Regression Test Selection for Distributed Software Histories

Milos Gligoric[1], Rupak Majumdar[2], Rohan Sharma[1], Lamyaa Eloussi[1], and Darko Marinov[1]

[1] University of Illinois at Urbana-Champaign, USA
[2] Max Planck Institute for Software Systems, Germany
{gliga,sharma27,eloussi2,marinov}@illinois.edu, rupak@mpi-sws.org

Abstract. Regression test selection analyzes incremental changes to a codebase and chooses to run only those tests whose behavior may be affected by the latest changes in the code. By focusing on a small subset of all the tests, the testing process runs faster and can be more tightly integrated into the development process. Existing techniques for regression test selection consider two versions of the code at a time, effectively assuming a development process where changes to the code occur in a linear sequence.

Modern development processes that use *distributed* version-control systems are more complex. Software version histories are generally modeled as directed graphs; in addition to version changes occurring linearly, multiple versions can be related by other commands, e.g., branch, merge, rebase, cherry-pick, revert, etc. This paper describes a regression test-selection technique for software developed using modern distributed version-control systems. By modeling different branch or merge commands directly in our technique, it computes safe test sets that can be substantially smaller than applying previous techniques to a linearization of the software history.

We evaluate our technique on software histories of several large open-source projects. The results are encouraging: our technique obtained an average of 10.89× reduction in the number of tests over an existing technique while still selecting all tests whose behavior may differ.

1 Introduction

Regression testing [22, 36, 37] reruns previously completed tests whenever a change is made to a piece of software, to ensure that the change has not affected the outcome of those tests. Regression testing can be expensive if the test suite is large and tests take a long time to run. Therefore, substantial research has focused on speeding up regression testing by selecting an adequate subset of tests (with several extensive surveys [5,11,37] on the topic). These test-selection techniques are usually based on computing changes between two program versions[3], the "old" and the "new" versions, and using a fast syntactic algorithm to

[3] We use the term "version" for what version-control systems often call "revision".

identify the subset of tests whose behavior *may* change between the old and new versions. Empirically, these techniques are effective in reducing the set of tests to be run and are widely used in companies such as Google [16] and Microsoft [34].

Existing test-selection techniques view software history as a linear sequence of commits to a centralized version-control system (such as CVS or SVN). However, modern software development processes that use distributed version-control systems (DVCSs) do not match this simplistic view. Software version histories that use DVCSs, such as Git and Mercurial, are complex graphs of branches, merges, and rebases of the code that mirror more complex sharing patterns between developers. For example, Figure 1 shows a part of the Linux Kernel Git repository [25]: this software history is a complex graph, with multiple branches being merged. (There is a case in Linux where 30 branches are merged at once.) We empirically find that such complexities are not isolated to the Linux kernel development: most open-source codebases perform frequent merges. Section 4 reports detailed results for a number of open-source projects; we find about third of the commits to be merge-related.

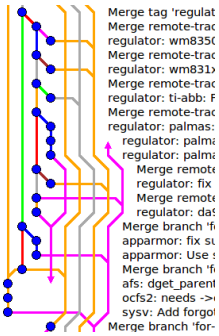
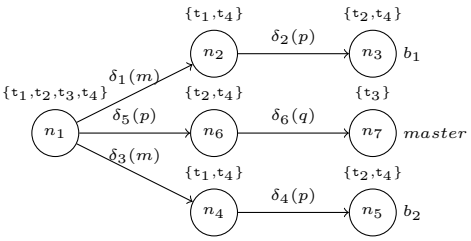


Fig. 1: Linux history

In this paper, we consider the problem of test selection for codebases that use DVCS commands. One possible baseline approach is to apply traditional test selection by picking an arbitrary linearization of the history. While this technique is *safe*, i.e., it does not miss tests whose outcome may be affected by the change, we empirically demonstrate that this approach can be very *imprecise*, i.e., it can select many tests whose outcome cannot be affected by the change. Instead, we propose a test-selection technique that explicitly takes into account the history graph of software versions. We have implemented our technique and show, through an evaluation on several open-source code repositories, that our technique selects on average an order of magnitude fewer tests than the baseline technique while still retaining safety.

We evaluate our technique both on real open-source code repositories that use DVCS and on distributed repositories that we systematically generate from projects that use a linear sequence of commits. We compare several options for selecting tests at each merge version of such repositories. These options have different trade-offs in terms of cost (how many traditional test selections need to be performed to compute the selected tests) and precision (how many tests are selected to be run, while maintaining safety). In particular, we describe a fast test-selection technique for code merges that does not require *any* test selection computation, but still achieves a reduction of 10.89× better than a baseline technique that performs one traditional test selection for a merge point, and only 2.78× worse than an expensive technique that performs one traditional test selection for each branch being merged.

The accompanying technical report [15] provides additional results, visualizations, and proofs.



(a) Example software history

		Code	Methods in C		
			m	p	q
Tests in T	t1	{m();}	✓	✗	✗
	t2	{p();}	✗	✓	✗
	t3	{q();}	✗	✗	✓
	t4	{m();p();}	✓	✓	✗

(b) Methods and tests in C and T

```
1  git init // initialize the repository
2  git add C // add C to the repository
3  git add T // add T to the repository
4  git commit -m 'C and T' // commit n1
5  git checkout -b b1 // go to a new branch 'b1'
6  δ1(m) // modify method 'm' in branch 'b1'
7  git commit -am 'Modified m' // commit n2
8  δ2(p) // modify method 'p' in branch 'b1'
9  git commit -am 'Modified p' // commit n3
10 git checkout master // go to 'master' branch
11 git checkout -b b2 // go to a new branch 'b2'
12 δ3(m) // modify method 'm' in branch 'b2'
13 git commit -am 'Modified m' // commit n4
14 δ4(p) // modify method 'p' in branch 'b2'
15 git commit -am 'Modified p' // commit n5
16 git checkout master // go to 'master' branch
17 δ5(p) // modify method 'p' in 'master' branch
18 git commit -am 'Modified p' // commit n6
19 δ6(q) // modify method 'q' in 'master' branch
20 git commit -am 'Modified q' // commit n7
```

(c) Sequence of commands that creates the history on the left

Fig. 2: Example of a software history and one potential sequence of changes and commands to create this history

2 Overview

We motivate regression test selection through an example session using Git [14], a popular DVCS.

Distributed software histories. Figure 2a visualizes a version history obtained by performing the sequence of Git commands from Figure 2c. First, we initialize the software history[4], add two files and make a commit n_1 with these files (lines 1-4). Figure 2b shows the abstract representation of the committed files `C` and `T`; file `C` ("*Code*") defines three methods `m`, `p`, and `q` that are checked by tests `t1`, `t2`, `t3`, and `t4` defined in file `T` ("*Test*"). Second, we create a new branch b_1 (line 5), make and commit changes to `m` (lines 6–7) and `p` (lines 8–9). Third, we create another branch b_2 (lines 10–11) and perform a similar sequence of commands as on the first branch (lines 12–15). Finally, we switch to the *master* branch (line 16) and perform a similar sequence of commands (lines 17–20). Although the sequence of commands is similar for each branch, we assume non-conflicting changes on different branches.

Figure 2b further shows which test executes which method; we will assume that we have available such a *coverage matrix* for every version in the software history. When a method changes, the tests that executed that method are called *modification-traversing* tests. We focus on modifications at a method level for simplicity; one can track coverage of other program elements as well [37].

[4] `git init` creates the initial node not shown in Figure 2a.

Traditional test selection. Traditional test selection takes as input an old version, a new version, and a coverage matrix for the old version, and returns a set of tests such that each test in the set either is new or traverses at least one of the changes made between the old and the new version. (We formally define test selection in Section 3.) Tests that traverse a change can be found from the coverage matrix by taking all the tests that have a checkmark ('✓') for any changed method (corresponding to the appropriate column in Figure 2b).

In our running example, all tests are new at n_1, thus all tests are selected. (Figure 2a indicates above each node the set of selected tests.) At version n_2, after modifying method `m`, test selection would take as input n_1 and n_2 and return tests that traverse the changed method. Based on our coverage matrix (Figure 2b), tests $\texttt{t}_1$ and $\texttt{t}_4$ should be selected. Following the same reasoning, we can obtain a set of selected tests for each version in the graph. For simplicity of exposition, we assume that the coverage matrix remains the same for all the versions. However, the matrix may change if a modification of any method leads to modification in the call graph. In case of a change, the matrix would be recomputed; however, note that for each test that is not selected (because it does not execute any changed method), the row in the coverage matrix would not change.

Test selection for distributed software histories. Test selection for distributed software histories has not been studied previously. We illustrate what the traditional test selection would select when a software history (Figure 2a) is extended by executing some of the commands available in DVCSs. Specifically, we show that a naive application of the traditional test selection leads to safe but imprecise results (i.e., selects too much), or requires several runs of traditional test-selection techniques, which introduces additional overhead and therefore reduces the benefits of test selection. We consider three commands: *merge*, *cherry-pick*, and *revert*.

Command: Merge. The merge command joins two or more development branches together. A merge without conflicts and any additional edits is called *auto-merge* and is the most common case in practice. Auto-merge has a property that the changes between the merge point and its parents are a subset of the changes between the lowest common ancestors [3,10] of the parents and the parents; we exploit this property in our technique and discuss it further in Section 3. If we execute `git merge` $\texttt{b}_1$ $\texttt{b}_2$ after the sequence shown in Figure 2c, while we are still on the *master* branch, we will merge branches b_1 and b_2 into a new version n_8 on the *master* branch; this version n_8 will have three parents: n_3, n_5, and n_7. The question is what tests to select to run at version n_8.

We propose multiple options (and Section 4 summarizes how to automatically choose between these options). First, we can use traditional test selection between the immediate dominator [1] of the new version (n_1) and the new version (n_8). In our example, the changes between these two versions modify all the methods, so test selection would select all four tests. The advantage of this option is that it runs traditional test selection only once, but there can be many changes, and therefore many tests are selected. Second, we can run the tradi-

tional test selection between the new version and each of its parents and take the intersection of the selected tests. In our example, we would run the traditional test selection between the following pairs: (n_3, n_8), (n_5, n_8), (n_7, n_8); the results for each pair would be: $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{t}_4\}$, $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{t}_4\}$, and $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_4\}$, respectively. The intersection of these sets gives the final result: $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_4\}$. The intuition is that the tests not in the intersection, $\{\mathtt{t}_3\}$, need not be run because their result for the new version, n_8, can be copied from at least one parent, in this case from n_7. Although the second option selects fewer tests, it requires running traditional test selection three times, which can lead to significant overhead. Third, we can collect tests that were modification-traversing on at least two branches (from the branching version at n_1 to the parents that get merged). In our example, we would select $\{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_4\}$. As opposed to previous options, this option requires *zero runs* of the traditional test-selection techniques. However, this option is only safe for *auto merge* and requires that the test selection results be stored for previous versions.

Command: Cherry-pick. Cherry-pick copies the changes introduced by some existing commit. If we execute `git cherry-pick` $\mathtt{n}_2$ after the sequence shown in Figure 2c, we will apply changes (δ_1) made between versions n_1 and n_2 on top of version n_7 in *master* branch (which is extended with a new version n_8). Naively applying the traditional test selection on versions n_7 and n_8 would select the same tests as at version n_2. However, test $\mathtt{t}_1$ does not need to be selected at n_8, as this test is not affected by changes on the *master* branch (on which the cherry-picked commit is applied). Therefore, the outcome of $\mathtt{t}_1$ at n_8 will be the same as at n_2.

Command: Revert. This command reverts some existing commits. If we execute `git revert` $\mathtt{n}_6$ after the sequence shown in Figure 2c, we will revert changes made between versions n_1 and n_6. The *master* branch will be extended with a new version n_8. Naively applying traditional test-selection techniques between versions n_7 and n_8 would select the same set of tests as at version n_6. Instead, if we consider the revert command being executed and changes being made, we can reuse the results of a test from version n_1 as long as the test is not modification-traversing for any other change after the version being reverted (n_6). In our example, we can see that the result of all tests can be reused, and therefore no test has to be selected.

To conclude, naively applying traditional test selection may lead to imprecise results and/or spend too much time on analysis. We believe that our technique, which reasons about the history and commands being executed, leads to a good balance between reduction (in terms of the number of tests being executed) and time spent on analysis.

3 Test Selection Technique

3.1 Modeling Distributed Software Histories

We model a distributed software history as a directed acyclic graph $G = \langle N, E \rangle$ with a unique root $n_0 \in N$ corresponding to the initial version. Each node $n \in N$

corresponds to a version, and each edge corresponds to the parent-child relation among versions. Each node is created by applying one of the DVCS commands to a set of parent nodes; we assume the command is known. (While the command that creates a node is definitely known at the point of creation, it is not usually kept in the DVCS and cannot always be uniquely determined from the history.) The functions $\mathsf{pred}(n) = \{n' \in N \mid \langle n', n\rangle \in E\}$ and $\mathsf{succ}(n) = \{n' \in N \mid \langle n, n'\rangle \in E\}$ denote the set of parents and children of version n, respectively. We write $n \preceq n'$ if there exists a directed path from n to n' or the two nodes are the same. We write $n \preceq^* n'$ to denote the set of all nodes between versions n and n': $n \preceq^* n' = \{n'' \mid n \preceq n'' \text{ and } n'' \preceq n'\}$. Similarly, we write $n \preceq^e n'$ to denote the set of all edges between versions n and n': $n \preceq^e n' = \{\langle n'', n'''\rangle \in E \mid n'', n''' \in n \preceq^* n'\}$. The function $\mathsf{sdom}(n) = \{n' \mid n_0 \preceq^e n' \cup n' \preceq^e n = n_0 \preceq^e n \text{ and } n \neq n'\}$ denotes the set of nodes that strictly dominate n. For $n \neq n_0$, the function $\mathsf{imd}(n)$ denotes the unique immediate dominator [1] of n, i.e., $\mathsf{imd}(n) = n'$ such that $n' \in \mathsf{sdom}(n)$ and $\nexists n'' \in \mathsf{sdom}(n)$ such that $n' \in \mathsf{sdom}(n'')$. The function $\mathsf{dom}(n, n')$ denotes the lowest common dominator of n and n', i.e., for a version n'' such that $\mathsf{pred}(n'') \supseteq \{n, n'\}$, $\mathsf{dom}(n, n') = \mathsf{imd}(n'')$. The function $\mathsf{lca}(n, n')$ denotes the lowest common ancestors [3,9,10] (also known as "merge-bases" or "best common ancestors" in Git terminology [13,20]) for two versions, i.e., $\mathsf{lca}(n, n') = \{n'' \mid n'' \preceq n \text{ and } n'' \preceq n' \text{ and } \nexists n''' \neq n'' \text{ such that } n''' \preceq n \text{ and } n''' \preceq n' \text{ and } n'' \preceq n'''\}$. (We illustrate the difference between lca and dom in the technical report [15].) The following property holds for all nodes:

$$\mathsf{dom}(n, n') \preceq \mathsf{lca}(n, n') \tag{1}$$

3.2 Test Selection for Two Versions

We formalize test selection following earlier work in the area [32, 37] and also model changes and modification-traversing tests. This section focuses on test selection between *two* software versions. Next sections present our technique for distributed software histories.

Let G be a distributed software history. For a version n, let $\mathcal{A}(n)$ denote the set of tests *available* at the version n. Let n and n' be two versions such that $n \preceq n'$. A *test selection* technique takes as input the versions n and n' and returns a subset $\mathcal{S}_{sel}(n, n')$ of $\mathcal{A}(n')$. Note that new tests, i.e, $\mathcal{A}(n') \setminus \mathcal{A}(n)$ are always in $\mathcal{S}_{sel}(n, n')$. A test-selection technique is *safe* [31] if every test in $\mathcal{A}(n') \setminus \mathcal{S}_{sel}(n, n')$ has the same outcome when run on the versions n and n'.

A trivially safe test-selection technique returns $\mathcal{A}(n')$. However, we are interested in selection techniques that select as small a subset as possible. One way to obtain a minimal set is to run each test in $\mathcal{A}(n')$ on the two versions and keep those that have different outcomes. However, the purpose of the test selection technique is to be more efficient than running all tests. A compromise between minimality and efficiency is provided by the notion of *modification-traversing* tests [32], which syntactically approximate the set of tests that may have a different outcome.

Let $\partial(n, n')$ be the set of static code changes between versions n and n' (which need not be parent-child versions). Various techniques compute these changes at various levels of granularity (e.g., basic blocks, statements, methods, or other program elements). By extension, we denote the set of changes on all edges from n to n' as

$$\partial^\star(n, n') = \bigcup_{\langle n'', n''' \rangle \in n \preceq^e n'} \partial(n'', n''')$$

We use the following property:

$$\partial(n, n') \subseteq \partial^\star(n, n') \tag{2}$$

It is not an equality because some changes can be reverted on a path from n to n', e.g., consider a graph with three versions n_1, n_2, and n_3, where all the changes between n_1 and n_2 are reverted between n_2 and n_3: the code at n_3 is exactly the same as the code at n_1, and therefore $\partial(n_1, n_3) = \{\}$.

A test is called *modification-traversing* if its execution on n executes any code element that is modified in n'. (Note that "modified" includes all the cases where the existing elements from n are *changed or removed* in n' or where new elements are *added* in n'.) We define a predicate $\varsigma(t, \partial)$ that holds if the test t is modification-traversing for any change in the given set of changes ∂. The predicate can be computed by tracking code paths during a test run and intersecting covered program elements with a syntactic difference between the two versions. We define a function $\mathsf{mt}(\mathcal{T}, \partial) = \{t \in \mathcal{T} \mid \varsigma(t, \partial)\}$ that returns every test from the set of tests $\mathcal{T}$ that is modification-traversing for any change in the set of changes ∂. Two properties that we will need later are that mt distributes over changes:

$$\mathsf{mt}(\mathcal{T}, \partial_1 \cup \partial_1) = \mathsf{mt}(\mathcal{T}, \partial_1) \cup \mathsf{mt}(\mathcal{T}, \partial_2) \tag{3}$$

and thus mt is monotonic with respect to the set of changes:

$$\partial \subseteq \partial' \text{ implies } \mathsf{mt}(\mathcal{T}, \partial) \subseteq \mathsf{mt}(\mathcal{T}, \partial') \tag{4}$$

Traditional test selection selects all modification-traversing tests from the old version that remain in the new version and the new tests from the new version:

$$\mathsf{tts}(n, n') = \mathsf{mt}(\mathcal{A}(n) \cap \mathcal{A}(n'), \partial(n, n')) \ \cup \ (\mathcal{A}(n') \setminus \mathcal{A}(n)) \tag{5}$$

As $\mathsf{pred}(n')$ is often a singleton $\{n\}$, we also write $\mathsf{tts}(\{n\}, n') = \mathsf{tts}(n, n')$.

Under the assumption that tests execute deterministically, test selection based on modification-traversing tests is provably safe [32,33].

3.3 Test Selection for Distributed Software Histories

Our technique for test selection takes as inputs (1) the software history $G = \langle N, E \rangle$ optionally annotated with tests selected at each version, (2) a specific version $h \in N$ that represents the latest version (which is usually called `HEAD` in

DVCS), and (3) optionally the DVCS command used to create the version h. It produces as output a set of selected tests $\mathcal{S}_{sel}(h)$ at the given software version. We define our technique and prove that it guarantees safe test selection.

Command: Commit. The h version has one parent, and the changes between the parent and h can be arbitrary, with no special knowledge of how they were created. The set of selected tests can be computed by applying the traditional test selection between the h version and its parent:

$$\mathcal{S}_{commit}(h) = \mathsf{tts}(\mathsf{pred}(h), h) \tag{6}$$

Command: Merge. *Merge* joins two or more versions and extends the history with a new version that becomes h. We propose two options to compute the set of selected tests at h: the first is fast but possibly imprecise, the second is slower but more precise.

Option 1: This option performs the traditional test selection between the immediate dominator of h and h itself:

$$\mathcal{S}^1_{merge}(h) = \mathsf{tts}(\mathsf{imd}(h), h) \tag{7}$$

This option is fast: it computes only one traditional test selection, even if the merge has many parents. However, the number of modifications between the two versions being compared can be large, leading to many tests being selected unnecessarily. Our empirical evaluation in Section 4 shows that this option indeed selects too many tests, discouraging the straightforward use of this option.

Option 2: This option performs one traditional test selection between each parent of the merged version and the merged version h itself, and then intersects the resulting sets:

$$\mathcal{S}^k_{merge}(h) = \bigcap_{n \in \mathsf{pred}(h)} \mathsf{tts}(n, h) \tag{8}$$

This option can be more precise, selecting substantially fewer tests. However, it has to run k traditional test selections for k parents.

Theorem 1 *$\mathcal{S}^1_{merge}(h)$ and $\mathcal{S}^k_{merge}(h)$ are safe for every merge version h.*

Command: Automerge. A common special case of *merge* is *auto merge*, where versions are merged automatically without any manual changes to resolve conflicts. (Using the existing DVCS commands can quickly check if a merge is an auto merge.) Empirically (see Figure 3), auto merge is very common: on average over 90% of versions with more than one parent are auto merges.

The key property of auto merge is that the merged code version has a union of all code changes from all branches but has only those changes (i.e., no other manual changes). Formally, given k parents $p_1, p_2, \ldots p_k$ that get merged into a new version h, the changes from each parent p to the merged version h reflect the changes on all the branches for different parents:

$$\partial(p, h) = \bigcup_{p' \in \mathsf{pred}(h), p' \neq p} \; \bigcup_{l \in \mathsf{lca}(p, p')} \partial(l, p') \tag{9}$$

The formula uses lca because of the way Git auto merges branches [13, 20].

For auto merge, we give a test-selection technique, $\mathcal{S}^0_{merge}$, that is based entirely on the software history up to the parents being merged and does not require running any traditional test selection between pairs of code versions at the point of merge (although it assumes that test selection was performed on the versions up to the parents of the merge). The set of selected tests consists of (1) existing tests (from the lowest common dominator of two (different) parents of h) affected by changes on at least two different branches being merged (because the interplay of the changes from various branches can flip the test outcome):

$$S_{aff}(h) = \bigcup_{p,p' \in \mathsf{pred}(h), p \neq p', d = \mathsf{dom}(p,p')} \left(\bigcup_{n \in d \preceq^* p \setminus \{d\}} S_{sel}(n) \right) \cap \left(\bigcup_{n \in d \preceq^* p' \setminus \{d\}} S_{sel}(n) \right) \tag{10}$$

and (2) new tests available at the merge point but not available on all branches:

$$S_{new}(h) = \mathcal{A}(h) \setminus \bigcap_{p'' \in \mathsf{pred}(h)} \mathcal{A}(p'') \tag{11}$$

Finally, $\mathcal{S}^0_{merge}(h) = S_{aff}(h) \cup S_{new}(h)$. The advantage of this option is that it runs *zero* traditional test selections. One disadvantage is that it could select more tests than $\mathcal{S}^k_{merge}$. Another disadvantage is that it requires storing tests selected at each version.

Theorem 2 *$\mathcal{S}^0_{merge}(h)$ is safe for every* auto *merge version h.*

Intuitively, $\mathcal{S}^0_{merge}$ is safe because a test that is affected on only one branch need not be rerun at the merge point: it has the same result at that point as on that one branch. The proof is in the technical report [15].

Command: Cherry-pick. *Cherry-pick* reapplies the changes that were performed between a commit n_{cp} and one of its parents $n'_{\mathsf{cp}} \in \mathsf{pred}(n_{\mathsf{cp}})$ (the parent can be implicit for non-merge n_{cp}), and extends the software history (on the branch where the command is applied) with a new version h. We propose two options to determine the set of selected tests at h. The first option uses the general selection for a commit (the traditional test selection between the current node and its parent): $\mathcal{S}^1_{cherry}(h) = \mathsf{tts}(\mathsf{pred}(h), h)$.

The second option, called $\mathcal{S}^0_{cherry}$, does not require running traditional test selection, but is safe only for *auto cherry-pick*. This option selects each test that satisfies one of the following three conditions: (1) tests selected between n'_{cp} and n_{cp} as well as between the point p at which cherry-pick is applied ($\{p\} = \mathsf{pred}(h)$) and $d = \mathsf{dom}(p, n'_{\mathsf{cp}})$, (2) tests selected between n'_{cp} and n_{cp} and also selected before n'_{cp} up to d, and (3) new tests at n_{cp}.

$$\begin{aligned} \mathcal{S}^0_{cherry}(h) = (S_{sel}(n'_{\mathsf{cp}}, n_{\mathsf{cp}}) \cap ((\cup_{n \in d \preceq^* p \setminus \{d\}} S_{sel}(n)) \cup (\cup_{n \in d \preceq^* n'_{\mathsf{cp}} \setminus \{d\}} S_{sel}(n)))) \\ \cup (\mathcal{A}(n_{\mathsf{cp}}) \setminus \mathcal{A}(n'_{\mathsf{cp}})) \end{aligned} \tag{12}$$

The intuition for (1) is that the combination of changes that affected tests on both branches, from d to p and from d to $n'_{\texttt{cp}}$, may lead to different test outcomes. The intuition for (2) is that changes before $n_{\texttt{cp}}$ may not exist in the branch on which the cherry-pick is applied and so the outcome of these tests may change. If neither (1) nor (2) hold, the test result can be copied from $n_{\texttt{cp}}$ itself. The formula for cherry pick is similar to that for auto merge but applies to only one commit being cherry picked rather than to an entire branch being merged.

Command: Revert. *Revert* computes inverse changes of some existing commit $n_{\texttt{re}}$ and extends the software history by applying those inverse changes to create a new version that becomes h. (Reverting a merge creates additional issues that we do not handle specially: one can always run the traditional test selection.) Similar to cherry-pick, we propose two options to determine the set of selected tests. The first option is a naive application of the traditional test selection between h and its parent, i.e., $\mathcal{S}^1_{revert}(h) = \mathsf{tts}(\mathsf{pred}(h), h)$.

The second option, called $\mathcal{S}^0_{revert}$, does not run traditional test selection, but is safe only for *auto revert*. It selects each test that satisfies one of the following three conditions: (1) tests selected between $n_{\texttt{re}}$ and its parent ($\{p'\} = \mathsf{pred}(n_{\texttt{re}})$) as well as before the point to which the revert is applied ($\{p\} = \mathsf{pred}(h)$) up to their dominator ($d = \mathsf{dom}(p, p')$), (2) tests selected between $n_{\texttt{re}}$ and its parent p' and also selected before the point that is being reverted (p') up to d, and (3) tests that were deleted at the point being reverted (such that in the inverse change tests are added):

$$\begin{aligned}\mathcal{S}^0_{revert}(h) = (S_{sel}(p', n_{\texttt{re}}) \cap ((\cup_{n \in d \preceq^* p \setminus \{d\}} S_{sel}(n)) \cup (\cup_{n \in d \preceq^* p' \setminus \{d\}} S_{sel}(n)))) \\ \cup (\mathcal{A}(p') \setminus \mathcal{A}(n_{\texttt{re}}))\end{aligned} \tag{13}$$

Intuitively, revert is an inverse of cherry-pick and safe for the same reasons: the tests that are not selected would have the same outcome at the h version as at the version prior to $n_{\texttt{re}}$.

4 Evaluation

We performed several experiments to evaluate the effectiveness of our technique. First, we demonstrate the importance of having a test-selection technique for distributed software histories. Second, we evaluate the effectiveness of our test-selection technique by comparing the number of tests selected using $\mathcal{S}^1_{merge}$, $\mathcal{S}^k_{merge}$, and $\mathcal{S}^0_{merge}$ on a number of software histories (both real and systematically generated), i.e., we consider how much test selection would have saved had it been run on the versions in the history. Third, we compare $\mathcal{S}^1_{cherry}$ and $\mathcal{S}^0_{cherry}$ on a number of real cherry-pick commits.

Real software histories are highly non-linear. We collected statistics for software histories of several large open-source projects that use Git. To check whether software histories are non-linear across many project types, we chose projects from different domains (e.g., Cucumber is a tool for running acceptance tests, JGit is a pure Java implementation of the Git version-control system, etc.),

Project	SHA	Size MB	Authors†	(C)ommits	(M)erges	(R)ebases†	Cherrypicks & Reverts (CR)	(M+R+CR) /C	M/C on master	Auto-merges %
Activator	a3bc65e	1.2	14	1499	446	10	29	32.35	93.93	95.73
TimesSquare	d528622	0.31	22	145	50	1	1	35.86	65.71	96.00
Astyanax	ba58831	2.0	59	725	134	3	14	20.82	23.04	94.02
Bootstrap	c75f8a5	3.2	474	6893	1573	21	557	31.20	25.75	83.21
Cucumber	5416686	1.2	145	2495	413	21	148	23.32	15.92	77.48
Graphhopper	e2805e4	7.2	13	1265	59	3	59	9.56	3.64	55.93
JGit	7995d87	9.0	83	2801	615	774	24	50.44	33.35	97.48
LinuxKernel	e62063d	484.5	11133	400479	27472	151569	–	–	30.12	–
LinuxKVM	b796a09	406.2	8542	273639	17483	107768	–	–	8.92	–
Retrofit	5bd3c1e	0.62	61	631	216	4	2	35.18	58.54	99.07
Others (14)	-	231.28	2150	86380	14066	12928	3829	35.68	20.64	85.15
Min	-	0.31	13	145	50	1	0	9.30	3.64	55.93
Max	-	484.5	11133	400479	27472	151569	1973	63.37	93.93	100.00
Median	-	5.20	60.50	2175.00	373.00	19.49	34.50	31.77	27.03	94.62
Ari. mean	-	47.77	945.66	32373.00	2605.29	11379.25	211.95	31.76	34.05	90.25
Geo. mean	-	5.69	79.04	2275.60	415.71	45.60	21.11	18.91	20.49	51.93
Std. Dev.	-	121.71	2717.26	93955.04	6343.27	36281.83	447.38	14.19	26.56	10.30

† We use a heuristic to determine the number of authors and rebases

Fig. 3: Statistics for several projects that use Git

implemented in different languages, of various sizes, having different number of unit tests and developers. Figure 3 shows the collected statistics (in detail for 10 projects and averages for 14 others; we provide an extended table in the technical report [15]). The key column is (M+R+CR)/C that shows the ratio of the number of merges, rebases[5], cherry-picks, and reverts over the total number of commits for the entire software history. The ratio can be as high as 63.37% and is 31.76% on average. Stated differently, we may be able to improve test selection for about a third of the commits in an average DVCS history. Additionally, we collected a similar ratio only for the *master* branch, because most development processes run tests for all commits on that branch but not necessarily on other branches (e.g., see the Google process for testing commits [16]). While this ratio included only merges (and not rebases, cherry-picks, or reverts), its average is even higher for the *master* branch than for the entire repository (34.05% vs. 31.76%), which increases the importance of test selection for distributed software histories. Finally, to confirm that the ratio of merges is independent of the DVCS, we collected statistics on three projects that use Mercurial [27]—OpenJDK, Mercurial, and NetBeans—and the average ratio of merges was 20%, which is slightly lower than the average number for Git but still significant.

Implementation. We implemented a tool in Java to perform test selection proposed in Section 3. The tool is independent of the DVCS being used and

[5] Note that we approximate the number of rebases by counting commits with different author and committer field.

Subject	Available Tests		Total Execution [sec]	
	min	max	min	max
Cucumber (core)	156	308	10	14
Graphhopper (core)	626	692	14	20
JGit	2231	2232	106	116
Retrofit	181	184	10	10

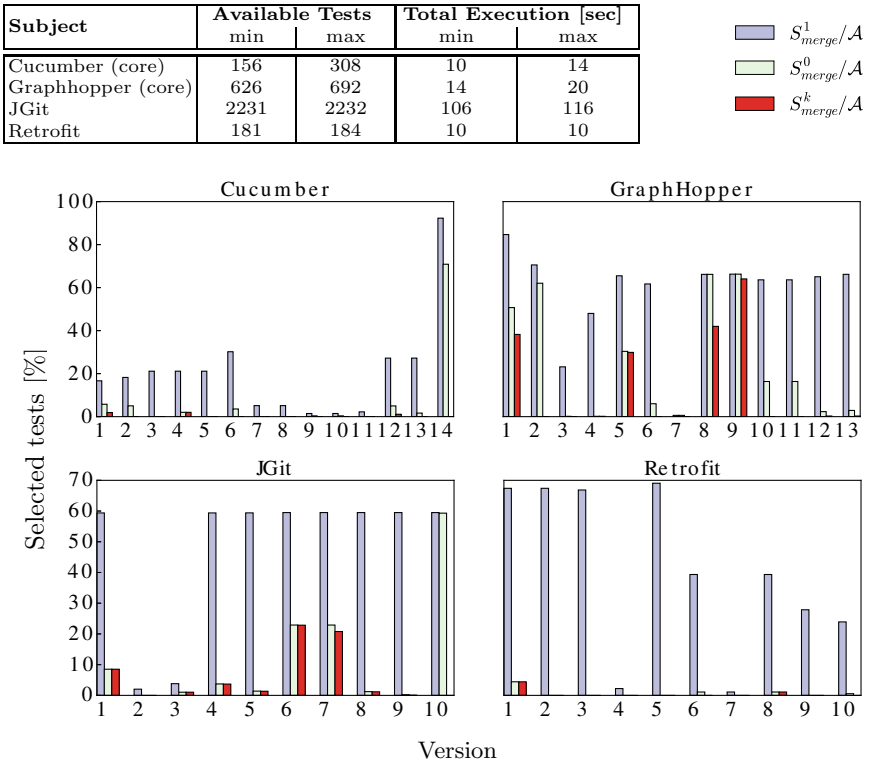
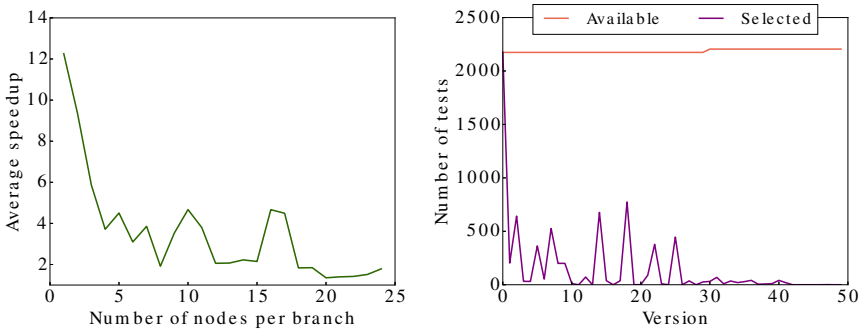


Fig. 4: Percentage of selected tests for real merges using various techniques

scales to quite large projects. Because any test-selection technique for distributed histories would require a traditional test selection between two versions (`tts`) for linear histories, and because there is no publicly available tool for the traditional test selection that scales to the large projects used in our study, we implemented a simple prototype tool for projects written in Java, following known results [5, 28,37,38]. Specifically, our `tts` computes changes and tracks executed code at the class level but still guarantees safety [28].

Real merges. Our first set of experiments evaluates our technique on the actual software histories. We used software histories of four large open-source projects (downloaded from GitHub): `Cucumber`, `GraphHopper`, `JGit`, and `Retrofit`. We selected these projects as their setup was not too complex[6], and they differ in size, number of authors, number of commits, and number of merges. For each project, we identify the last merge commit in the current software history and then run our test-selection tool on all the merge commits whose immediate dominator was in the 50 commits before the last merge commit.

[6] We have to build and run tests over a large number of commits, and dependencies in many real projects make running tests from older commits rather non-trivial.



(a) Average speedup across all subjects (b) `JFreeChart` history statistics

Fig. 5: (a) $\mathcal{S}^1_{merge}/\mathcal{S}^0_{merge}$ (speedup) for various numbers of commits in each branch constructed from linear software histories, (b) example linear history

At every merge, we run all three options—$\mathcal{S}^1_{merge}$, $\mathcal{S}^k_{merge}$, and $\mathcal{S}^0_{merge}$—and compare the number of tests they select. Testing literature [5, 12, 33, 35, 37] commonly measures the speedup of test selection as the ratio of the number of selected tests over the number of available tests ($\mathcal{S}_{sel}/\mathcal{A}$)[7]. In addition, Figure 4 reports the min and max number of available tests across the considered merge commits, and the min and max total time to execute these tests. All tests in these projects are unit tests and take a similar amount of time to execute, so computing the ratio of the numbers of tests is a decent approximation of the ratio of test execution times.

Figure 4 plots the results for these four projects. In most cases, $\mathcal{S}^k_{merge}$ and $\mathcal{S}^0_{merge}$ achieve substantial saving compared to $\mathcal{S}^1_{merge}$. (Calculated differently, the average speedup of $\mathcal{S}^0_{merge}$ over $\mathcal{S}^1_{merge}$ was 10.89× and $\mathcal{S}^k_{merge}$ over $\mathcal{S}^0_{merge}$ was 2.78×.) Although $\mathcal{S}^0_{merge}$ achieved lower saving than $\mathcal{S}^k_{merge}$ in a few cases (that we discuss below in more detail), it is important to recall that $\mathcal{S}^k_{merge}$ requires k runs of traditional test selection, while $\mathcal{S}^0_{merge}$ requires 0 runs.

We inspected in more detail the cases where $\mathcal{S}^k_{merge}/\mathcal{S}^0_{merge}$ was low. For `GraphHopper` (versions 2, 10, and 11), two branches have a large number of exactly the same commits (in particular, one branch has 11 commits and another has 10 of those 11 commits, which were created with some cherry-picking); when these branches were merged, the differences between the merged version and parents were rather small, resulting in a few tests being selected by $\mathcal{S}^k_{merge}$, although the changes between the parents and the dominator were rather big, resulting in many tests being selected by $\mathcal{S}^0_{merge}$. For `JGit` (version 10) and `Cucumber` (version 14), some new tests were added on one branch before merging it with another; $\mathcal{S}^0_{merge}$ is rather conservative in selecting (all) new tests, but new tests are not added frequently.

[7] For space reasons, we omit the set cardinality from the ratios.

Based on this inspection, we propose the following heuristic for choosing the best option for test selection at a merge version:

$$\mathcal{S}_{\texttt{merge}}(\texttt{h}) = \textbf{if } (\texttt{automerge } \& \texttt{ selection done at every commit})$$
$$\textbf{if } (\texttt{many new tests})\ \mathcal{S}^{\texttt{k}}_{\texttt{merge}}(\texttt{h})\ \textbf{else}\ \mathcal{S}^{0}_{\texttt{merge}}(\texttt{h})$$
$$\textbf{else if } (\texttt{short branches})\ \mathcal{S}^{1}_{\texttt{merge}}(\texttt{h})\ \textbf{else}\ \mathcal{S}^{\texttt{k}}_{\texttt{merge}}(\texttt{h})$$

Systematically generated merges. Our second set of experiments systematically compares the merge selection options on a set of graphs generated to represent *potential* software histories. Specifically, for a given number of nodes k, we generate all the graphs where nodes have the out degree (branching) of at most two, each branch contains between 1 and $k/2 - 2$ nodes, all the branches have the same number of nodes, and there are no linear segments on the master branch (except the last few nodes that remained after generating the branches). In other words, the generated graphs are diamonds of different length. For example for $k = 7$, we have the following two graphs: ·<:>·<:>· and ·<: =:>·−·. The total number of merges for the given number of nodes k is $\lfloor (k-1)/3 \rfloor + \lfloor (k-1)/5 \rfloor + \ldots + \lfloor (k-1)/(k-1) \rfloor$.

In addition to generating history graphs, we need to assign code and tests to each node of the graph. As random code or tests could produce too unrealistic data, we use the following approach: (1) we took the latest 50 versions of four large open-source projects with *linear* software histories: `JFreeChart` (SVN: 3021), `Goldman Sachs collections` (Git: 28070efd), `Ivy` (SVN: 1550956), and `Functor` (SVN: 1439120) (as an example, Figure 5b shows the number of available and selected tests for `JFreeChart`), (2) we assigned a version from the linear history to a node of the graph by preserving the relative ordering of versions such that a linear extension of the generated graph (partial order) matches the given linear history (total order). Using the above formula to calculate the number of merges for generated graph, for 50 versions, there are 68 merges (in 24 graphs); as we have four projects, the total number of merges is 272.

After the software histories are fully generated, we perform test selection on each of the graphs for each of the projects and collect the number of tests selected by all three options at each merge commit. As for the experiments on real software histories, we calculate the speedup as the ratio of the number of tests. Figure 5a shows the average speedup (across all four projects) for various number of nodes per branch. As expected, with more commits per branch, the speedup decreases, because the sets of changes on each branch become bigger and thus their intersection (as computed by our $\mathcal{S}^0_{merge}$ option) becomes larger. However, the speedup remains high for quite long branches. In fact, this speedup is likely an under-approximation of what can be achieved in real software projects because the assignment of changes across branches may not be representative of actual software histories: many related changes may be sprinkled across branches, which leads to a smaller speedup. Also, linear software histories are known to include more changes per commit [2]. We can see from the comparison of absolute values of the speedup in Figure 5a and Figure 4 that real software histories have an even higher speedup than our generated histories.

Real cherry-picks. We also compared $\mathcal{S}^1_{cherry}$ and $\mathcal{S}^0_{cherry}$ on 7 cherry-picks identified in the `Retrofit` project. (No other version from the other three projects

in our experiments used a cherry-pick command.) For 6 cases, $\mathcal{S}^0_{cherry}$ selected 7 tests more than $\mathcal{S}^1_{cherry}$, but all these tests were new. As mentioned, our current technique is rather conservative in selecting new tests; in future, we plan to improve our technique by considering coverage matrices across branches. In the remaining case, $\mathcal{S}^0_{cherry}$ selected 43% fewer tests (42 vs. 73 tests) than $\mathcal{S}^1_{cherry}$.

5 Related Work

Test selection is the most common optimization technique in regression testing [5, 11, 37]. Regression testing in general, and test selection in particular, have been studied for more than three decades [5,11,19,22,36,37] and are quite important in practice [16,34]. Prior research has investigated regression-testing techniques for various languages and domains [5,6,8,11,12,21,26,33–35,37], but all previous techniques considered only two program versions at a time. Most traditional test-selection techniques are safe; the key difference is how they define the coverage matrix and identify differences between software versions. For example, Rothermel and Harrold [33] presented a test-selection technique based on control-flow graphs. Zhang et al. [38] defined the coverage matrix on extended call graphs. Harrold and Soffa [18] and Gupta et al. [17] defined the coverage matrix on definition-use pairs. Several researchers [23, 24, 28] used a coverage matrix on modules (also known as "firewall" approach). Many other approaches have been proposed; for an overview, see the recent surveys [5,11,37].

Our technique for *distributed histories* is compatible with all these traditional techniques for *linear histories* as we abstract them in the core `mt` and `tts` functions. We are the first to propose a technique for safe test selection for distributed software histories; we use traditional test selection when a version is created by a commit command, and we reason about software history, modification-traversing tests, and commands being executed when a version is created by other DVCS commands (merge, cherry-pick, and revert).

Others [2, 4, 7, 29, 30] have observed several pitfalls of mining DVCS, e.g., DVCS commands are not recorded. We assume our test-selection technique is run at the time a new version is created (when the executed command is known).

6 Conclusions

We proposed the first test-selection technique that takes into account version histories arising out of distributed development, and proposed several options that trade off computation effort and precision. Our experimental results on real software histories demonstrate that our technique scales to large projects and achieves high effectiveness over a naive application of traditional test selection.

Acknowledgments. We thank Hsien-Chih Chang, Pranav Garg, Alex Gyori, Sarfraz Khurshid, P. Madhusudan, Aleksandar Milicevic, Ben Raichel, August Shi, and Mahesh Viswanathan for discussions, and the anonymous reviewers for comments. This research was partially supported by the US National Science Foundation Grant Nos. CNS-0958199 and CCF-1012759.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
2. Alali, A., Kagdi, H., Maletic, J.I.: What's a typical commit? A characterization of open source software repositories. In: International Conference on Program Comprehension. pp. 182–191 (2008)
3. Bender, M.A., Pemmasani, G., Skiena, S., Sumazin, P.: Finding least common ancestors in directed acyclic graphs. In: Symposium on Discrete Algorithms. pp. 845–853 (2001)
4. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The promises and perils of mining Git. In: International Working Conference on Mining Software Repositories. pp. 1–10 (2009)
5. Biswas, S., Mall, R., Satpathy, M., Sukumaran, S.: Regression test selection techniques: A survey. Informatica (Slovenia) 35(3), 289–321 (2011)
6. Briand, L., Labiche, Y., He, S.: Automating regression test selection based on UML designs. Information and Software Technology 51(1), 16–30 (2009)
7. Brindescu, C., Codoban, M., Shmarkatiuk, S., Dig, D.: How do centralized and distributed version control systems impact software changes? In: International Conference on Software Engineering (2014), to appear
8. Chittimalli, P.K., Harrold, M.J.: Regression test selection on system requirements. In: India Software Engineering Conference. pp. 87–96 (2008)
9. Czumaj, A., Kowaluk, M., Lingas, A.: Faster algorithms for finding lowest common ancestors in directed acyclic graphs. Theor. Comput. Sci. 380(1-2), 37–46 (2007)
10. Eckhardt, S., Mühling, A., Nowak, J.: Fast lowest common ancestor computations in dags. In: Annual European Symposium on Algorithms. vol. 4698, pp. 705–716. Springer Berlin Heidelberg (2007)
11. Engström, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Inf. Softw. Technol. 52(1), 14–30 (2010)
12. Engström, E., Skoglund, M., Runeson, P.: Empirical evaluations of regression test selection techniques: a systematic review. In: International Symposium on Empirical Software Engineering and Measurement. pp. 22–31 (2008)
13. git-merge-base, https://www.kernel.org/pub/software/scm/git/docs/git-merge-base.html
14. Git home page, http://git-scm.com/
15. Gligoric, M., Majumdar, R., Sharma, R., Eloussi, L., Marinov, D.: Regression test selection for distributed software histories. Technical report (2014), https://www.ideals.illinois.edu/handle/2142/49112
16. Gupta, P., Ivey, M., Penix, J.: Testing at the speed and scale of Google (Jun 2011), http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html
17. Gupta, R., Harrold, M.J., Soffa, M.L.: Program slicing-based regression testing techniques. Softw. Test., Verif. Reliab. 6(2), 83–111 (1996)
18. Harrold, M.J., Soffa, M.L.: Interprocedual data flow testing. In: Third Symposium on Software Testing, Analysis, and Verification. pp. 158–167 (1989)
19. Harrold, M., Soffa, M.: An incremental approach to unit testing during maintenance. In: International Conference on Software Maintenance. pp. 362–367 (1988)
20. How does merging work?, http://cbx33.github.io/gitt/afterhours4-1.html
21. Jones, J., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. Transactions on Software Engineering 29, 195–209 (2003)

22. K.F. Fischer, F. Raji, A.C.: A methodology for retesting modified software. In: National Telecommunications Conference (1981)
23. Kung, D.C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y.: Class firewall, test order, and regression testing of object-oriented programs. Journal of Object-Oriented Programming 8(2), 51–65 (1995)
24. Leung, H.K.N., White, L.: Insights into regression testing. In: International Conference on Software Maintenance. pp. 60–69 (1989)
25. LinuxKernel Git repository, git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
26. Memon, A.M., Soffa, M.L.: Regression testing of GUIs. In: International Symposium on Foundations of Software Engineering. pp. 118–127 (2003)
27. Mercurial home page, http://mercurial.selenic.com/
28. Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. In: International Symposium on Foundations of Software Engineering. pp. 241–251 (2004)
29. Perez De Rosso, S., Jackson, D.: What's wrong with Git?: A conceptual design analysis. In: International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 37–52 (2013)
30. Rigby, P., Barr, E., Bird, C., Devanbu, P., German, D.: What effect does distributed version control have on OSS project organization? In: International Workshop on Release Engineering. pp. 29–32 (2013)
31. Rothermel, G., Harrold, M.J.: A safe, efficient algorithm for regression test selection. In: Conference on Software Maintenance. pp. 358–367 (1993)
32. Rothermel, G., Harrold, M.J.: A framework for evaluating regression test selection techniques. In: International Conference on Software Engineering. pp. 201–210 (1994)
33. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. Trans. Softw. Eng. Methodol. 6(2), 173–210 (1997)
34. Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: International Symposium on Software Testing and Analysis. pp. 97–106 (2002)
35. Willmor, D., Embury, S.M.: A safe regression test selection technique for database driven applications. In: International Conference on Software Maintenance. pp. 421–430 (2005)
36. Yau, S.S., Kishimoto, Z.: A method for revalidating modified programs in the maintenance phase. In: Signature Conference on Computers, Software, and Applications (1987)
37. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)
38. Zhang, L., Kim, M., Khurshid, S.: Localizing failure-inducing program edits based on spectrum information. In: International Conference on Software Maintenance. pp. 23–32 (2011)