suites are more likely to reveal failures than others. In particular, Elbaum et al. [17] use time windows, which are closely related to caches, to track recent test suites that revealed failures. In that work, time windows are used for test suite selection and prioritization. In contrast, our work considers recently and frequently failed web-browser environments.

7 CONCLUSION

We have presented cost-effective techniques for prioritizing browser environments when regression testing web applications, that function by considering recently and frequently failed browser environments. We have empirically compared our approaches with baseline approaches, including the original ordering and random orderings of browser environments for several web applications. We show that the rate of fault detection ($APFD_C$) achieved by our techniques is better than that of the baseline approaches, and the improvement is statistically significant.

We intend to conduct additional studies on more web applications. In addition, if we can obtain information about failure severities, we can incorporate this into techniques. To do this we can convert failure severity to a weight value, and multiplied the priority score of each browser environment generated by each approach by that weight. We also plan to use other software artifacts to schedule browser environments, such as information on updated code. By combining such data with build history data, we hope to be able to create more cost-effective prioritization techniques.

ACKNOWLEDGMENTS

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2017M3C4A7066210).

REFERENCES

[1] Backbone 2018. Backbone. https://github.com/jashkenas/backbone. (2018).

[2] Backbone 2018. Example of failure cause in Backbone. https://github.com/jashkenas/backbone/pull/3998. (2018).

[3] Backbone 2018. Example of failure cause in Backbone. https://github.com/jashkenas/backbone/pull/4000. (2018).

[4] Christopher M Bishop. 2006. Pattern recognition. *Machine Learning* 128 (2006), 1–58.

[5] Bootstrap 2018. Bootstrap. https://github.com/twbs/bootstrap. (2018).

[6] Bootstrap 2018. Example of failure cause in Bootstrap. https://github.com/twbs/bootstrap/issues/14851. (2018).

[7] Bootstrap 2018. Example of failure cause in Bootstrap. https://github.com/twbs/bootstrap/pull/13423. (2018).

[8] Bower 2017. Bower. https://bower.io/search/. (2017).

[9] Browser environment prioritization 2018. Implementation and data. http://webeng.kaist.ac.kr/webengpress/browser-env-prio/. (2018).

[10] BrowserStack 2018. BrowserStack. https://www.browserstack.com. (2018).

[11] CLOC 2018. CLOC. https://www.npmjs.com/package/cloc. (2018).

[12] Myra B Cohen, Joshua Snyder, and Gregg Rothermel. 2006. Testing across configurations: implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes* 31, 6 (2006), 1–9.

[13] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. 2012. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*. ACM, 11–15.

[14] David M Diez, Christopher D Barr, and Mine Cetinkaya-Rundel. 2012. *OpenIntro statistics*. Vol. 12. CreateSpace.

[15] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 329–338.

[16] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 28, 2 (2002), 159–182.

[17] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 235–245.

[18] Emelie Engström, Per Runeson, and Greger Wikstrand. 2010. An empirical evaluation of regression testing based on fix-cache recommendations. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 75–78.

[19] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 789–800.

[20] Sylvain Hallé, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. 2015. Testing Web Applications Through Layout Constraints. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–8.

[21] Mark Harman and Nadia Alshahwan. 2008. Automated session data repair for web application regression testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE, 298–307.

[22] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24rd International Conference on Software Engineering (ICSE)*. IEEE, 119–129.

[23] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. 2015. REMI: Defect prediction for efficient API testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 990–993.

[24] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 489–498.

[25] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using multi-locators to increase the robustness of web test cases. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[26] Lodash 2018. Example of failure cause in Lodash. https://github.com/lodash/lodash/commit/410969. (2018).

[27] Lodash 2018. Issue report in Lodash. https://github.com/lodash/lodash/issues/2898. (2018).

[28] Lodash 2018. Lodash. https://github.com/lodash/lodash. (2018).

[29] Sonal Mahajan and William GJ Halfond. 2014. Finding html presentation failures using image comparison techniques. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 91–96.

[30] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 540–543.

[31] John D. McGregor. 2001. *Testing a Software Product Line.* Software Engineering Institute Technical Report CMU/SEI-2001-TR-022. Carnegie Mellon University, Pittsburgh, PA.

[32] Ali Mesbah and Mukul R Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 561–570.

[33] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 67–78.

[34] Shabnam Mirshokraie and Ali Mesbah. 2012. JSART: JavaScript assertion-based regression testing. In *Proceedings of the International Conference on Web Engineering (ICWE)*. Springer, 238–252.

[35] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. Pythia: Generating Test Cases with Oracles for JavaScript Applications. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE), New Ideas Track*. IEEE, 610–615.

[36] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSeft: Automated JavaScript unit test generation. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[37] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2016. Atrina: Inferring Unit Oracles from GUI Test Cases. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE, 330–340.

[38] Mootools 2018. Mootools. http://mootools.net. (2018).

[39] Mootools-core in Travis CI 2018. Mootools-core in Travis CI. https://travis-ci.org/mootools/mootools-core. (2018).

[40] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An empirical study of client-side JavaScript bugs. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 55–64.

[41] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated incremental pairwise testing of software product lines. In *Proceedings of the International Conference on Software Product Lines*. Springer, 196–210.

[42] Xiao Qu, Myra B Cohen, and Gregg Rothermel. 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 75–86.

[43] QUnit 2018. QUnit. https://qunitjs.com. (2018).

[44] John Resig. 2009. JavaScript Testing Does Not Scale. (Mar 2009). https://johnresig.com/blog/javascript-testing-does-not-scale/

[45] Danny Roest, Ali Mesbah, and Arie Van Deursen. 2010. Regression testing ajax applications: Coping with dynamism. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 127–136.

[46] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948.

[47] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2013. X-PERT: accurate identification of cross-browser issues in web applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, 702–711.

[48] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2014. Cross-platform feature matching for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 82–92.

[49] Sreedevi Sampath, Renee C Bryce, Gokulanand Viswanath, Vani Kandimalla, and A Gunes Koru. 2008. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE, 141–150.

[50] Sauce Labs 2018. Sauce Labs. https://saucelabs.com. (2018).

[51] Matthias Schur, Andreas Roth, and Andreas Zeller. 2013. Mining behavior models from enterprise web applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 422–432.

[52] Scipy. 2018. scipy.stats.f_oneway — SciPy v0.19.1 Reference Guide. (2018). Retrieved January 2, 2018 from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

[53] Nataliia Semenenko, Marlon Dumas, and Tõnis Saar. 2013. Browserbite: Accurate cross-browser testing via machine learning over image features. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 528–531.

[54] Travis CI 2018. Travis CI. https://travis-ci.org. (2018).

[55] Underscore 2018. Example of failure cause in Underscore. https://github.com/jashkenas/underscore/pull/2464. (2018).

[56] Underscore 2018. Underscore. https://github.com/jashkenas/underscore. (2018).

[57] Wikipedia. 2016. List of web browsers — Wikipedia, The Free Encyclopedia. (2016). https://en.wikipedia.org/w/index.php?title=List_of_web_browsers&oldid=753422787 [Online; accessed 7-December-2016].

[58] Wikipedia. 2017. Jaccard index — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Jaccard_index&oldid=788825557 [Online; accessed 21-August-2017].

[59] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. 2014. Virtual DOM coverage for effective testing of dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 60–70.