Regression Mutation Testing

Lingming Zhang[1], Darko Marinov[2], Lu Zhang[3], Sarfraz Khurshid[1]

[1]Electrical and Computer Engineering, University of Texas at Austin, USA
zhanglm@utexas.edu, khurshid@ece.utexas.edu
[2]Department of Computer Science, University of Illinois, Urbana, IL 61801, USA
marinov@illinois.edu
[3]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
zhanglu@sei.pku.edu.cn

ABSTRACT

Mutation testing is one of the most powerful approaches for evaluating quality of test suites. However, mutation testing is also one of the most expensive testing approaches. This paper presents *Regression Mutation Testing (ReMT)*, a new technique to speed up mutation testing for evolving systems. The key novelty of ReMT is to incrementally calculate mutation testing results for the new program version based on the results from the old program version; ReMT uses a static analysis to check which results can be safely reused. ReMT also employs a mutation-specific test prioritization to further speed up mutation testing. We present an empirical study on six evolving systems, whose sizes range from 3.9KLoC to 88.8KLoC. The empirical results show that ReMT can substantially reduce mutation testing costs, indicating a promising future for applying mutation testing on evolving software systems.

Categories and Subject Descriptors

D2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Regression Mutation Testing, Mutation Testing, Regression Testing, Software Evolution, Static Analysis

1. INTRODUCTION

Mutation testing [9, 15, 39, 45] is a methodology for assessing quality of test suites. The process of mutation testing has two basic steps. One, generate desired variants (known as *mutants*) of the original program under test through small syntactic transformations. Two, execute the generated mutants against a test suite to check whether the test suite can distinguish the behavior of the mutants from the original program (known as *killing the mutants*). The more mutants the test suite can kill, the more effective the test suite is considered to be.

Mutation testing is often viewed as the strongest test criterion in terms of characterizing high-quality test suites [3, 13]. Researchers have used mutation testing in numerous studies on software testing; see a recent survey by Jia and Harman [20]. Some studies have even shown that mutation testing can be more suitable than manual fault seeding in simulating real program faults for software testing experimentation [4, 12].

However, despite the potential mutation testing holds for software testing, it primarily remains confined to research settings. One of the main reasons is the costly analysis that underlies the methodology: the requirement to execute many tests against many mutants. A number of techniques aim to scale mutation testing, for example, by selecting a subset of mutants to generate instead of generating all of them [28, 31, 41, 45], by partially executing mutants to determine whether a test (weakly) kills a mutant [19, 42], and by executing some mutants in parallel [23, 26, 33]. While these techniques are able to reduce some cost of mutation testing, it still remains one of the most costly software testing methodologies.

Our key insight is that we can amortize this high cost of mutation testing in the context of software systems that undergo evolution by incrementally updating the results for successive applications of mutation testing. Real software systems undergo a number of revisions to implement bug fixes, add new features, or refactor existing code. An application of existing mutation testing techniques to an evolving system would require repeated, independent applications of the technique to each software version, inducing expensive costs for every version. Our approach utilizes the mutation testing results on a previous version to speed up the mutation testing for a subsequent version. Our approach opens a new direction for reducing the cost of mutation testing; it is orthogonal to the previous techniques for optimizing mutation testing, and it is applicable together with these previous techniques.

This paper presents *Regression Mutation Testing (ReMT)*, a novel technique that embodies our insight. ReMT identifies mutant-test pairs whose execution results (i.e., whether the test killed the mutant or not) on the current software version can be reused from the previous version without re-executing the test on the mutant. ReMT builds on the ideas from regression test selection techniques that traverse control flow graphs of two program versions to identify the set of *dangerous edges* which may lead to different test behaviors in the new program version [17, 34, 37]. More precisely, ReMT reuses a mutant-test result if (1) the execution of the test *does not* cover a dangerous edge *before* it reaches the mutated statement for the first time and (2) the execution of the test *cannot* reach a dangerous edge *after* executing the mutated statement. ReMT determines (1) with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA
Copyright 12 ACM 978-1-4503-1454-1/12/07 ...$10.00.

dynamic coverage and determines (2) with a novel static analysis for *dangerous-edge reachability* based on Context-Free-Language (CFL) reachability.

As an additional optimization to our core ReMT technique, we introduce *Mutation-specific Test Prioritization* (MTP). For each mutant, MTP reorders the tests that need to be executed based on their effectiveness in killing that mutant on previous versions and their coverage of the mutated statement. Combining ReMT with MTP can further reduce the time to kill the mutants.

Specifically, this paper makes the following contributions:

- **Regression Mutation Testing**. We introduce the idea of unifying regression testing with mutation testing—two well-researched methodologies that previous work has explored independently—to make mutation testing of evolving systems more efficient.

- **Technique**. We develop a core technique for regression mutation testing (ReMT) using dangerous-edge reachability analysis based on CFL reachability.

- **Optimization**. We introduce the idea of mutation-specific test prioritization (MTP) and present an MTP technique to optimize our core ReMT technique.

- **Implementation**. We implement ReMT and MTP on top of Javalanche [39], a recent mutation testing tool for Java programs with JUnit test suites.

- **Evaluation**. We present an empirical study on version repositories of six open-source Java programs between 3.9KLoC and 88.8KLoC. The results show that ReMT substantially reduce the costs of mutation testing on evolving systems.

2. PRELIMINARIES

This section describes some core concepts in mutation testing (Section 2.1) and regression testing (Section 2.2). It also provides some basic definitions that we use to present our Regression Mutation Testing (Section 2.3).

2.1 Mutation Testing

Mutation testing, first proposed by DeMillo et al. [9] and Hamlet [15], is a fault-based testing methodology that is effective for evaluating and improving the quality of test suites. Given a program under test, P, mutation testing uses a set of *mutation operators* to generate a set of *mutants* M for P. Each mutation operator defines a rule to transform program statements, and each mutant $m \in M$ is the same as P except for a statement that is transformed. Given a test suite T, a mutant m is said to be *killed* by a test $t \in T$ if and only if the execution of t on m produces a different result from the execution of t on P. Conceptually, mutation testing builds a mutant execution matrix:

DEFINITION 2.1. *A* ***mutant execution matrix*** *is a function $M \times T \rightarrow \{\mathtt{U}, \mathtt{E}, \mathtt{N}, \mathtt{K}\}$ that maps a mutant $m \in M$ and a test $t \in T$ to: (1)* `U` *if t has not been executed on m and thus the result is unknown, (2)* `E` *if the execution of t cannot reach the mutated statement in m (and thus m cannot be killed by test t), (3)* `N` *if t executes the mutated statement but does not kill m, and (4)* `K` *if t kills m.*

The aim of our ReMT technique is to speed up the computation of the mutant execution matrix for a new program version based on the mutant execution matrix for an old program version. Note that for the very first version the old matrix has all cells as `U` because there is no previous version. For future versions, the old matrix may in the limit be *full*, having no cell as `U`. However, our ReMT technique does *not* require such full matrices. Indeed, to compute the mutation score for a given program, for each mutant m, it suffices that the matrix has (1) *at least one* cell as `K` (while others can be `E`, `N`, or even `U`), or (2) *all* cells as `E` or `N` (indicating that the test suite T does not kill m).

Some existing mutation testing tools, such as Javalanche [39] and Proteum [7], support two mutation testing scenarios: (1) *partial mutation testing* – where a mutant is only run until it is killed and thus the matrix may have some `U` cells; and (2) *full mutation testing* – where a mutant is run against each test and thus the mutant execution matrix has no `U` cells. Our ReMT technique is applicable for both scenarios.

2.2 Regression Testing

A key problem studied in regression testing is Regression Test Selection (RTS): determine how changes between program versions influence regression tests and select to run only tests that are related to changes. RTS techniques [17, 34, 37] commonly use the control-flow graph (CFG) and its extended forms, e.g., the Java Interclass Graph [17], to represent program versions and analyze them. A typical RTS technique first traverses CFGs of two program versions using depth-first search (DFS) to identify the set of *dangerous edges*, $E_\triangle$, i.e., the edges which may cause the program behavior to change in the new program version. Then, for each test t in the regression test suite, the technique matches its coverage information on the old version with the set of dangerous edges $E_\triangle$ to determine whether t could be *influenced* by the dangerous edges.

Following previous work [17, 34], we consider RTS techniques that use inter-procedural CFGs:

DEFINITION 2.2. *An* ***inter-procedural CFG*** *of a program is a directed graph, $\langle N, E \rangle$, where N is the set of CFG nodes, and $E : N \times N$ is the set of CFG edges.*

Each inter-procedural CFG has several intra-procedural CFGs:

DEFINITION 2.3. *An* ***intra-procedural CFG*** *within an inter-procedural CFG $\langle N, E \rangle$ is a subgraph $\langle N_i, E_i \rangle$, where $N_i \subseteq N$ and $E_i \subseteq E$ denote edges that start from nodes in N_i. Each intra-procedural CFG has a unique entry node and a unique exit node.*

Note that E_i includes edges that are method invocation edges connecting invocation nodes in N_i with entry nodes of other intra-procedural CFGs, as well as edges that are return edges connecting the exit node with return nodes of other intra-procedural CFGs. Thus, $E_i \subseteq N_i \times N$. Moreover, each invocation node can be linked to different target methods based on the possible receiver object types, and thus each invocation edge is labeled with a run-time receiver object type to identify dangerous edges caused by dynamic dispatch changes.

Traditional RTS techniques [17, 34, 37] explore CFG nodes of two programs versions using DFS search to determine the equivalence of node pairs by examining the syntactic equivalence of the associated statements. They determine the set of dangerous edges:

DEFINITION 2.4. *The set of* ***dangerous edges*** *between two inter-procedural CFGs $\langle N, E \rangle$ and $\langle N', E' \rangle$ is the set of edges $E_\triangle \subseteq E$ whose target nodes have been changed to non-equivalent nodes or whose edge labels have been changed.*

2.3 Regression Mutation Testing

To reuse mutation testing results from an old program version for the new program version, ReMT maintains a mapping between the mutants of the two program versions. This mutant mapping is based on the CFG node mapping:

DEFINITION 2.5. *For two inter-procedural CFGs $\langle N, E\rangle$ and $\langle N', E'\rangle$, the **CFG node mapping** is defined as function* `mapN`*: $N' \to N \cup \{\bot\}$ that maps each node in N' to its equivalent node in N or to $\bot$ if there is no such equivalent node.*

Note that the node mapping is constructed during the DFS search by RTS for identifying dangerous edges.

The mapping between mutants of two program versions is defined as follows:

DEFINITION 2.6. *For two program versions P and P' and their corresponding sets of mutants M and M', **mutant mapping** between P and P' is defined as function* `mapM`*: $M' \to M \cup \{\bot\}$, that returns mutant $m \in M$ of P for mutant $m' \in M'$ of P', if (1) the mutated CFG node $n_{m'}$ of m' maps to the mutated CFG node n_m of m (i.e., $n_m = \texttt{mapN}(n_{m'})$) and (2) m' and m are mutated by the same mutation operator at the same location; otherwise,* `mapM` *returns $\bot$.*

The traditional RTS techniques [17,37] compute influenced tests by intersecting edges executed by the tests on the old program version with the dangerous edges. However, such computation of intersection for original, unmutated programs does *not* work for regression *mutation* testing, because the test execution path for each mutant may differ from the path for the original program. Therefore, for ReMT, we introduce a static analysis for checking the reachability of dangerous edges for each mutant when it is executed by each test. Our ReMT technique computes the set of dangerous edges reachable from each node n along the execution of each test t in the test suite T based on inter-procedural CFG traversal:

DEFINITION 2.7. *For an inter-procedural CFG $\langle N, E\rangle$ with a set of dangerous edges $E_\triangle$, the **dangerous-edge reachability** for node $n \in N$ with respect to test $t \in T$ is a predicate* `reach` *$\subseteq N \times T$;* `reach`(n, t) *holds iff an execution path of t could potentially go through node n and reach a dangerous edge after n.*

Note that a node n can have different reachability results with respect to different tests, i.e., `reach`(n, t) for a test t may differ from `reach`(n, t') for another test t'.

Our ReMT technique also utilizes the test coverage of CFG nodes and edges. Specifically, we utilize *partial* test coverage on CFG nodes and edges before a given CFG node is executed:

DEFINITION 2.8. *For a program with CFG $\langle N, E\rangle$, **test coverage** is a function* `trace`*: $T \times (N \cup \{\bot\}) \to 2^{N \cup E}$ that returns a set of CFG nodes $N_{sub} \subseteq N$ and a set of CFG edges $E_{sub} \subseteq E$ covered by test t before the first execution of node $n \in N$;* `trace`$(t, \bot)$ *is the set of all nodes and edges covered by test t.*

Note that this notation allows simply using `trace`$(t, \texttt{mapN}(n_m))$ to evaluate to (1) the set of nodes and edges covered before n_m if there is a corresponding mapped node for n_m, and (2) the set of all nodes and edges covered by t if there is no mapped node.

3. EXAMPLE

Figure 1 shows two versions of a small program, `Account`, which provides basic bank account functionality. Lines 20 and 25 in the old version are changed into lines 21 and 26 in the new version, respectively. As the change on line 25 would cause the regression test suite (`TestSuite`) to fail on `test3`, the developer also modifies `test3` to make the suite pass.

Figure 2 shows the inter-procedural CFG. We depict the changed nodes in gray; *dangerous edges* are the edges incident to the gray nodes (e.g., $\langle 19, 20\rangle$, $\langle 11, 25\rangle$, and $\langle \texttt{return}, 40\rangle$). The CFG consists of six intra-procedural sub-CFGs, which are connected using inter-procedural invocation and return edges. Each invocation site is represented by an invocation node and a return node, which are connected by a virtual path edge.

```
1  public class Account {
2    double balance;  double credit;
3    public Account(double b,double c){
4      this.balance=b; // deposit balance
5      this.credit=c; // consumed credit
6    }
7    public double getBalance(){
8      return balance;
9    }
10   public String withdraw(double value){
11     if(value>0){
12       if(balance>value){//deposit enough?
13         balance=balance-value;
14         return "Success code: 1";
15       }
16       double diff=value-balance;
17       if(credit+diff<=1000){//credit enough?
18         balance=0;
19         credit=credit+diff;
20       - return "Success code:  1";
21       + return "Success code:  2";
22       }
23       else return "Error code: 1";
24     }
25   - return "Error code:  1";
26   + return "Error code:  2";
27   }
28 }
29 public class TestSuite {
30   public void test1(){
31     Account a=new Account(20.0,0.0);
32     assertEquals(20.0,a.getBalance());}
33   public void test2(){
34     Account a=new Account(20.0,0.0);
35     String result=a.withdraw(10.0);
36     assertEquals("Success code: 1",result);}
37   public void test3(){
38     Account a=new Account(20.0,0.0);
39     String result=a.withdraw(-10.0);
40   - assertEquals("Error code:  1",result);
41   + assertEquals("Error code:  2",result);}
42 }
```

Figure 1: Example code evolution and test suite.

To illustrate ReMT, consider the mutants in Table 1. Assuming we already have some mutant execution results from the old version, we collect mutant execution results for the new version incrementally. We demonstrate both full mutation testing and partial mutation testing scenarios. Following Definition 2.1, the example input matrices of the old version in both scenarios are shown in the top parts of tables 2 and 3. In both scenarios, we initialize the mutation results of the new program version as a new mutant execution matrix with all `U` elements to denote that the mutant execution results are initially unknown. In total, at most 27 mutant-test executions are needed for computing each mutant execution matrix for the new version.

To reduce the number of mutant-test executions, several mutation testing tools [1,21,39] utilize the following fact: when a test executed on the original, unmutated program does not cover the mutated statement of a mutant, then that test cannot kill that mutant. One can thus filter out a set of tests for each mutant (or dually a set of mutants for each test). Figure 2 highlights the execution traces of `test1`, `test2`, and `test3` after evolution with bold solid (red) lines, bold dashed (blue) lines, and bold dotted (gray) lines, respectively. Here, for example, any mutant that does not

Table 1: Mutants for illustration.

Mutant	Mutated Location	Mutant Statement
m_1	n_4	`this.balance=`**`0`**
m_2	n_5	`this.credit=`**`0`**
m_3	n_8	`return` **`1`**
m_4	n_{11}	`if(value`**`<=`**`0)`
m_5	n_{12}	`if(balance`**`<`**`value)`
m_6	n_{13}	`balance=balance`**`+`**`value`
m_7	n_{13}	`balance=balance`**`*`**`value`
m_8	n_{13}	`balance=balance`**`/`**`value`
m_9	n_{14}	`return` **`""`**

Table 2: Incrementally collecting full matrix.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9
t1	K	N	K	N	N	N	N	N	N
t2	N	N	N	K	N	N	N	N	K
t3	N	N	N	K	N	N	N	N	N
t1	K	N	K	E	E	E	E	E	E
t2	U	U	E	U	U	N	N	N	K
t3	U	U	E	U	E	E	E	E	E

Table 3: Incrementally collecting partial matrix.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9
t1	K	N	K	N	N	N	N	N	N
t2	U	N	U	K	N	N	N	N	K
t3	U	N	U	U	N	N	N	N	U
t1	K	N	K	E	E	E	E	E	E
t2	(U)	U	E	U	U	N	N	N	K
t3	(U)	U	E	U	E	E	E	E	E

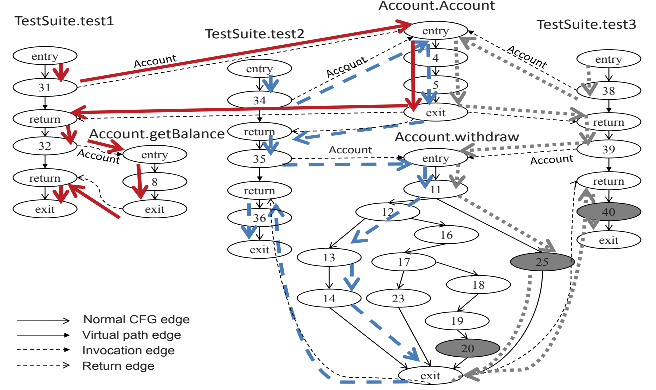


Figure 2: Inter-procedural CFG for the example.

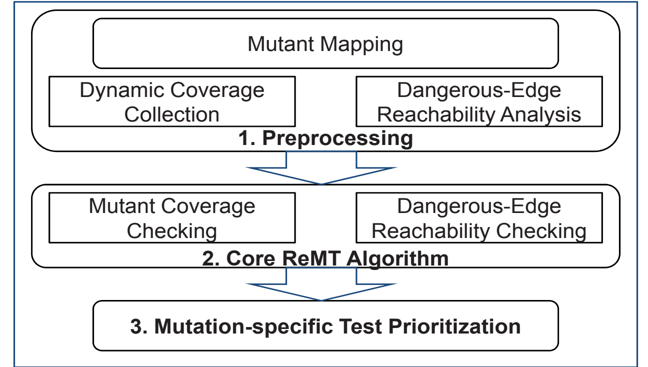


Figure 3: General approach of ReMT.

occur on the nodes in bold solid (red) lines cannot be killed by `test1`, and any mutant that does not occur on the nodes in bold dashed (blue) lines cannot be killed by `test2`. The matrices for the new version can then be updated with `E` elements to denote a mutant that cannot be killed by a test because its mutated statement is not reached by the test. The light-gray cells in tables 2 and 3 show the cells updated with `E`'s. Now, 14 mutant-test executions (i.e., cells not marked as `E`'s) are required for computing the full matrix, and at most 14 executions are required for computing the partial matrix.

To further reduce the number of mutant-test executions, our ReMT leverages program evolution information. For instance, mutants m_6, m_7, m_8, and m_9 would need to be executed against `test2` when not consider evolution; however, we can compute that those mutants cannot modify the `test2`'s execution trace to reach any dangerous edge, because there is no CFG path from the nodes where those mutants occur (i.e., n_{13} and n_{14}) to the evolved code (i.e., n_{20}, n_{25}, and n_{40}). Therefore, the mutation testing results for mutants on these two nodes cannot differ from their previous results for the program before evolution, and these results can be directly reused from the old mutant execution matrix.

We use a static analysis to determine which dangerous edges can be reached by mutants. It is important to point out that this analysis is done *with respect to each test* (Definition 2.7) because the results can differ for different tests. For example, consider node n_4 and mutant m_1. Although n_4 *cannot* reach any dangerous edge through inter-procedural CFG traversal with respect to `test1`, n_4 *can* potentially reach dangerous edges through traversal from `test2`. When executing m_1 on `test2`, the execution path takes a different branch at n_{12} than the execution path takes when executing the unmutated new version. Thus m_1 executes the dangerous edge $\langle 19, 20\rangle$, which actually causes m_1 to be killed for the new version although it is not killed for the old version. Therefore, our dangerous-edge reachability analysis considers potential execution paths for each test. The matrices for the new version can now be updated with history information from the old version (shown as dark-gray cells in tables 2 and 3). Thus, only 7 mutant-test executions (i.e., cells with `U`'s) are required for obtaining the full matrix, and at most 5 executions are required for obtaining the partial matrix (the two `U`s within brackets of m_1 do not need to be filled because m_1 is already killed by `test1`). In sum, for this example, compared with the state-of-the-art mutation testing techniques, ReMT reduces the number of mutant-test executions 2X (7 vs. 14) for full mutation testing and over 2X for partial mutation testing (depending on the order in which tests are executed for a mutant as we discuss later).

4. REGRESSION MUTATION TESTING

4.1 Overview

This section presents regression mutation testing (ReMT). Figure 3 shows the three key components. The Preprocessing component (Section 4.2) builds a mapping between the mutants of the two versions and gathers initial data for the checking performed by the core ReMT component (Section 4.3), which consists of two steps: *mutant-coverage checking* and *dangerous-edge reachability checking*. Mutant-coverage checking follows previous work [1, 21, 39] in using the coverage information of all tests on the new program version to select the subset of tests that actually execute the mutated statement and thus may kill the mutant for the new version. For the selected tests that do not have execution history (i.e., are newly added tests), ReMT executes them for gathering mutation

testing results for the mutant. For the selected tests that have execution history, ReMT's dangerous-edge reachability checking determines whether the mutation results can be reused. More precisely, a mutant-test result can be reused if (1) no dangerous edge is executed from the beginning of the test to the mutated statement and (2) no dangerous edge can be executed from the mutated statement to the end of the test. For (1), ReMT uses dynamic coverage, and for (2), ReMT uses a novel dangerous-edge reachability analysis. When possible, ReMT directly reuses execution results from their previous execution on the mapping mutant of the old version. Finally, as the order of test execution matters for killing mutants faster, ReMT's Mutation-specific Test Prioritization component (Section 4.4) reorders tests to further optimize regression mutation testing.

4.2 Preprocessing

Preprocessing consists of mutant mapping, coverage collection, and dangerous-edge reachability analysis. Coverage collection uses the common code instrumentation, so we present details of the mutant mapping and dangerous-edge reachability analysis. The construction of mutant mapping also identifies dangerous edges that are used in dangerous-edge reachability analysis.

4.2.1 Mutant Mapping

Following existing regression test selection (RTS) techniques [17, 34, 37], ReMT uses control-flow graph (CFG) to represent program versions and identifies program changes as dangerous edges. ReMT uses a standard depth-first search (DFS) for detecting dangerous edges [17,37]. In addition, ReMT's CFG comparison algorithm builds `mapN`, which stores the CFG node mapping between the two program versions (Definition 2.5) and is used to calculate mutant mapping `mapM` (Definition 2.6). When visiting a node pair, the CFG comparison algorithm first marks the node pair as visited and puts the matched node pair into `mapN`. Then, the algorithm iterates over all the outgoing edges of the node pair: (1) for the edges without matched labels or target nodes, the algorithm puts the edges into the dangerous edge set $E_\triangle$ and backtracks the traversal along those edges; (2) for the matched edges (i.e., when both labels and target nodes are matched) whose target nodes have been visited, the algorithm backtracks; (3) for the matched edges whose target nodes have not been visited, the algorithm recursively traverses the target node pairs. Finally, the algorithm returns all dangerous edges $E_\triangle$, node mapping `mapN`, and mutant mapping `mapM` between the old and new program versions.

4.2.2 Dangerous-Edge Reachability Analysis

Given a test t, a node n in the CFG, and a set of dangerous edges $E_\triangle$, dangerous-edge reachability computes if n can reach a dangerous edge with respect to t (Definition 2.7). We reduce the dangerous-edge reachability problem to the *Context-Free-Language (CFL) Reachability* [27,36] problem. The use of CFL-reachabillity on the *inter*-procedural CFG allows us to obtain more precise results than we would obtain by running a simple reachability that would mix invocation and return edges. (For example, in Figure 1, a naïve reachability could mix the invocation of the `Account` constructor from `test1` with the return from the `Account` constructor to `test3` and could then (imprecisely) find that `test1` can reach a dangerous edge that ends in n_{40}.) In CFL-reachability, a path is considered to connect two nodes only if the concatenation of the labels on the edges of the path is a word in a particular context-free language:

DEFINITION 4.1. *Let L be a context-free language over alphabet Σ and G be a graph whose edges are labeled with elements of Σ. Each path in G defines a word over Σ formed by concatenating the labels of the edges on the path. A path in G is an* L-path *if its word is a member of the language L.*

We reduce our dangerous-edge reachability analysis to a CFL-reachability problem as follows. For a CFG $\langle N, E\rangle$ with I invocation sites, the alphabet Σ has symbols $(_i$ and $)_i$ for each i from 1 to I, as well as two unique symbols, e and d. Following the existing inter-procedural program analysis [27,36], our analysis labels all the *intra*-procedural edges with e, and for each invocation site i labels its invocation edge and return edge with $(_i$ and $)_i$, respectively. In contrast with the existing techniques, our analysis further labels all dangerous edges with d. A path in $\langle N, E\rangle$ is a *matched path* iff the path's word is in the language $L(matched)$ of balanced-parenthesis strings according to the following context-free grammar:

$$\begin{aligned} matched \;&\rightarrow\; matched\ matched \\ &\quad\;\; \mid\, (_i\, matched\,)_i \qquad \forall i: 1 \le i \le I \\ &\quad\;\; \mid\, e \mid d \mid \varepsilon \end{aligned} \tag{1}$$

The language $L(dangerous)$ that accepts all possible valid execution paths to dangerous edges is defined as:

$$\begin{aligned} dangerous \;&\rightarrow\; matched\ dangerous \\ &\quad\;\; \mid\, (_i\, dangerous \qquad \forall i: 1 \le i \le I \\ &\quad\;\; \mid\, d \end{aligned} \tag{2}$$

The language $L(dangerous)$ is a language of partially balanced parentheses, which allows representing that the execution might go into some deeper stacks and not return as long as it encounters a dangerous edge. A path is a *dangerous path* iff the path's word is in the language $L(dangerous)$.

The problem of determining all possible nodes that can reach dangerous edges with respect to each test is transformed into the problem of finding all the possible nodes reachable from the root node of each test in the language $L(dangerous)$. For a node n and a test t, $\texttt{reach}(n, t)$ holds if n is reachable from the root node of t in the language $L(dangerous)$; otherwise $\texttt{reach}(n, t)$ does not hold (Definition 2.7). Our implementation uses the general dynamic-programming algorithm to efficiently solve the CFL-reachability problem [27] and record all the nodes that can appear on the dangerous paths for each test. Note that the analysis for one test gives the dangerous-edge reachability for *all* mutants that the test can execute, i.e., ReMT does not repeat this static analysis for each mutant-test pair. Also note that we apply dangerous-edge reachability analysis on the *old* (not new) program version.

4.3 ReMT Algorithm

Algorithm 1 shows our core ReMT algorithm, which supports both the partial mutation testing and full mutation testing scenarios. The underlined statements are specific to the partial mutation testing scenario. Note that ReMT does not require a full input matrix on old version. The algorithm expects that preprocessing (Section 4.2) has been performed, which enables the use of mutant mapping `mapM` in line 12, mutant-coverage checking (denoted as `MCoverageCheck`) in line 5, and dangerous-edge reachability checking (denoted as `DReachabilityCheck`) in line 10.

4.3.1 Basic Algorithm

Lines 2-19 iterate over the mutants of P' and the tests in T' to get the mutation testing results. For each mutant m, lines 3 and 4 first initialize all the test results as `U`. Line 5 applies mutant-coverage checking [1,21,39] between P' and m to select the subset of tests

Algorithm 1: Algorithm for ReMT

```
Input: P and P′, old and current program versions; M and M′, the
       mutants for P and P′; T and T′, test suites for P and P′;
       matrix, the mutant execution results for P.
Output: matrix′, the mutant execution results for P′.
Require: Preprocessing (Mutant Mapping, Coverage Collection, and
         Dangerous-Edge Reachability Results).
1  begin ReMT
2    foreach mutant m : M′ do
3      foreach test t : T′ do
4        matrix′(m, t) ← U // initialization
5      T_c ← MCoverageCheck(T′, P′, m)
6      killed ← false
7      foreach test t : T′ − T_c do
8        matrix′(m, t) ← E // t cannot kill the mutant
9      T′_c ← T_c ∩ T // tests with execution history
10     T_r ← DReachabilityCheck(E_Δ, T′_c, m, P, P′)
11     foreach test t : T′_c − T_r do
12       matrix′(m, t) ← matrix(mapM(m), t)
13       if matrix′(m, t) = K then
14         killed ← true // m has been killed
15     if killed = true then continue
16     foreach test t : T_c do
17       if matrix′(m, t) = U then
18         matrix′(m, t) ← Execution(t, m)
19         if matrix′(m, t) = K then continue
20   return matrix′ // return mutation testing result
```

within test suite T' that cover the mutated node n_m of m on P'. Formally, the mutant-coverage checking is computed as:

$$\texttt{MCoverageCheck}(T', P', m) = \{t \in T' | n_m \in \texttt{trace}'(t, \bot)\}$$

where $\texttt{trace}'(t, \bot)$ is the entire coverage of t on P' (Definition 2.8). The tests that do not cover n_m in P' cannot kill m, so lines 7-8 assign E to all such tests. Line 9 stores in T'_c the tests in T_c that have execution history (i.e., the tests that also exist in the old suite of P). Line 10 finds the tests from T'_c that can potentially reach dangerous edges in $E_\triangle$ when executing the mutated statement n_m (Section 4.3.2). For the tests in $T'_c - T_r$ that cannot reach any dangerous edge, ReMT directly copies the execution results from the corresponding mapping mutant of P to the execution results on m of P' (lines 11-14). Note that when the input matrix is partial, ReMT may also copy U values to the new matrix. When ReMT is applied in the partial mutation testing scenario, it sets the flag $killed$ to `true` if the mapping mutant has been killed for P and proceeds to the next mutant (line 15). Lines 16-19 run all the tests in T_c with value U on m (i.e., the newly added tests without execution history in T_c, the potentially influenced tests that could reach a dangerous edge, and the tests whose results are copied as Us from the input matrix). When ReMT is applied in the partial mutation testing scenario, it terminates the test execution for m as soon as m is killed by some test. Finally, line 20 returns the mutation testing results for P'.

4.3.2 Dangerous-Edge Reachability Checking

Algorithm 1 invokes `DReachabilityCheck` at line 10 to perform dangerous-edge reachability checking. After computing T'_c, the set of tests that execute the mutated statement for m and have execution history, ReMT further computes T_r, the tests from T'_c that can potentially reach dangerous edges $E_\triangle$ between P and P'. There are two types of tests from T'_c that can potentially reach $E_\triangle$: (1) the tests that directly execute edges in $E_\triangle$ before the first execution of the mutated CFG node n_m; and (2) the tests that can potentially reach edges in $E_\triangle$ from the mutated CFG node. The first type of tests is easily identified by intersecting $E_\triangle$ with edge coverage before the mutated node, while the second type of tests is identified by checking the reachability to dangerous edges from the mutated node with respect to the corresponding test. Formally, we define the reachability checking as follows:

$$\texttt{DReachabilityCheck}(E_\triangle, T'_c, m, P, P') =$$
$$\{t \in T'_c | \texttt{trace}(t, \texttt{mapN}(n_m)) \cap E_\triangle \neq \emptyset \ \vee \texttt{reach}(\texttt{mapN}(n_m), t)\}$$

where `trace` denotes the test coverage for P, and `reach` denotes the reachability for dangerous edges. Note that the checking is performed on the old version P because $E_\triangle$ are edges from P. Thus, we need to map n_m back to its mapped node in P. (If there is no mapped node, there must be a dangerous edge before n_m and thus $\texttt{trace}(t, \texttt{mapN}(n_m)) = \texttt{trace}(t, \bot)$ is overlapped with $E_\triangle$.)

4.4 Mutation-Specific Test Prioritization

We next present *mutation-specific test prioritization* (MTP) that aims to prioritize remaining tests for each mutant to kill it as early as possible in the partial mutation testing scenario. Given a mutant m of a program P' (that evolved from P), MTP calculates the priority of each test based on its coverage of the mutated statement as well as the mutation testing history. Formally, the priority of test t for m is calculated as:

$$\texttt{Pr}(t, m) = \begin{cases} \langle 1, \texttt{CovNum}(t, n_m)\rangle, & \text{if } \texttt{matrix}(\texttt{mapM}(m), t) = \texttt{K} \\ \langle 0, \texttt{CovNum}(t, n_m)\rangle, & \text{otherwise.} \end{cases}$$

The priority is a pair whose first element represents the mutation testing result for the test on the corresponding mutant of the old version P (`1` if killed, `0` otherwise), and the second element, $\texttt{CovNum}(t, n_m)$, is the number of times the test covers the statement (in the unmutated new version P') to be mutated (to form the mutant). Note that if the test does not have an execution history on the old version (e.g., the test is newly added or was not executed in the partial scenario), or the mutant does not have a mapping mutant for the old version, the first element is set to `0`.

For each mutant, ReMT prioritizes tests lexicographically based on their priority pairs. The tests with first elements set to 1 are executed earlier based on the intuition that a test that kills a mutant in the old version might also kill its mapping mutant in the new version. The tests with second elements that indicate more execution are executed earlier based on the intuition that a mutant is more likely to be killed if its mutated statement was covered more times by a test. If two tests have the same priority values, ReMT executes them according to their order in the original test suite.

4.5 Discussion and Correctness

While we presented ReMT for Java and JUnit tests, it is also applicable for other languages and test paradigms. When the test code does not have unit tests, our dangerous-edge reachability analysis can be directly applied on the `main` method of the system under test. Note that ReMT only works for traditional mutation operators that change statements in methods. In the future, we plan to support class-level mutation operators [24] that can change class hierarchy.

We need to show that for each mutant-test result reused from the old version, the same result would be obtained if the corresponding mutant and test were run on the new version. Intuitively, ReMT is correct as it works similarly to regression test selection: for each mutant, any test that might potentially reach dangerous edges is

selected. Due to space limitation, we do not show a detailed correctness proof here, but it can be found online[1].

5. IMPLEMENTATION

We built ReMT on top of Javalanche [39], a state-of-the-art tool for mutation testing of Java programs with JUnit tests. Javalanche allows efficient mutant generation as well as efficient mutant execution. It uses a small set of sufficient mutation operators [31], namely replace numerical constant, negate jump condition, replace arithmetic operator, and omit method calls [39]. Javalanche manipulates Java bytecode directly using mutant schemata [40] to enable efficient mutant generation. For efficient mutant execution, Javalanche does not execute the tests that do not reach the mutated statement, and it executes mutants in parallel. It provides the `javalanche.stop.after.first.fail` configuration property to select partial or full mutation scenario.

Our ReMT implementation extends Javalanche with dangerous-edge reachability checking and mutation-specific test prioritization. For static analysis, our implementation uses the intra-procedural CFG analysis of the *Sofya tool* [22] to obtain basic intra-procedural CFG information and uses the *Eclipse JDT toolkit*[2] to obtain the inter-procedural information (method-overriding hierarchy, type-inheritance information, etc.) for inter-procedural CFG analysis. As a way to test our implementation, our experimental study verified that the incrementally collected mutation testing results by ReMT are the same as (non-incrementally collected) mutation testing results by Javalanche.

6. EXPERIMENTAL STUDY

ReMT aims to reduce the cost of mutation testing by utilizing the mutation testing results from a previous program version. To evaluate ReMT, we compare it with Javalanche [39], the state-of-the-art tool for mutation testing.

6.1 Research Questions

Our study addresses the following research questions:

- **RQ1:** How does ReMT compare with Javalanche, which does not use history information, in the full mutation testing scenario in terms of both efficiency and effectiveness?
- **RQ2:** How does ReMT compare with Javalanche in the partial mutation testing scenario under different original test-suite orders?
- **RQ3:** How does the mutation-specific test prioritization (MTP) further optimize ReMT in the partial mutation testing scenario?

6.2 Independent Variables

We used the following three independent variables (IVs):
IV1: Different Mutation Testing Techniques. We considered the following choices of mutation testing techniques: (1) Javalanche, (2) ReMT, and (3) ReMT+MTP.
IV2: Different Mutation Testing Scenarios. We considered two mutation testing scenarios for applying mutation testing: (1) full mutation testing and (2) partial mutation testing.
IV3: Different Test-Suite Orders. As the performance of all evaluated techniques under the partial mutation testing scenario depends on the test-suite orders, we used 20 randomized original test-suite orders for each studied revision to evaluate the performance of each technique under that scenario.

[1]https://webspace.utexas.edu/lz3548/www/publications/tr2012a.pdf
[2]http://www.eclipse.org/jdt/

Table 4: Subjects overview.

Projects	Description	Source+Test(LoC)
JDepend	Design quality metrics	2.7K+1.2K
Time&Money	Time and money library	2.7K+3.1K
Barbecue	Bar-code creator	5.4K+3.3K
Jaxen	Java XPath library	14.0K+8.8K
Commons-Lang	Java helper utilities	23.3K+32.5K
Joda-Time	Time library	32.9K+55.9K

6.3 Dependent Variables

Since we are concerned with the effectiveness as well as efficiency achieved by our ReMT technique, we used the following two dependent variables (DVs):
DV1: Number of Mutant-Test Executions. This variable denotes the total number of mutant-test pairs executed by the compared techniques.
DV2: Time Taken. This variable records the total time (including test execution time and technique overhead) taken by the compared techniques.

6.4 Subjects and Experimental Setup

We used the source code repositories of six open-source projects in various application domains. Table 4 summarizes the projects. The sizes of the studied projects range from 3.9K lines of code (LoC) (*JDepend*, with 2.7KLoC source code and 1.2KLoC test code) to 88.8KLoC (*Joda-Time*, with 32.9KLoC source code and 55.9KLoC test code). We applied our ReMT on five recent revisions of each project. We treated each commit involving source code or test code changes as a revision; for commits conducted within the same day, we merged them into one revision. Table 5 shows more details for each revision in the context of mutation testing: Column 1 names the studied revision; Column 2 shows the number of source/test files committed; Columns 3 and 4 show the number of tests and mutants; Column 5 shows the ratio of killed mutants to all mutants and the ratio of killed mutants to reached mutants.

In this experimental study, for the full mutation testing scenario, both the input and output mutation matrices are full (no `U`), and for the partial mutation testing scenario, both the input and the output mutation matrices are partial (but with enough information to compute the mutation score). The experimental study was performed on a Dell desktop with Intel i7 8-Core 2.8GHz processor, 8G RAM, and Windows 7 Enterprise 64-bit version.

6.5 Results and Analysis

6.5.1 RQ1: Full Mutation Testing Scenario

In Table 5, Column 6 shows the total possible number of mutant-test executions without any reduction techniques, i.e., the product of the numbers of tests and mutants. Columns 7-9 show the actual number of executions performed by Javalanche and ReMT, and the reduction in the number of executions by ReMT over Javalanche. First, we observe that both Javalanche and ReMT significantly reduce the number of executions from the total possible executions. For instance, for all five revisions of *Barbecue*, the total possible number of executions are more than 5 million, while both Javalanche and ReMT are able to reduce the number of executions to around or below 0.02 million. Second, although the reductions of ReMT over Javalanche vary greatly across subject revisions, ReMT is able to further achieve reductions of more than 50% on the majority of all the revisions. Furthermore, ReMT is able to achieve reductions of more than 90% on 12 of the 30 studied revisions. For instance, on revision *Time&Money*-4, ReMT is even able to iden-

Table 5: Experimental results of Javalanche and ReMT under the full mutation testing scenario.

Revision	Chg Files	Tests	Mutants	Mutant Kill Rates (%)	Total Executions	Number of Executions			Time Taken		
						Javalanche	ReMT	Reduction	Javalanche	ReMT (Overhead)	Reduction
JDepend-1	2	53	1,067	65.97/84.00	56,551	10,769	1,196	88.89%	00:05:45	00:02:04 (00:05)	64.05%
JDepend-2	8	53	1,166	67.40/84.24	61,798	12,000	10,516	12.37%	00:06:14	00:02:21 (00:05)	62.29%
JDepend-3	3	54	1,174	67.46/83.98	63,396	12,528	7,492	40.19%	00:06:00	00:02:11 (00:06)	63.61%
JDepend-4	2	55	1,174	67.97/84.62	64,570	12,956	7,920	38.87%	00:06:04	00:02:07 (00:06)	65.10%
JDepend-5	2	55	1,174	67.97/84.62	64,570	12,956	6,826	47.31%	00:06:06	00:02:40 (00:05)	56.28%
Time&Money-1	1	235	2,293	72.21/87.02	538,855	15,320	58	99.62%	00:09:08	00:02:11 (00:08)	76.09%
Time&Money-2	2	236	2,305	72.27/87.08	543,980	15,663	3,637	76.77%	00:09:19	00:02:23 (00:07)	74.41%
Time&Money-3	4	236	2,305	72.27/87.08	543,980	15,663	0	100.00%	00:09:18	00:02:12 (00:07)	76.34%
Time&Money-4	2	236	2,305	72.27/87.08	543,980	15,663	0	100.00%	00:09:18	00:02:12 (00:07)	76.34%
Time&Money-5	7	237	2,300	73.82/86.67	545,100	16,805	12,478	25.75%	00:09:22	00:07:30 (00:08)	19.92%
Barbecue-1	3	154	36,419	2.75/68.39	5,608,526	21,267	20,852	1.95%	00:09:31	00:07:23 (00:15)	22.41%
Barbecue-2	1	154	36,419	2.75/68.39	5,608,526	21,267	11,352	46.62%	00:09:32	00:05:57 (00:21)	37.58%
Barbecue-3	1	154	36,419	2.75/68.39	5,608,526	21,267	8,850	58.39%	00:09:36	00:06:10 (00:12)	35.76%
Barbecue-4	3	154	36,419	2.75/68.39	5,608,526	21,267	36	99.83%	00:09:47	00:04:21 (00:12)	55.53%
Barbecue-5	1	154	36,419	2.75/68.39	5,608,526	21,267	4,715	77.82%	00:09:11	00:05:27 (00:12)	40.65%
Jaxen-1	3	688	9,937	46.49/70.00	6,836,656	1,495,822	25,641	98.28%	01:06:35	00:16:05 (00:33)	75.84%
Jaxen-2	5	689	9,876	46.69/70.53	6,804,564	1,489,630	1,488,367	0.08%	01:06:10	01:06:52 (00:42)	-1.05%
Jaxen-3	3	690	9,881	46.71/70.53	6,817,890	1,493,341	998,045	33.16%	01:06:21	00:42:25 (00:23)	36.07%
Jaxen-4	3	694	9,891	46.79/70.59	6,864,354	1,504,587	98,411	93.45%	01:06:50	00:21:07 (00:25)	68.40%
Jaxen-5	2	695	9,901	46.84/70.63	6,881,195	1,508,683	430,521	71.46%	01:07:35	00:27:40 (00:25)	59.06%
Commons-Lang-1	5	1,689	19,747	65.63/86.21	33,352,683	93,423	46	99.95%	01:31:57	00:32:11 (01:42)	64.99%
Commons-Lang-2	8	1,691	19,747	65.64/86.21	33,392,177	93,425	19	99.97%	01:32:07	00:32:08 (01:32)	65.11%
Commons-Lang-3	2	1,691	19,747	65.69/86.25	33,392,177	93,430	1,124	98.79%	01:32:10	00:31:55 (01:32)	65.37%
Commons-Lang-4	2	1,691	19,747	65.68/86.23	33,392,177	93,430	352	99.62%	01:32:33	00:32:09 (01:31)	65.26%
Commons-Lang-5	3	1,692	19,747	65.68/86.24	33,411,924	93,450	10,915	88.31%	01:32:15	00:40:31 (01:31)	56.07%
Joda-Time-1	2	3,818	24,175	65.09/85.41	92,300,150	1,064,395	776,299	27.06%	04:21:58	03:12:21 (03:33)	26.57%
Joda-Time-2	2	3,828	24,190	66.47/87.19	92,599,320	1,076,987	865,922	19.59%	04:30:47	03:27:05 (03:23)	23.52%
Joda-Time-3	3	3,829	24,219	66.43/87.09	92,734,551	1,077,851	619	99.94%	03:58:55	00:54:04 (01:56)	77.37%
Joda-Time-4	7	3,832	24,236	66.71/87.44	92,872,352	1,077,757	795,152	26.22%	03:59:48	03:14:03 (03:29)	19.07%
Joda-Time-5	1	3,834	24,236	66.41/87.05	92,920,824	1,078,573	1,443	99.86%	04:15:57	00:54:07 (01:44)	78.85%

tify that no executions are required to get the new mutation testing results. Manually inspecting the code changes in this revision, we found that the developers changed parts of two source files that cannot be reached by any tests, and thus the mutation testing results cannot be influenced. However, there are also revisions for which ReMT cannot achieve much reduction. For instance, on revision *Jaxen*-2, ReMT is able to achieve a reduction of only 0.08% over Javalanche. We looked into the revision history and found that the developers conducted an import patch across all methods of that `org.jaxen.saxpath.base.XPathLexer` class that is a central class used by nearly all the tests in the suite.

Columns 10-12 compare the actual tool time rather than the number of executions. Column 10 of Table 5 shows the mutation testing time taken by Javalanche. Column 11 shows the overall mutation testing time taken by ReMT, including the time taken by the preprocessing steps of ReMT (specifically by mutant mapping and dangerous-edge reachability analysis)[3]. Column 12 shows the reduction of costs by ReMT over Javalanche in terms of time. First, we observe that the reduction in terms of time does not directly match the reduction in terms of the number of executions; sometimes the reduction for time is lower (e.g., *JDepend*-1), and sometimes it is higher (e.g., *JDepend*-2). The likely reasons for this include the following: (1) the times for different executions vary significantly, (2) the reachability checking (lines 9-14 in Algorithm 1) needs extra time, and (3) Javalanche's parallel thread scheduling, database setup, and database access can influence the execution time. Second, we observe that our preprocessing step scales quite well: it takes at most 3 minutes and 33 seconds across all revisions (*Joda-Time*-1) and is negligible compared to the mutant-test execution time.

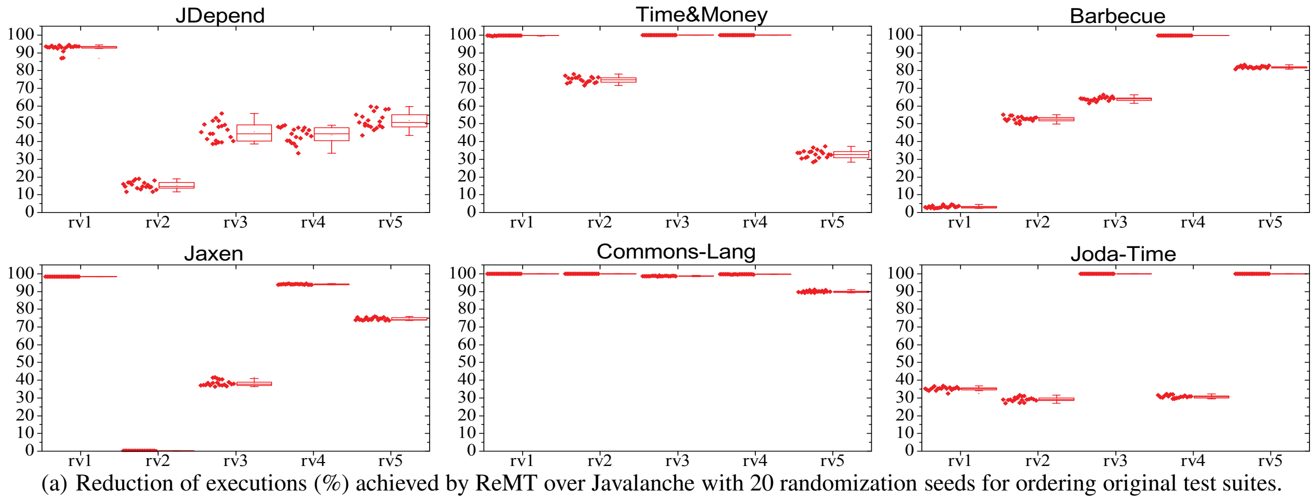
[3]We do not explicitly measure the coverage preprocessing time because node coverage is already traced by Javalanche, and edge coverage is available for any system using regression test selection.

6.5.2 *RQ2: Partial Mutation Testing Scenario*

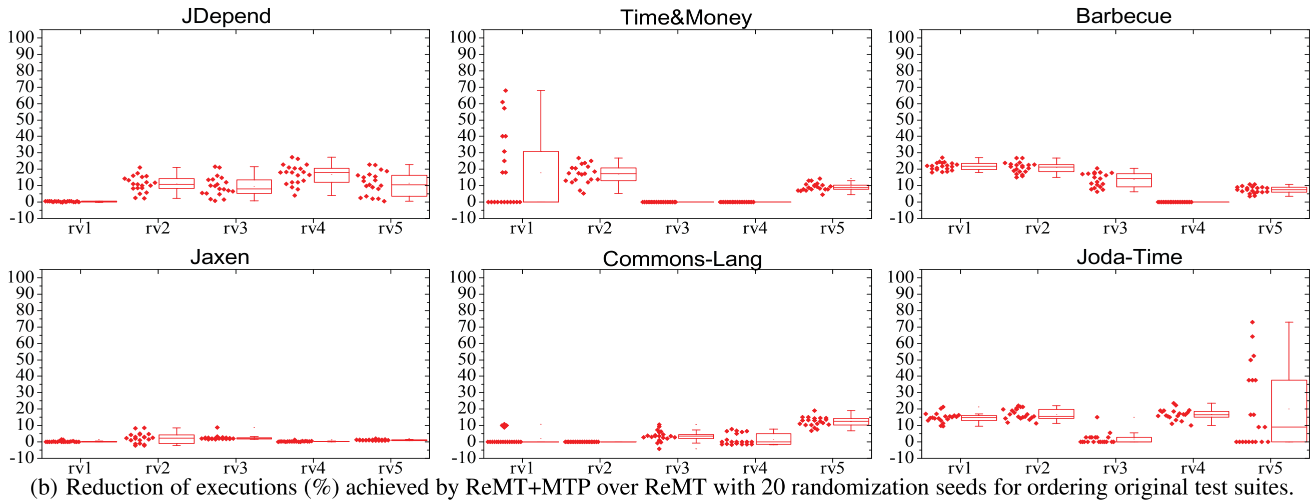
As different test-suite orders influence the performance of techniques under the partial mutation testing scenario, we evaluated the performance of ReMT and Javalanche under 20 different original test-suite orders. Figure 4(a) shows the reduction that ReMT achieves over Javalanche in terms of executions. In each plot, the horizontal axis shows different revisions of each subject, and the vertical axis shows the ratios of executions reduced by ReMT over Javalanche. Each box plot shows the mean (a dot in the box), median (a line in the box), upper/lower quartile, and max/min values for the reduction ratios achieved over 20 randomized original test-suite orders on each revision of each studied subject. The corresponding data dots are also shown to the left of each box plot. First, we observe that the reduction achieved by ReMT in the partial mutation testing scenario follows a similar trend as the reduction achieved in the full mutation testing scenario. In addition, the reductions achieved by ReMT over Javalanche under the partial mutation testing scenario are even slightly greater than under the full mutation testing scenario for 23 of the 30 revisions. Second, the reduction achieved by ReMT over Javalanche for each revision is not greatly influenced by different test suite orders: the standard deviation values for the reduction only range from 0 to 5.33. While the reduction ratios are not greatly influenced by test-suite orders, an interesting finding is that the reduction ratios tend to be more stable when the reduction grows higher. For example, for all revisions with reduction ratios of more than 90%, different test-suite orders tend to have almost no impact at all on the reduction ratios.

6.5.3 *RQ3: Mutation-Specific Test Prioritization*

Figure 4(b) shows the further reduction of executions achieved by mutation-specific test prioritization (MTP) over ReMT using 20 randomized original test-suite orders for each revision. Each box plot has the same format as in Figure 4(a) except that the verti-



(a) Reduction of executions (%) achieved by ReMT over Javalanche with 20 randomization seeds for ordering original test suites.



(b) Reduction of executions (%) achieved by ReMT+MTP over ReMT with 20 randomization seeds for ordering original test suites.

Figure 4: Reduction of executions achieved by ReMT and MTP under the partial mutation testing scenario.

cal axis represents the ratios of executions reduced by *ReMT+MTP over ReMT (and not overJavalanche)*. First, we observe that technique ReMT+MTP further achieves a reduction over ReMT on 26 of the 30 revisions. There are also four revisions where MTP does not make any further reductions over ReMT: *Time&Money*-3, *Time&Money*-4, *Barbecue*-4, and *Commons-Lang*-2. The reason is that ReMT has already reduced the number of executions greatly, e.g., 0 executions for *Time&Money*-3 and *Time&Money*-4. Second, we observe that the reduction achieved by ReMT+MTP over ReMT for each revision can be greatly influenced by different original test-suite orders: the standard deviation values of the reduction range from 0 to 24.60, which contrasts with the reduction of ReMT over Javalanche. For example, the reductions achieved on *Joda-Time*-5 range from 0.00% to 72.97%. The reason is that MTP is just a reordering of all the tests identified by ReMT and can even execute *more* tests than the original test order executed by ReMT. Although the prioritization is done for each mutant, the experimental study shows MTP is quite lightweight: on average, its total prioritization time for all mutants is less than 1sec. for the revisions of four projects (i.e., *JDepend*, *Time&Money*, *Barbecue*, and *Commons-Lang*), and is 2.43sec. and 3.58sec. for the revisions of *Joda-Time* and *Jaxen*, respectively.

7. RELATED WORK

In this section, we present related work on mutation testing. We also discuss potential applications of ReMT to other related research problems in mutation testing (e.g., detection of equivalent mutants and test generation for killing mutants). Furthermore, as ReMT is based on incremental analysis, we also discuss existing related applications of incremental program analysis.

7.1 Tackling Cost of Mutation Testing

In general, research efforts to tackle the cost of mutation testing can be divided into three categories: selective mutation testing, weakened mutation testing, and accelerated mutation testing.

Selective mutation testing aims to select a representative subset of all mutants that can achieve similar results as the entire set of mutants. Since Mathur [25] first proposed the idea of selective mutation testing, there have been a number of research efforts in this area. Typically, selective mutation testing can be classified into operator-based mutant selection and random mutant selection. Wong and Mathur [41] investigated two mutation operators among the 22 mutation operators in Mothra [10], and found that mutants generated with the two mutation operators can achieve similar results as all the 22 mutation operators. Offutt et al. [31] experi-

mentally expanded the two mutation operators to five mutation operators (which are known as the five key mutation operators) to ensure that the representative set of mutants achieves almost the same results as the whole mutant set. Barbosa et al. [5] proposed six guidelines to determine 10 key mutation operators. Namin et al. [28] used variable reduction to determine 28 key mutation operators. Acree et al. [2] first proposed the idea of random mutant selection. Wong and Mathur [41] empirically studied random mutant selection using Mothra. While random mutant selection attracted less research attention than operator-based mutant selection, a recent study [45] demonstrated that operator-based mutant selection is actually not more effective than random selection.

Weakened mutation testing relaxes the definition of mutant killing. In *strong mutation*, a test kills a mutant if and only if the *output* of executing the mutant with the test differs from the output of executing the original program with the test. Howden [19] proposed *weak mutation*, which checks whether the test produces a different program *state* when executing the mutant than when execution the original program. Woodward and Halewood [42] proposed *firm mutation*, which is a spectrum of techniques between weak and strong mutation.

Accelerated mutation testing aims to use efficient ways to generate, compile, and execute mutants. DeMillo et al. [8] proposed compiler-integrated mutation, which extends a compiler to compile all mutants at once, so as to reduce the cost of generating and compiling a large number of mutants. Similarly, Untch et al. [40] proposed schema-based mutation, which transforms all mutants into one metamutant that can be compiled by a standard compiler. Researchers have also investigated the use of parallel processing (e.g., vector processors [26], SIMD [23], and MIMD [33]) to speed up mutation testing. Offutt et al. considered reordering tests to kill mutants faster [32]. However, their technique uses the same test order, such as executing all tests forward or reverse, for *all mutants*. In contrast, our MTP uses different orders for different mutants.

Our ReMT technique opens a new direction for tackling the cost of mutation testing and differs from the previous techniques in several ways. One, to the best of our knowledge, ReMT is the first attempt to direct mutation testing on the differences between two program versions. Two, ReMT obtains the same mutation results as standard mutation but obtains them faster, whereas selective and weakened mutation testing typically produce different results than standard mutation. Three, ReMT is orthogonal to existing techniques that optimize mutation testing and can be directly combined with those techniques to further reduce the cost of mutation testing.

7.2 Detecting Equivalent Mutants

Equivalent mutants are mutants that are semantically identical to the original program. As equivalent mutants would impact the calculation of a test suite's quality, it is preferable to identify equivalent mutants for mutation testing. However, the problem of detecting equivalent mutants is undecidable in general. Therefore, researchers investigate approximation techniques for this problem. Offutt and Craft [29] proposed a technique to detect equivalent mutants via compiler optimization. Hierons et al. [18] used slicing to reduce the numbers of possible equivalent mutants. Schuler et al. [38] proposed to use execution information to detect equivalent mutants. While not directly targeting detection of equivalent mutants, our ReMT technique may also be utilized for this purpose: ReMT determines that some tests have the same execution on a mutant for the old and new versions and thus may reduce the cost of collecting execution information needed by equivalent mutant detection [38].

7.3 Generating Tests to Kill Mutants

A number of research projects consider the problem of generating high-quality tests based on mutation testing. DeMillo and Offutt [11] proposed constraint-based testing (CBT), which uses control-flow analysis and symbolic evaluation to generate tests each killing one mutant. Offutt et al. [30] further proposed dynamic domain reduction to address some limitations of CBT. In addition to developing dedicated test-generation techniques to kill mutants, researchers have also used existing test generation engines to kill mutants. Fraser and Zeller [14] used search-based software testing (SBST) to generate tests that kill mutants. Zhang et al. [46] used dynamic symbolic execution (DSE) to generate tests that kill mutants. Harman et al. [16] combined SBST and DSE to generate tests that kill multiple mutants. Our ReMT may be also utilized to reduce the cost of generating such high-quality test suites: it is possible to incrementally generate tests for killing mutants in a way similar to augmenting existing test suites [43].

7.4 Incremental Program Analysis

Regression test selection (RTS) [17, 34, 37] identifies the differences between two program versions as dangerous edges, and only incrementally re-executes the subset of those tests whose behavior might have been influenced by the dangerous edges. Our ReMT technique also uses dangerous edges to represent program changes. However, because each mutant makes a minor change to a program CFG node, ReMT computes dangerous-edge reachability to capture *all* potentially influenced tests for each mutant. In fact, our novel dangerous-edge reachability analysis can also be applied to RTS. The main benefit of this analysis for test selection is that it enables selection in the absence of previous coverage information since it is computed statically. The main drawback is that it considers all possibilities and might select redundant tests.

There are also other applications of incremental analysis. RECOVER [6] uses dangerous edges and node mapping between two program versions to incrementally collect program coverage. Regression model checking [44] uses dangerous edges between two program versions to drive the pruning of the state space when model checking the new program version. Directed incremental symbolic execution [35] leverages program differences in the form of changed CFG nodes to guide symbolic execution to explore and characterize the effects of program changes. Our ReMT differs from existing techniques in three ways. One, ReMT is incremental program analysis for a totally different area, mutation testing. Two, the previous techniques only deal with the evolution changes between program versions, while ReMT deals with two dimensions of changes: (1) the mechanical changes introduced by mutations and (2) the evolution changes. Three, ReMT uses a novel static dangerous-edge reachability analysis based on CFL reachability, whereas the previous techniques mainly use dynamic coverage information to directly determine the reachability to dangerous edges.

8. CONCLUSIONS

We introduced Regression Mutation Testing (ReMT), a technique that leverages program differences to reduce the costs of mutation testing. ReMT is based on a static analysis that checks dangerous edge reachability for each program node with respect to different tests. We also presented a novel *mutation-specific test prioritization* technique to further speed up mutation testing. We implemented ReMT in Javalanche, a state-of-the-art mutation testing system for Java programs, and evaluated its cost and effectiveness using real-world revision repositories of six applications ranging from 3.9KLoC to 88.8KLoC. The experimental results show that ReMT can substantially reduce mutation testing costs.

In the future, we plan to explore whether different combinations of static and dynamic analyses, as well as reusing various partial results from previous runs, could further speed up mutation testing for evolving code. We also plan to explore techniques that could reduce the time to find the first mutant that is not killed (rather than reduce the time to find all the mutants that are killed).

9. ACKNOWLEDGEMENTS

The authors would like to thank Milos Gligoric and Vilas Jagannath for the initial discussions on regression mutation testing, David Schuler for providing information about Javalanche, and Jeff Offutt for providing information about optimizations for mutation testing. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-0746856 and CCF-0845628, AFOSR grant FA9550-09-1-0351, National 973 Program of China No. 2009CB320703, and the Science Fund for Creative Research Groups of China No. 61121063.

10. REFERENCES

[1] PIT. http://pitest.org/.
[2] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, 1979.
[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press.
[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.
[5] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11(2):113–136, 2001.
[6] P. K. Chittimalli and M. J. Harrold. Recomputing coverage information to assist regression testing. *IEEE TSE*, 35(4):452–469, 2009.
[7] M. E. Delamaro and J. C. Maldonado. Proteum–A tool for the assessment of test adequacy for C programs. In *Proc. the Conference on Performability in Computing Sys PCS 96*, pages 79–95, 1996.
[8] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proc. COMPSAC*, pages 351–356, 1991.
[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
[10] R. A. DeMillo and R. J. Martin. The Mothra software testing environment user's manual. Technical report, Software Engineering Research Center, 1987.
[11] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE TSE*, 17(9):900–910, 1991.
[12] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE TSE*, pages 733–752, 2006.
[13] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *JSS*, 38(3):235–253, 1997.
[14] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA*, pages 147–158, 2010.
[15] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE TSE*, pages 279–290, 1977.
[16] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proc. FSE*, pages 212–222, 2011.
[17] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proc. OOPSLA*, pages 312–326, 2001.
[18] R. Hierons and M. Harman. Using program slicing to assist in the detection of equivalent mutants. *STVR*, 9(4):233–262, 1999.
[19] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE TSE*, pages 371–379, 1982.
[20] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE TSE*, 37(5):649–678, 2011.
[21] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.
[22] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analyses for java software. Technical report, CSE, UNL, 2006.
[23] E. W. Krauser, A. P. Mathur, and V. Rego. High performance software testing on SIMD machines. *IEEE TSE*, 17(5):403–423, 1991.
[24] Y. S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system. *STVR*, 15(2):97–133, 2005.
[25] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proc. COMPSAC*, pages 604–605, 1991.
[26] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical report, Purdue University, 1988.
[27] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.
[28] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. ICSE*, pages 351–360, 2008.
[29] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *STVR*, 4(3):131–154, 1994.
[30] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, 1999.
[31] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM TOSEM*, 5(2):99–118, 1996.
[32] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. International Conference on Testing Computer Software*, 1995.
[33] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using MIMD computer. In *Proc. International Conference on Parallel Processing*, pages 257–266, 1992.
[34] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. FSE*, pages 241–252, 2004.
[35] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proc. PLDI*, pages 504–515, 2011.
[36] T. Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.
[37] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.
[38] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. ISSTA*, pages 69–80, 2009.
[39] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. FSE*, pages 297–298, 2009.
[40] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proc. ISSTA*, pages 139–148, 1993.
[41] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *JSS*, 31(3):185–196, 1995.
[42] M. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, 1988.
[43] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proc. FSE*, pages 257–266, 2010.
[44] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *Proc. ICSM*, pages 115–124, 2009.
[45] L. Zhang, S. S. Hou, J. J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.
[46] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, pages 1–10, 2010.