Isomorphic Regression Testing: Executing Uncovered Branches without Test Augmentation

Jie Zhang[1], Yiling Lou[1], Lingming Zhang[2], Dan Hao[1]*, Lu Zhang[1], Hong Mei[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
{zhangjie_marina, louyiling,haodan,zhanglucs,meih}@pku.edu.cn
[2]Department of Computer Science, University of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

ABSTRACT

In software testing, it is very hard to achieve high coverage with the program under test, leaving many behaviors unexplored. To alleviate this problem, various automated test generation and augmentation approaches have been proposed, among which symbolic execution and search-based techniques are the most competitive, while each has key challenges to be solved. Different from prior work, we present a new methodology for regression testing – **Isomorphic Regression Testing**, which explores the behaviors of the program under test by creating its variants (i.e., modified programs) instead of generating tests. In this paper, we make the first implementation of isomorphic regression testing through an approach named *ISON*, which creates program variants by negating branch conditions. The results show that *ISON* is able to additionally execute 5.3% to 80.0% branches that are originally uncovered. Furthermore, *ISON* also detects a number of faults not detected by a popular automated test generation tool (i.e., *EvoSuite*) under the scenario of regression testing.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

software testing; regression testing; branch negation;

1. INTRODUCTION

Software testing aims to assure the quality of the software under test through detecting its abnormal behaviors [50, 48]. However, it is practically impossible to explore all behaviors [48, 59, 7, 6], especially for large complex software. To measure to what extent the software behaviors are covered by software testing, a large number of test adequacy criteria [22, 1] have been proposed.

*Corresponding author

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950313

To help developers to explore more behaviors of a program, traditional work [16, 12, 10, 17, 33] usually focused on the automatic test generation methodology – automatically generating more tests with high test adequacy. To our knowledge, symbolic execution based test generation [28, 10, 5] and search-based test generation [36, 1] are among the most competitive test generation techniques, both seeking high test adequacy by generating specific tests. However, although traditional automatic test generation approaches are widely used in generating new tests from scratch or augmenting existing tests [47, 56], they have various well-known challenges to be solved. For example, symbolic execution approaches have trouble dealing with program paths related to library code, structurally complex objects, strings, arrays, loops, etc. [54, 55], whereas search-based approaches suffer greatly from the "flat fitness landscape" problem (i.e., the fitness function produces the same fitness value for different tests) [30]. Due to these challenges, traditional automatic test generation techniques can hardly achieve high test adequacy, leaving potential abnormal behaviors undetected.

Different from prior work, in this paper we present a new idea – **Isomorphic Regression Testing**, which detects abnormal behaviors of the program under test by creating its variants (i.e., modified programs) instead of generating tests. Isomorphic regression testing is proposed based on the hypothesis that the behaviors of some variants of the program under test can be regarded as the program's extended behaviors, and thus can also be explored to help detect faults. Based on this hypothesis, when making some identical modifications on two versions of programs, the behaviors of the two modified programs would be the same, unless there are faults (or changes) in the new version. In particular, isomorphic regression testing forces the existing tests to execute the originally uncovered code by modifying the isomorphic code (i.e., identical code fragments) of two program versions. The behaviors[1] of the modified programs are then compared to check if abnormal behaviors are induced in the new version. It is worth mentioning that since isomorphic regression testing detects faults through the extended behaviors of a program, there is a threat that some extended behaviors may be unreliable. This threat can be alleviated by executing the program variants for multiple times, and the tests with unreliable behaviors can be skipped via further analysis.

In this paper, we propose *ISON* (**ISO**morphic regression testing through **N**egation) – the first step to implementing isomorphic regression testing which generates program vari-

[1]Program behaviors can be expressed through test outputs.

ants by negating branch conditions. With *ISON*, the current tests would be forced to execute the branches that are originally uncovered, and thus can assist in revealing faults. To evaluate the effectiveness of *ISON*, we applied it to 10 real-world Java projects downloaded from Github. The results show that *ISON* enhances the test effectiveness of these projects by executing 5.3% to 80.0% uncovered branches. Besides, the experimental results show that *ISON* detects a number of faults that cannot be detected by state-of-the-art *EvoSuite* in regression testing.

The idea of **Isomorphic Regression Testing**, including its implementation *ISON*, has the following inherent innovation. First, it is the first attempt in software testing that detects abnormal behaviors of a program without test augmentation. In particular, it strengthens the effectiveness of an individual test without modifying the test itself. Second, it presents a new type of test oracles, which determines whether the program under test behaves as expected by comparing the execution results of this program and its variants. Actually, this type of test oracles can be viewed as an extension of metamorphic relations [8]. Third, it implies a new application of mutation testing [25, 60, 61] for fault detection, since the program variants generated by *ISON* are one type of mutants (i.e., generated by *branch negation* operator). Finally, *ISON* can also be treated as a lightweight variant of traditional symbolic execution [29, 9, 28], i.e., systematic path exploration without expensive path constraint solving. The paper makes the following contributions:

- An **Isomorphic Regression Testing** methodology to detect abnormal program behaviors, which generates program variants instead of tests for exploring uncovered program behaviors.
- An ***ISON*** approach that implements isomorphic regression testing through systematically generating program variants by negating branch conditions.
- A thorough evaluation on 10 real-world projects, which indicates that *ISON* can additionally execute up to 80% originally-uncovered branches, and detect up to 69.6% originally-undetected faults.
- An empirical comparison and combination with state-of-the-art automated test generation approach, *EvoSuite*, demonstrating the promising future of applying *ISON* in detecting faults in regression testing.

2. ISON

In this section, we first illustrate *ISON* with a motivating example in Section 2.1 and then introduce the three steps of *ISON* from Section 2.2 to Section 2.4.

2.1 Motivating Example

Figure 1 shows two versions of an example program, which is adapted from the prior work [10] in symbolic execution. This program checks if the element in an array with a certain index is 5. If it is, the program returns the element's next element, otherwise, the program returns 0. However, developers induce a fault in developing version P', i.e., changing the fourth element of array *numCreated* from 4 to 5. To guarantee the quality of the new program (shown by Figure 1(2)), developers write some test inputs $\{t_1 = 0, t_2 = 1, t_3 = 2\}$ for method *indexParam*, which cover only the false branch of *indexParam* (i.e., Lines 10). Note that in this case, both t=3 and t=4 will cover Line 9, while only t=3 can reveal the fault in Line 2. To test version P', automatic test generation approaches can be used to generate new tests. However, symbolic-execution based test generation tools fail to cover method *indexParam* [10] (i.e., array *numbers* is unknown). Although search-based test generation tools are able to generate tests, they may seek only code coverage and omit the proper tests (i.e., t=3) to reveal the fault. For example, *EvoSuite*[2] generates 6 tests for this program, which only invoke method *indexParam* with the parameter value of 4 (i.e., t=3 is not included in the 6 tests).

(1) old version P

```
1  public static int[] createNumbers() {
2      int[] numCreated = new int[] {1,2,3,4,5,6,7,8,9,10};
3      return numCreated;
4  }
5
6  public static int indexParam(int index) {
7      int[] numbers = createNumbers();
8      if (numbers[index] == 5){
9          return numbers[index+1];
10     } else{ return 0;}
11 }
```

(2) new version P'

```
1  public static int[] createNumbers() {
2      int[] numCreated = new int[] {1,2,3,5,5,6,7,8,9,10,11};
3      return numCreated;
4  }
5
6  public static int indexParam(int index) {
7      int[] numbers = createNumbers();
8      if (numbers[index] == 5){
9          return numbers[index+1];
10     } else{ return 0;}
11 }
```

Figure 1: Motivating example.

Different from traditional automatic test augmentation approaches, *ISON* executes the uncovered branch through generating program variants. Figure 2 shows the basic process: when aiming to test program P', first, *ISON* compares the methods of program P and P' through their CFGs, and detects that P and P' have identical code portion (i.e., method *indexParam*), as shown in Figure 2(1); second, *ISON* detects that the true branch of P' in Line 9 is uncovered after executing the existing tests, as shown in Figure 2(2); third, for both program P and P', *ISON* negates a branch condition (i.e., node L8, in Line 8 of Figure 1) so as to create program variants P_m and P'_m (i.e., which would return the next element responding to each index except element 5), and forces the tests to execute the true branch, as shown in Figure 2(3); finally, *ISON* compares the outputs of program variants against each test (i.e., $t(P_m)$ and $t(P'_m)$), and detects a different behavior through test t_3=*2*, as shown in Figure 2(4). As program P and P' behave equally with the existing tests before negation, the different behaviors detected after negation are exposed by the newly executed branch, and may indicate a potential fault in the newly developed program P'[3].

[2]A popular search-based test generation tool for Java (http://www.evosuite.org/).

[3]Also, in this example, if the developers change the third element from 3 to 2 by mistake, no test augmentation approaches can detect the fault, while *ISON* can.

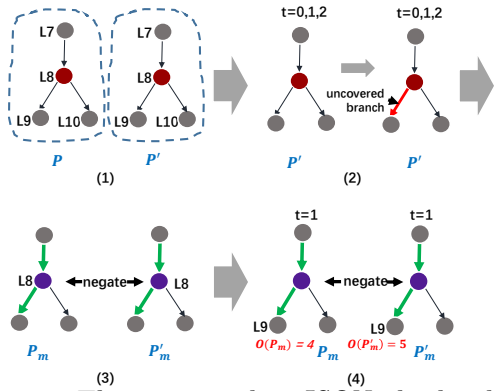

Figure 2: The process that *ISON* checks the motivating example. Each subgraph is a control flow graph (i.e., CFG) of method *indexParam*. Red nodes represent branch conditions. Purple nodes represent negated conditions. Labels beside each node indicate the line number of the corresponding code.

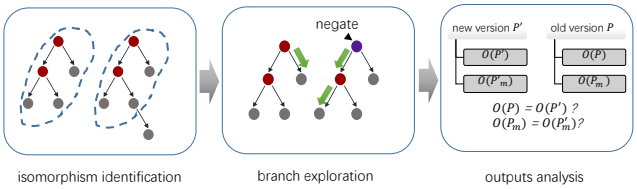

Figure 3: Overview of *ISON*

From the process above, *ISON* contains three steps, as Figure 3 shows.

- **Step1: Isomorphism identification**. *ISON* constructs a CFG for each method of P and P' so as to identify the matched code portions inside each method between P and P'. More details are in Section 2.2.
- **Step2: Branch exploration**. For each pair of matched code inside each method, *ISON* systematically negates their branch conditions to generate program variants to cover the uncovered branches of P'. More details are in Section 2.3.
- **Step3: Output analysis**. *ISON* compares the outputs of P and P' as well as the outputs of their program variants so as to identify the different behaviors induced from P to P'. Developers may further check whether such a different behavior indicates a fault. More details are in Section 2.4.

2.2 Isomorphism Identification

Isomorphic regression testing requires identical modifications on the program P' and its previous version P, to make sure that their behaviors would be changed in the same way and thus are comparable. To ensure this, *ISON* first identifies isomorphic code (i.e., isomorphism) inside each method between these two programs. In this work, we use control flow graphs (CFGs)[4] [64] to identify isomorphic code. More concretely, *ISON* first identifies the methods of P and P' that share the same source code path, method names, and parameters, which will most probably contain isomorphic code. Then, for each pair of matched methods, *ISON* constructs their CFGs and makes a comparison to identify their

[4]In this paper, CFGs are generated with *Soot*: http://sable.github.io/soot/.

ALGORITHM 1: Isomorphism identification

```
   Input: P′: a program under test
   Input: P: previous version of P′
   Output: G: identical sub-CFGs in P and P′
   // get matched methods in P and P′
1  for m′ in P′ do
2      for m in P do
3          if m=m′ then
4              M ← M ⋃ < m, m′ >;        // M: {< m_i, m′_i >}
5          end
6          break
7      end
8  end
9  for each < m, m′ > in < M, M′ > do
10     U ← generateCFG(m)
11     U′ ← generateCFG(m′)
       // get matched sub-CFGs for each matched methods
12     N ← matchCFG(U.root, U′.root)
13     < g, g′ >← getSubGraph(U,U′,N)
14     G ← G ⋃ < g, g′ >;                // G: {< g_i, g′_i >}
15 end
```

Function matchCFG

```
   Input: U, U′: the CFGs of two matched methods
   Input: n, n′: the root node of U and U′
   Output: N: the set of matched node pairs inside each
           CFG pair
1  if n and n′ are compared then
2      return
3  else
4      mark n and n′ as compared
5  end
6  if n equals n′ then
7      mark n and n′ as matched
8      N ← N ⋃ < n, n′ > ;           // N: {< n_i, n′_i >}
9      for n_next, n′_next in the succeeding nodes of n, n′ do
10         matchCFG(n_next, n′_next)
11     end
12 end
```

mismatched nodes[5]. Note that any difference in two nodes would lead to mismatches, and all the subsequent nodes of a mismatched node would be regarded as mismatched, too. By removing the mismatched nodes, *ISON* constructs matched sub-CFGs for these matched methods, which make the final identified isomorphic code portion. Note that as the motivating example shows, the code that contains faults which are newly introduced in the new version will not be matched, but they may still be covered after negation.

The detailed process is shown in Algorithm 1. For any P and P', this algorithm produces a set of identical sub-CFGs in these two programs. In particular, Lines 1–8 are to find the matched method set M (i.e., $\{< m_i, m'_i >\}$) of P and P'. Only the methods with the same source code path, names, and parameters are regarded as matched (i.e., Line 3). The remaining lines are to identify the isomorphic code in these methods one by one. Within the loop, Lines 10 and 11 are to generate CFGs U and U' for each pair of matched methods $< m, m' >$. Line 12 is to compare CFGs U and U' so as to identify their common node set N, through function *matchCFG*. Line 13 is to take out the common sub-CFG from U and U' based on the common node set N.

[5]Each node is a Soot unit corresponding to a statement.

Furthermore, we present the detailed implementation of function *matchCFG*. Taking the root nodes of two CFGs as inputs, this function compares each pair of corresponding nodes $< n_i, n_i' >$ through CFG traverse, and produces the common node set N. Specifically, Lines 1–5 are to avoid comparing node pairs that have been compared before. Line 6 judges if two nodes are matched (i.e., the two nodes must have exactly the same content). If yes, the node pair would be remarked as matched, and put into the node set N (i.e, $\{< n_i, n_i' >\}$) (i.e., Line 8), while the function goes on traversing each CFG (e.g., depth-first[6]) and comparing the successor nodes of n and n' (i.e., Line 9–11). If no, this function stops visiting the current node, and backtracks.

2.3 Branch Exploration

In this section, we present how *ISON* negates branch conditions so as to explore uncovered branches. For the matched sub-CFGs in P and P', *ISON* first executes P and P' with a common test set and identifies the branches that are not covered, then removes the branches that cannot be explored even after negation, and generates program variants P_m and P'_m by negating one branch condition at a time. The program variants that are generated by negating n conditions are called **n-depth-negation program variants**. After generating the 1-depth-negation program variants, *ISON* checks the uncovered branches in the program variants, filters them, and goes on generating the 2-depth-negation program variants, and so on, until all the possible branches are executed. The outputs of P and P', as well as all their variants P_m and P'_m, are recorded for output analysis, which is the next step of *ISON* and will be introduced in Section 2.4.

Algorithm 2 shows the detailed process. The algorithm takes the matched sub-CFGs G as well as a common test set T of P and P' as inputs, generates program variants depth by depth, and produces the outputs of P, P' and their program variants. Specifically, the algorithm first generates the **1-depth-negation program variants**, which can be divided into the following 3 steps.

- **Step1: Collect original outputs**. Lines 3 and 4 are to execute the original program P and P' against each test so as to collect outputs $O(P)$ and $O(P')$ (Lines 5 and 6), as well as the uncovered branches B_u and B'_u.
- **Step2: Prepare negation queue**. In Line 2, negation queue Q is initialized, which is a collection of negation sets. Each negation set is composed of the conditions to negate for executing an uncovered branch (Line 10). To construct each negation set, the algorithm identifies those uncovered branches that (1) have conditional statements within the matched sub-CFGs (i.e., G) between two versions, (2) can be executed after negation, i.e., the branches whose sibling branch[7] is covered, and (3) are not covered by other test cases by filtering B_u and B'_u (Lines 7 and 8). Note that the third condition is optional (more details in Section 3.3). The constructed negation sets are then put into the negation queue to wait for negation (Line 11) to generate the 1-depth-negation program variants.

[6]We adopt the same traversing strategy on each CFG to make sure that the matched nodes also have the same position.

[7]The branch that shares the same branch condition with this branch.

ALGORITHM 2: Branch exploration

```
   Input: G: the set of matched sub-CFGs of P and P'
   Input: T: a common test set of P and P'
   Output: O(P),O(P_m): outputs of P and its variant P_m
   Output: O(P'),O(P'_m): outputs of P' and its variant P'_m
1  for each t in T do
2      Q← ∅ ;                          // initialize queue Q
       // get original outputs and uncovered branches
3      t(P), B_u ← execute(P, t);
4      t(P'), B'_u ← execute(P', t);
5      O(P) ← O(P) ⋃ t(P)
6      O(P') ← O(P') ⋃ t(P')
7      B_f ← filter(B_u,G);      // filter uncovered branches
8      B'_f ← filter(B'_u,G);
9      for each n in B_f or B'_f do
10         S ← <n.condition>;      // 1-depth negation set
11         Q.push(S);
12     end
13     while Q is not empty do
14         curNegSet ← Q.poll();    // current negation set
15         curDepth ← size of curNegSet;  // negation depth
16         P_m ← negate(P,curNegSet);
17         P'_m ← negate(P',curNegSet);
           // get new outputs and uncovered branches
18         t(P_m), B_u ← execute(P_m, t);
19         t(P'_m), B'_u ← execute(P'_m, t);
20         O(P_m) ← O(P_m) ⋃ t(P_m)
21         O(P'_m) ← O(P'_m) ⋃ t(P'_m)
22         B_f ← filter(B_u,G);
23         B'_f ← filter(B'_u,G);
24         for each n in B_f or B'_f do
25             S_next ← curNegSet ⋃ n.condition ;  // depth+1
26             Q.push(S_next);
27         end
28     end
29 end
30 return O(P),O(P_m),O(P'),O(P'_m)
```

- **Step3: Generate and execute program variants**. For each negation set in the negation queue, Lines 16 and 17 are to generate program variants, and Lines 18 and 19 are to collect the new outputs as well as the newly uncovered branches.

After generating the 1-depth-negation program variants, some branches may remain un-executed, which need variants with multi-depth negation (i.e., the variants generated by negating more than one conditions at the same time). To solve this problem, the algorithm updates the uncovered branches and further filters branches (through Lines 18–23) after 1-depth-negation, and prepares a negation set for each updated identified branch. The process above is repeated, until there is no more negation set added. Note that a negation set may contain more than one condition, which should be negated together so as to generate a program variant executing the corresponding branch. The largest size of the negation sets is called **negation depth**. Also, as usually there is more than one negation set in the negation queue, a good negation order may help to find more faults in limited time. For *ISON*, there are many possible negation strategies, such as choosing negation set randomly, or prioritizing negation sets. In the future, we will systematically investigate how negation strategies affect the effectiveness of *ISON*.

2.4 Output Analysis

To observe the behavior difference between P and P' so as to identify possible faults in P', *ISON* records the test

outputs of P, P' and their program variants P_m and P'_m[8] through the branch exploration process. More specifically, for each common test t, its preceding four outputs are denoted as $t(P)$, $t(P')$, $t(P_m)$ and $t(P'_m)$, which can be compared to expose newly induced behaviors in P.

Under the scenario of regression testing, when $t(P)! = t(P')$, test t directly detects different behaviors between P and P' without the help of *ISON*. However, when $t(P) = t(P')$, the branches covered by t fail to expose any differences between P and P', while the uncovered branches are unexplored. With this prerequisite, *ISON* compares $t(P_m)$ and $t(P'_m)$: if they are equal, the newly executed branches do not expose any differences, neither; otherwise, some different behaviors are propagated through and only through the newly executed branches in program variants. To determine whether the different behaviors imply any faults, the same with existing automatic test generation/augmentation techniques [47, 56, 12, 13], developers or testers may perform manual inspection.

3. EXPERIMENTAL SETUP

In this study, we address four research questions.

- **RQ1:** What's the overall effectiveness (in terms of program coverage improvement and fault detection improvement) of *ISON*?
- **RQ2:** How does *ISON* perform in improving each test's effectiveness?
- **RQ3:** How does negation depth affect the effectiveness of *ISON*?
- **RQ4:** How does *ISON* perform comparing to traditional automated test generation approaches?

3.1 Tools

***ISON* implementation tool**. *ISON* has been implemented as a prototype for Java programs with JUnit tests. In implementation, we use *Soot program analysis framework*[9] for isomorphism identification, *ASM bytecode manipulation engine*[10] for branch negation and output analysis. The source code of *ISON* is available on our homepage[11].

Mutant generation tool. We use *Major*[12] to generate mutants in the evaluation, because most mutation tools (e.g., Javalanche[13] and Pitest[14]) do not produce mutants with accessible source code whereas *Major* does. On the other side, *Major* is the only mutation tool whose mutants are proved to simulate real faults well [26].

Test generation tool. We compare our approach with state-of-the-art test generation tool *EvoSuite*[15] in our evaluation. *EvoSuite* is a search-based test generation tool which is capable to generate test oracles and deal with various cases that other test generation techniques/tools cannot handle [12]. Moreover, it has been reported to be one of the most practical and robust test generation tools [10]. We also tried to compare our technique with state-of-the-art symbolic execution engine for Java, *JPF-symbc*[16], which has been widely used in the literature [53, 41, 66]. However, *JPF-symbc* is not directly applicable to our evaluation benchmarks due to some limitations of symbolic execution. Thus, we omit the empirical comparison with *JPF-symbc*.

3.2 Subjects, Faults, and Tests

In our experiments, we randomly chose 10 real-world Java projects with different sizes of source code and tests from the top-1000 most popular Java projects in GitHub in June 2015. We ensured that the selected subjects can be successfully processed by *EvoSuite* and *Major*. For each project, we used its two continuous versions to simulate the real regression testing scenario, and the new version is regarded as the project under test.

For each project, we seeded mutants using *Major* to simulate faulty programs [26, 2]. Similar to previous work [34], we randomly selected 200 mutants per-project. For the project generating less than 200 mutants, we used all their mutants. These mutants were viewed as faults in this study.

For each subject, we constructed a common test set for its two versions to compare the outputs of the same test (following Section 2.3). In particular, the original tests of the new version were regarded as the base test set, from which we removed the tests that cannot run through on the old version. Then, we further removed the tests that produce non-deterministic outputs in two steps: (1) we manually removed the tests that will definitely produce non-deterministic outputs, e.g., tests that return the current time, and (2) we automatically checked the remaining tests by running each of them 5 times and comparing their outputs to ensure their determination, as previous work does [49].

Table 1 shows the basic information of the subjects (i.e., the versions under test). "LOC" shows the number of lines of executable source code calculated by *LocMetrics*[17]. "Tests" refers to the number of tests actually used in the study. "B_{all}/B_{cov}" refers to the total number of branches and number of branches that are covered by the tests. M_{all}/M_{kill}" refers to the total number of mutants (or faults) and the number of mutants that are killed by the tests. From the table, we use subjects of various sizes, whose LOC ranges from 258 to 23,293. Also, these subjects have various proportion of covered branches as well as killed mutants.

Table 1: Subjects, faults, and tests

Subjects	LOC	Tests	B_{all}/B_{cov}	M_{all}/M_{kill}	Ver.
cors-filter	1,198	72	146/138	195/122	1.0.1
digester	23,293	184	1,432/885	200/84	3.2
evo-inflector	465	4	26/19	105/54	1.2.1
gelfj	1,416	27	216/107	200/41	1.1.12
gson-fire	895	31	194/114	200/92	1.3.1
hashids	258	11	74/46	200/135	1.0.1
jackson	654	43	154/23	200/48	1.5.1
java-jwt	422	55	104/93	158/89	2.1.0
jopt-simple	4,292	640	378/368	200/144	4.4
scrypt	467	20	116/48	200/93	1.4.0

3.3 Independent Variables

We consider the following two independent variables:

IV1: Negation list. For each test, it may execute program variants that are generated based on negating the conditions of two kinds of branches: 1) branches uncovered by the whole test set: 2) branches that are uncovered by this

[8]More discussion about the reliability of P_m and P'_m is in Section 3.5 and Section 5.
[9]http://sable.github.io/soot/
[10]http://asm.ow2.org/
[11]http://github.com/sei-pku/Ison
[12]http://mutation-testing.org/
[13]http://www.javalanche.org/
[14]http://pitest.org
[15]http://www.evosuite.org/
[16]http://babelfish.arc.nasa.gov/trac/jpf
[17]http://www.locmetrics.com/

test. We define the first collection of branches as *VL* (i.e., uniVersal negation List), which is the same for all the tests. Similarly, we define the second collection as *QL* (i.e., uniQue negation List), which is unique for this test alone. For *VL*, the whole test set should be executed on the program under test all together beforehand, to prepare for a universal negation queue (i.e., storing the conditions that should be negated for each branch in *VL*), and to generate a universal set of program variants. The program variants are then executed by each test to collect outputs. For *QL*, each test is executed against the program under test to prepare for its own negation queue and program variants. The program variants are executed by this test only. Note that the size of *VL* is smaller and may be much smaller than *QL* since the branches not executed by one test could be executed by other tests in the suite.

IV2: Negation depth. We generate program variants of different negation depth in our evaluation (more details in Section 2.3). Note that as running program variants with multi-depth would increase the cost of *ISON*, we set the default configuration of *ISON* as using only the 1-depth-negation program variants.

3.4 Measurements

To measure the effectiveness of *ISON*, we define the following measurements.

- **Number and proportion of additionally executed branches**, which refer to the number of additionally executed branches among the originally uncovered branches, represented as B_{ISON}, and the proportion of additionally executed branches against the originally uncovered branches, represented as B_{prop}. These two measurements indicate the branch coverage improvement (i.e., the ability of executing more branches) of *ISON*.
- **Number and proportion of additionally detected faults**, which refer to the number of additionally detected faults among those originally-undetected faults, represented as M_{ISON}, and the proportion of additionally detected faults against the originally-undetected faults, represented as M_{prop}. These two measurements indicate the fault detection improvement (i.e., the ability of revealing more faults) of *ISON*.

3.5 Procedure

In our experiments, we aim to evaluate how *ISON* performs in executing the branches that the existing tests fail to cover, and in detecting the faults that the existing tests fail to detect. Therefore, we regard each undetected faulty version (i.e., mutant) of each project as a program under test, and apply *ISON* on it with the default configuration. In particular, first, for each faulty version, we run the tests against it to collect the two kinds of negation lists (i.e., *VL* and *QL*), and record the additionally executed branches and reported different test outputs (i.e., the outputs checked by test assertions). Second, to learn the impact of negation depth, we further apply our approach with different negation depth to the subjects that require further negation. Third, we compare *ISON* with *EvoSuite*, a state-of-the-art traditional automated test generation tool.

Similar to *EvoSuite*, the different behaviors would be caused by either changes or faults. To investigate how many faults *ISON* can help to detect, we differentiate faults from changes in the following way: for each original new version before mutation and its faulty version after mutation (i.e., the program under test), we compare their outputs after negation. If their outputs are the same, the reported different behaviors (i.e., between the variants of the two versions) are actually due to changes. Otherwise, they are due to the fault[18].

At the same time, we make efforts to reduce two major threats in the experiments. First, since *ISON* changes the source code of the program under test, a threat is that *ISON* may generate program variants with unreliable behaviors (i.e., a program variant may behave differently given the same test input). To reduce this threat, we re-execute each program variant with each test five times to make sure that all their test outputs are reliable. Second, we also run *ISON* on the original program of the new version to check if *ISON* would report false alarms for correct programs. For the negations that would generate unreliable behaviors or false alarms, we abandon them, so that the experimental results will not be affected (see more in Section 4.1.2).

3.6 Threats to Validity

Threats to internal validity. The threats to internal validity are mainly concerned with the potential faults in our implementation. To reduce the threats, we used the widely used Soot static analysis framework as well as the ASM bytecode manipulation engine to implement *ISON*. Also, the program variants generated by *ISON* may have unreliable behaviors, which may affect the comparison results when detecting faults. To reduce this threat, we re-executed each program variant 5 times and compared their test outputs to ensure that all the behaviors of program variants we used are stable. In addition, we also carefully reviewed and tested our implementation to ensure its correctness.

Threats to external validity. The main threat to external validity is that our findings may not be generalizable to other programs, tests, and faults. To reduce this threat, we randomly chose 10 real-world Java programs with original JUnit tests from GitHub, and also used many mutation faults to evaluate the test effectiveness. However, 10 subject programs are still not representative of other programs. In addition, although mutation faults have been shown to be suitable for simulating real faults in software testing experimentation [26, 2], our use of mutation faults may still pose potential threats to external validity. Further reduction of this threat requires more study on more subject programs with real faults.

Threats to Construct validity. The main threat to construct validity lies in the metrics that we used to assess the effectiveness of the proposed technique. To reduce this threat, we used the widely used branch coverage, as well as fault-detection capability to measure the test effectiveness.

4. EXPERIMENTAL RESULTS

In this section, we present and analyze the experimental results to answer the four research questions.

4.1 RQ1: Overall Effectiveness

4.1.1 Effectiveness in Executing More Branches

Table 2 presents the effectiveness of *ISON* in terms of branch coverage (on the original new version), where Col-

[18]In practice, however, the changes and faults have to be differentiated manually, as the correct version is unknown.

umn "B_{uncov}" presents the number of uncovered branches, Column "B_{ISON}" presents the number of branches executed by *ISON* among "B_{uncov}", Column "B_{prop}" presents the proportion of "B_{ISON}" against B_{uncov}. In this table we do not present the results of *ISON* with *VL* and *QL* separately because their results are equal for the studied subjects. Note that in theory, *ISON* with *VL* and *QL* can have different results since *ISON* with *QL* may negate more conditions for each test and execute more additional branches through the negation.

From the table, *ISON* additionally executes 5.3% to 80.0% uncovered branches. On only one subject (i.e., *jackson*) *ISON* executes less than 14.0% additional branches. We checked the source code of *jackson*, and found that most of its class files have 0% code coverage with the original tests, which means that the sibling branches of most originally uncovered branches in *jackson* are also not covered. Such condition is beyond *ISON*'s capability. In summary, *ISON* with the default 1-depth negation (using *VL* or *QL*) performs well in executing uncovered branches.

Table 2: Effectiveness in branch execution

Branch execution results				
Subjects	B_{all}	B_{uncov}	B_{ISON}	B_{prop}
cors-filter	146	8	4	**50.0%**
digester	1432	547	318	**58.1%**
evo-inflector	26	7	1	**14.3%**
gelfj	216	109	55	**50.5%**
gson-fire	194	80	20	**25.0%**
hashids	74	28	17	**60.7%**
jackson	154	131	7	**5.3%**
java-jwt	104	11	4	**36.4%**
jopt-simple	378	10	8	**80.0%**
scrypt	116	68	43	**63.2%**

4.1.2 *Effectiveness in Detecting More Faults*

Table 3 presents the effectiveness of *ISON* in terms of fault detection[19], where Column "M_{unkill}" presents the number of faults undetected by the existing tests, Column "M_{ISON}" presents the number of faults detected among M_{unkill} with *ISON*, and Column "M_{prop}" presents the proportion of additionally detected faults against "M_{unkill}".

From the table, with *VL*, no more faults are detected for *cors-filter*, *java-jwt*, or *scrypt*, while 1.8%∼30.8% faults are additionally detected for the other 7 subjects. Also, with *QL*, 5.3%∼69.6% faults are additionally detected.

In summary, *ISON* can detect additional faults with either *VL* or *QL*. What's more, compared with *VL*, *ISON* with *QL* increases the same number of uncovered branches, but detects more faults. The reason is that *ISON* with *QL* generates more program variants and strengthens more tests' effectiveness. This finding further confirms a recent work on test effectiveness [23] that code coverage does not necessarily have strong relationship with fault-detection capability.

4.2 RQ2: Effectiveness on Individual Tests

According to the observations on Tables 2 and 3, *ISON* with *QL* may significantly strengthen the fault-detection effectiveness of each test, as it focuses on making each test cover more branches. In this section, we further investigate the effectiveness of *ISON* (i.e., with *QL*) on an individual test by randomly choosing 3 tests per subject and running each test against each faulty version (as we did in Section 4.1.2). Note that some tests may detect duplicate faults (i.e., faults that other tests also detect) in this study. To learn whether each test contributes to the total number of detected faults, we also identify the number of unique faults, i.e., faults that can only be additionally detected by this test. The effectiveness results of individual tests are shown in Figure 4. The top sub-figure shows the effectiveness in terms of branches, where each bar represents a test, the total height of each bar represents the total number of executed branches with the help of *ISON*, and the height of the green bars represents the number of executed branches without *ISON*. The bottom sub-figure shows the effectiveness in terms of fault detection, where the total height of each bar represents the total number of detected faults with the help of *ISON*, the height of the blue bars represent the number of detected faults without *ISON*, and the height of the green bars represent the number of unique faults that cannot be additionally detected by other tests.

From the table, *ISON* does strengthen the effectiveness of most tests. For example, for project *cors-filter*, with *ISON* test t_1 executes about 80% more branches, and de-

[19]The distribution and analysis of changes and faults can be referred to in Table 4, Section 4.4.

Table 3: Effectiveness in fault detection

Fault detection results with *VL*				
Subjects	M_{all}	M_{unkill}	M_{ISON}	M_{prop}
cors-filter	195	73	0	**0.0%**
digester	200	116	22	**19.0%**
evo-inflector	105	51	8	**15.7%**
gelfj	200	149	5	**3.4%**
gson-fire	200	108	7	**6.5%**
hashids	200	65	20	**30.8%**
jackson	200	152	3	**2.0%**
java-jwt	158	69	0	**0.0%**
jopt-simple	200	56	1	**1.8%**
scrypt	200	92	0	**0.0%**
Fault detection results with *QL*				
Subjects	M_{all}	M_{unkill}	M_{ISON}	M_{prop}
cors-filter	195	73	15	**20.5%**
digester	200	116	22	**19.0%**
evo-inflector	105	51	8	**15.7%**
gelfj	200	149	9	**6.0%**
gson-fire	200	108	12	**11.1%**
hashids	200	65	25	**38.5%**
jackson	200	152	8	**5.3%**
java-jwt	158	69	33	**47.8%**
jopt-simple	200	56	3	**5.4%**
scrypt	200	92	64	**69.6%**

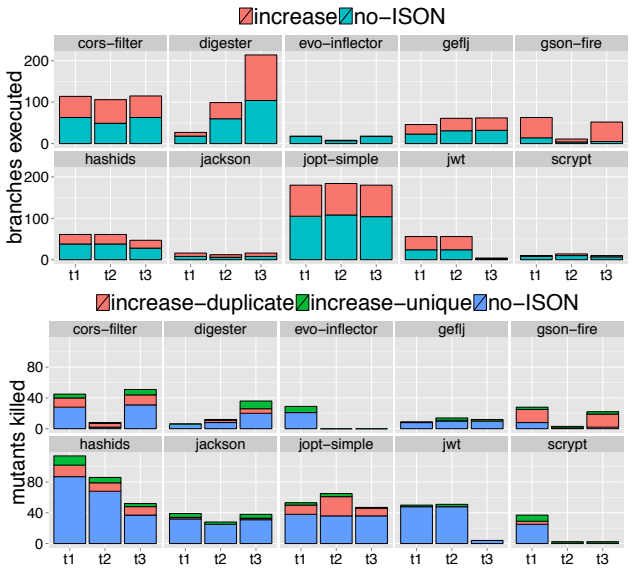

Figure 4: Effectiveness in Strengthening Each Test

```
1   private void processPayloadOptions(Map<String,
        Object> claims, Options options) {
2     long now = System.currentTimeMillis() / 1000l;
3       ...
4
5     if (options.isJwtId())
6       claims.put("jti", UUID.randomUUID().toString());
7   }
```

Figure 5: Code snippet of *java-jwt*.

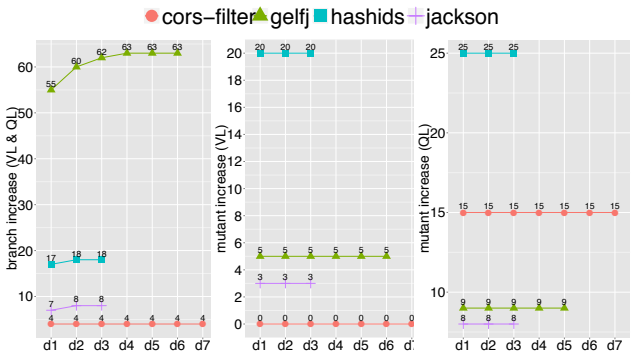

Figure 6: Results with different negation depth

tects about 60% more faults. Furthermore, t_1 also detects 5 unique faults that other tests cannot detect. In the future, we will further investigate what types of tests tend to detect more faults with *ISON*.

Besides, as we mentioned in Section 3.5, to reduce the threats in our experiments, we removed the negations that would generate unreliable behaviors beforehand. To illustrate, one such example for project *java-jwt* is shown in Figure 5. For this code snippet, none of the tests used in the study covers the true branch of Line 5. To force these tests execute Line 6, *ISON* negates condition *options.jsJwtId()*. This negation, however, triggers function *randomUUID()*, inside which random bytes would be created and returned, and then produce random outputs for the test inputs.

4.3 RQ3: Influence of Negation Depth

In this study, we aim to investigate how negation depth affects the effectiveness of *ISON*. Especially, the same as our last study, we investigate 8 projects excluding *digester* and *jopt-simple* (i.e., due to the cost issue). Among the remaining eight subjects, only *cors-filter*, *gelfj*, *hashids*, and *jackson* have uncovered branches that need multi-depth negation (i.e., Lines 22 and 23 of Algorithm 2 always return empty sets), so we present only the results of these 4 subjects.

Figure 6 shows how the numbers of additionally executed branches (shown in the first sub-figure) and additionally detected faults (shown in the second and third sub-figures, with *VL* and *QL* respectively) increase as the negation depth increases. For each sub-figure, the x axis represents negation depth. The number of points in each line represents the deepest depth of each project. From the first sub-figure, when the negation depth increases, no more additional branches are executed for *cors-filter*; for the other 3 subjects, the number of executed branches increases at first, then becomes saturate. From the second and third sub-figures, when the negation depth increases, the number of detected faults with *VL* remains the same for all the 4 subjects. Considering all the observations above, we get the conclusion that negation depth may affect the branch-execution effectiveness, but barely affects the fault-detection effectiveness. One potential reason is that few faults exist in the branches that can be executed through multi-depth negation. Another reason is that when increasing a negation depth, *ISON* explores new behaviors of the program, while the key program behaviors may have been explored through during the first-depth negation, thus the remaining exploration becomes less useful.

4.4 RQ4: Comparison with the Traditional Approach

In this study, we compare *ISON* with a state-of-the-art automated test generation approach: search-based test generation. In particular, we use the most widely used tool: *EvoSuite*. Furthermore, we regard each faulty version that is not detected by the existing tests as a program under test.

Under the scenario of regression testing, *EvoSuite* generates test inputs based on the old version, while regarding the corresponding test outputs as the possible oracles. Thus, when a test fails on a new version, it reports different behaviors between two versions, which may also be caused by either changes or faults, and we use the same way to differentiate faults from changes as introduced in Section 3.5.

In our experiments, we first use *EvoSuite* with the default setup[20] to generate tests on the old version, then run these newly generated tests against each faulty version to collect actual outputs (i.e., data checked in assertions) that can be used to find different behaviors, and against the original new version (before mutation) to collect the expected outputs that can be used to differentiate whether the changes or the faults induce the different behaviors.

The comparison results are shown in Table 4. Column "B_{uncov}" presents the number of branches uncovered by the original test suite. "B_{Evo}" and "B_{ISON}" represents the number of branches additionally executed by *EvoSuite* and *ISON*, respectively. "M_{Evo}" represents the number of faults that are additionally detected by *EvoSuite*. "M_{ISON}-VL" and "M_{ISON}-QL" represent the numbers of faults that are additionally detected by *ISON* with *VL* and *QL*, respectively. Moreover, the numbers inside the brackets represent the number of faults that are detected by *ISON* but not by *EvoSuite*. Column "Merged" lists the number of faults that are additionally detected by either *EvoSuite* or *ISON* in total. Columns "Change/Fault" show the total number of different behaviors induced by benign changes and faults, respectively, for both *ISON* and *EvoSuite*. Note that to ensure the accuracy of each judgement with *ISON*, we only manually checked every different behavior returned by *ISON* with *VL*, and thus do not present the unconfirmed results with *QL* (due to high time cost for such manual checking).

From the table, when considering branch execution improvement, *EvoSuite* performs better than *ISON* for 7 out of 10 projects (i.e., except *gelfj*, *hashids*, and *jopt-simple*). When considering fault detection improvement, for most of the projects, *EvoSuite* detects more faults than *ISON* with *VL*, but fewer faults than *ISON* with *QL*. What's more, *ISON* with either *VL* or *QL* detects many faults that *EvoSuite* missed. For example, for *gson-fire*, among the 108 faults unexposed by existing tests, *EvoSuite* detects 6 faults, while *ISON* with *VL* detects 7 faults, all of which are not detected by *EvoSuite*; *ISON* with *QL* detects twice as many

[20]We also tried the setup with doubled test-generation time limit, and found that test-generation time does not have much effect on the number of generated tests.

Table 4: Comparison with *EvoSuite*

Subjects	Branch Execution			Fault Detection					Change/Fault	
	B_{uncov}	B_{Evo}	B_{ISON}	M_{unkill}	M_{Evo}	M_{ISON}-VL	M_{ISON}-QL	Merged	*EvoSuite*	*ISON*
cors-filter	8	8	4	73	1	0(0)	15(14)	15	0/9	0/0
digester	547	226	138	116	7	22(20)	22(20)	27	116/63	921/130
evo-inflector	7	4	1	51	9	8(0)	8(0)	9	0/77	0/16
gelfj	109	26	55	149	33	5(4)	9(8)	41	1271/60	0/17
gson-fire	80	20	20	108	6	7(7)	12(12)	18	1620/36	1/35
hashids	28	0	17	65	23	20(9)	25(14)	37	0/58	0/235
jackson	131	110	7	152	8	3(3)	8(8)	16	0/114	0/27
java-jwt	11	9	4	69	0	0(0)	33(33)	33	0/62	0/0
jopt-simple	10	4	8	56	30	1(0)	3(2)	30	168/188	0/1
scrypt	68	55	18	92	64	0(0)	64(0)	64	0/8	0/0

as faults as *EvoSuite* does. In summary, *EvoSuite* outperforms *ISON* in executing new branches, while *ISON* with *QL* outperforms *EvoSuite* in fault detection for most of the projects. The reason is that test coverage is not directly related to fault-detection effectiveness, as shown in previous work [23] and our motivating example. Besides, *EvoSuite* is developed based on search-based test generation techniques, and has challenges in choosing tests with the equal fitness function, while *ISON* does not have this problem. Nevertheless, both *ISON* with *VL* and *QL* can be combined with *EvoSuite* to further achieve stronger fault-detection capability.

Furthermore, the results in this table indicate that both *EvoSuite* and *ISON* have high proportion of changes for some projects: *EvoSuite* has very high ratio of changes for *gelfj* and *gson-fire*, we suspect the reason to be that a large number of tests generated by *EvoSuite* are actually related to the changes in these two projects; *ISON* has high ratio of changes for *digester*, and we suspect the reason to be that the changes in *digester* may be related to the newly executed branches closely. It is worth mentioning that although many different behaviors detected by automatic test generation approaches or our approach are changes, these approaches may still alleviate the burden of developers in assuring software quality, as manually generating tests is quite a tedious and laborsome task. According to the survey conducted by Fraser et al. [14], 58 out of 72 developers think that there is great need for test generation tools, and 48 out of 72 developers think that it is easier to confirm the program behaviors than manually adding them to the tests. The opinions above may also fit *ISON*. More discussion about the detected changes can be referred to Section 5.1.

In general, *ISON* reveals additional faults compared with *EvoSuite*. In practice, *ISON* can be applied in tandem with existing automated test generation techniques (e.g., *EvoSuite*) to further reduce potential regression faults.

5. DISCUSSION

5.1 Cost

In regression testing, for the test generation approaches, e.g., *EvoSuite*, the cost mainly lies in generating tests, running the newly generated tests, and manually confirming different behaviors[21]; for *ISON*, the cost mainly lies in creating program variants, executing these variants, and manually confirming different behaviors. Note that the cost of *ISON* may be further reduced in many ways, such as generating only effective program variants or prioritizing the execution order of program variants, filtering tests based on program variants, and executing program variants in parallel.

[21]To our knowledge, no current test generation tools can provide reliable oracles.

The same as *EvoSuite*, in regression testing, *ISON* returns the different behaviors between the version under test and its previous version, which may contain both faults and changes. Note that the different behaviors due to program changes can still be beneficial for software developers. For example, they can help with program change comprehension, which can in turn help with program debugging or preventing future bugs [12]. Even if the developers only aim to detect faults, under which scenario the changes would be regarded as false alarms or false positives, *ISON* can also relieve their stress of assuring software quality according to the work of Fraser et al. [14], as we discussed in Section 4.4. Moreover, under this circumstance, *ISON* (as well as other automatic test generation techniques) may be regarded as providing a practical tradeoff aiming to seek more soundness in ensuring the quality of software: to reduce the false negatives (i.e., faults that are not reported), while allow some false positives (i.e., alarms that are not faults).

5.2 Dimension

ISON opens new dimensions in the following aspects:

First, *ISON* expands the application scenario of mutation testing. As discussed in the introduction, *ISON* checks if the behaviors of program variants of one version are the same as those of another version. The program variants are in fact a kind of mutants that are generated by *branch negation*, a traditional mutation operator. Also, isomorphic regression testing is proposed based on the hypothesis that the behaviors of some program variants can be regarded as the program's extended behaviors, and thus can also be explored to help detect faults. From this perspective, some behaviors of the mutants (i.e., program variants) may be regarded as another form of *invariants* [11, 66] – invariants that would be held if the program is mutated in a specific way.

Second, *ISON* proposes a new kind of metamorphic relations [8, 58] and also contributes to the oracle problem. Traditional metamorphic relations refers to the relations among the results of multiple executions of a program, with different test inputs. In our approach, the relationship between different execution results before and after negation of two versions can also be regarded as a metamorphic relation. Specifically, for the current version, for each test, we have its output before negation $t(P')$, and the output after negation $t(P'_m)$; equally, for the old program version, we have $t(P)$ and $t(P_m)$. $t(P) == t(P')\&\&t(P_m) == t(P'_m)$ is a metamorphic relation which is supposed to be satisfied for a program with no changes and no faults.

Lastly, *ISON* can also be treated as a lightweight variant of symbolic execution. Traditional symbolic execution [29, 9, 28] explores all the possible program paths to cover all the possible program behaviors and then solves the path constraints with off-the-shelf constraint solvers. Similarly, our

ISON also explores all the possible program paths. However, our *ISON* does not require the expensive path constraint solving since *ISON* directly negate the branch conditions to force the exising test inputs to execute all the possible program paths. Therefore, *ISON* can be treated as lightweight symbolic execution without constraint solving.

5.3 Application

ISON has three major application scenarios: (1) to serve as an enhancer of the existing tests alone; (2) to be applied in tandem with test generation tools to enhance the existing tests; (3) to serve as an enhancer of the tests generated by test generation tools. The first two scenarios have been evaluated in this paper. The third scenario will be further investigated in our future work.

In particular, *ISON* can be applied in tandem with traditional automated test generation approaches considering the following two aspects. First, *ISON* can deal with many challenges faced by traditional automated test generation approaches. For example, traditional symbolic execution techniques usually have difficulty in dealing with program paths related to library code, structurally-complex objects, strings, arrays, loops, etc. [54, 55], while *ISON* is able to handle all of them. Second, traditional automatic test generation approaches usually seek high code coverage and omit some key tests that will reveal faults (i.e., as shown in our motivating example), while *ISON* can help to explore more behaviors of the program under test. Note that both *ISON* and traditional automatic test generation approaches may encounter code with non-determinism, i.e., code that would generate different outputs during different executions due to randomness or multi-thread. This threat can be removed by executing the code multiple times to ensure reliable outputs.

6. RELATED WORK

Our work is mostly related to automatic test generation and regression testing. Besides, our technique also belongs to the work [42, 67, 52] that modifies the source code to achieve better testing and debugging.

6.1 Automated Test Generation

Since writing tests manually can be tedious and expensive, a huge body of research has been focused on automated test generation. Various techniques have been investigated for automated test generation, including random generation [38], specification-based generation [3, 27], symbolic execution [32, 28, 4, 65], search-based generation [12], and so on. However, even state-of-the-art test generation techniques still suffer from practical issues. Symbolic execution techniques are based on the expensive path constraint solving, and are not applicable to certain programs with hard-to-solve constraints and not scalable to large programs. In addition, symbolic execution techniques also have troubles in environment modeling as well as object creation [55]. Meanwhile, other techniques (e.g., random and search-based test generation) struggle in covering all code elements of the programs under test due to the unawareness of actual path constraints. In fact, a recent study by Fraser et al. [15] shows that there is not clear evidence that search-based test generation techniques can help developers detect more program faults. Furthermore, another study by Wang et al. [54] demonstrates that symbolic execution is relatively less effective than manually-written tests on covering hard-to-cover code and detecting faults. Also, the study shows that symbolic execution is less effective than manually-written tests on generating valid stucturally complex inputs and exploring meaningful program behaviors. In contrast with existing efforts on test generation, our *ISON* work opens a new dimension for automated testing – *ISON* avoids the limitations of traditional automated test generation techniques by directly utilizing and extending the existing manually written tests.

6.2 Regression Testing

Regression testing aims to test the new program version more efficiently and effectively. Traditional work in regression testing, including test-suite reduction [51, 20, 18, 63], test-case prioritization [57, 44, 45, 35, 46, 19], and test selection [43, 21, 37, 31], mainly manipulates existing regression tests to find faults faster. For example, a typical test-case prioritization technique [62, 57, 44, 45] reorders the tests to execute the effective tests earlier in order to speed up fault detection. Different from these traditional work, *ISON* aims to improve the effectiveness of the existing tests rather than run them more efficiently. More recent works on regression test augmentation [56, 47, 40] are closely related to *ISON* since they also aims to improve the effectiveness of the regression test suite. Existing test augmentation techniques usually use off-the-shelf test generation techniques, e.g., symbolic execution [56, 39, 47, 40] and search-based test generation [56], to augment the existing test suite. Therefore, regression test augmentation suffers from the same issues with the automated test generation techniques. In contrast, our *ISON* work does not require any additional test generation, and can directly utilize the existing tests to improve code coverage and fault-detection effectiveness.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed **isomorphic regression testing**, which explores the behaviors of the program under test by creating its variants (i.e., modified programs), instead of generating tests. We also proposed the implementation approach *ISON*, and evaluated *ISON* on 10 projects. The results show that *ISON* is able to execute many uncovered branches. We also compared *ISON* with *EvoSuite*. The results show that *ISON* detects a number of faults that *EvoSuite* fails to detect. In future, we will evaluate *ISON* on more and larger projects, as well as on multiple faults (e.g., high-order mutants [24]) and real faults, and make comparison with more automated test augmentation approaches

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. This work is partially supported by the National Basic Research Program of China (973) under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61421091, 91318301, 61225007, 61529201, 61522201, 61272157. This work is also partially supported by NSF Grant No. CCF-1566589 and Google Faculty Research Award.

9. REFERENCES

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering (TSE)*, 36(6):742–762, 2010.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 123–133, 2002.

[4] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[6] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. A text-vector based approach to test case prioritization. In *Proc. ICST*, pages 266–277, 2016.

[7] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. An empirical comparison of compiler testing techniques. In *Proc. ICSE*, pages 180–190, 2016.

[8] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology, 1998.

[9] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.

[10] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *Proc. ICST*, pages 1–10, 2015.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[12] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. FSE*, pages 416–419, 2011.

[13] G. Fraser, A. Arcuri, and P. McMinn. Test suite generation with memetic algorithms. In *Proc. GECCO*, pages 1437–1444, 2013.

[14] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2014.

[15] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol*, 2014.

[16] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2):278–292, 2012.

[17] D. Hao, L. Zhang, M. Liu, H. Li, and J. Sun. Test-data generation guided by static defect detection. *Journal of Computer Science and Technology (JCST)*, 24(2):284–293, 2009.

[18] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proc. ICSE*, pages 738–748, 2012.

[19] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test-case prioritization approach. *ACM Transactions on Software Engineering and Methodology(TOSEM)*, 24(2), 2014.

[20] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[21] 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 *ACM SIGPLAN Notices*, volume 36, pages 312–326. ACM, 2001.

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proc. ICSE*, pages 191–200, 1994.

[23] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. ICSE*, pages 435–445. ACM, 2014.

[24] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Proc. SCAM*, pages 249–258. IEEE, 2008.

[25] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, 2011.

[26] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing. In *Proc. FSE*, 2014.

[27] S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. *Proc. ASE*, 11(4):403–434, 2004.

[28] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. 2003.

[29] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[30] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for c programs using cute and austin. *Journal of Systems and Software*, 83(12):2379–2391, 2010.

[31] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *Proc. FSE*, page to apper, 2016.

[32] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. *ACM SIGPLAN Notices*, 48(10):19–32, 2013.

[33] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang. Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In *Proc. ASE*, pages 494–505. IEEE/ACM, 2015.

[34] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel.

Dodona: Automated oracle data set selection. In *Proc. ISSTA*, pages 193–203. ACM, 2014.

[35] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *Proc. ICSE*, pages 535–546, 2016.

[36] P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[37] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 241–251. ACM, 2004.

[38] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[39] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.

[40] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proc. PLDI*, pages 504–515, 2011.

[41] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid. Compositional symbolic execution with memoized replay. In *Proc. ICSE*, volume 1, pages 632–642, 2015.

[42] M. Renieris, S. Chan-Tin, and S. P. Reiss. Elided conditionals. In *Proc. PASTE*, pages 52–57. ACM, 2004.

[43] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[44] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering (TSE)*, 27(10):929–948, 2001.

[45] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. pages 268–279, 2015.

[46] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proc. ICSE*, pages 268–279, 2015.

[47] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. ASE*, pages 218–227, 2008.

[48] K. Sen, D. Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*, volume 30. 2005.

[49] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Proc. ASE*, pages 201–211. IEEE, 2015.

[50] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.

[51] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *Proc. FSE*, pages 246–256. ACM, 2014.

[52] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proc. ICST*, pages 200–209. IEEE, 2011.

[53] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[54] X. Wang, L. Zhang, and P. Tanofsky. Experience report: how is dynamic symbolic execution different from manual testing? a study on klee. In *Proc. ISSTA*, pages 199–210. ACM, 2015.

[55] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux. Precise identification of problems for structural test generation. In *Proc. ICSE*, pages 611–620. ACM, 2011.

[56] 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. ACM, 2010.

[57] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proc. ISSTA*, pages 140–150, 2007.

[58] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations. In *Proc. ASE*, pages 701–712. ACM, 2014.

[59] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei. A survey on bug-report analysis. *Science China, Information Sciences*, 58(2):1–24, 2015.

[60] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang. Predictive mutation testing. In *Proc. ISSTA*, pages 342–353. ACM, 2016.

[61] J. Zhang, M. Zhu, D. Hao, and L. Zhang. An empirical study on the scalability of selective mutation testing. In *Proc. ISSRE*, pages 277–287. IEEE, 2014.

[62] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proc. ICSE*, pages 192–201. IEEE, 2013.

[63] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *Proc. ISSRE*, pages 170–179, 2011.

[64] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. ISSTA*, pages 331–341, 2012.

[65] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. d. Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, pages 1–10, 2010.

[66] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proc. ISSTA*, pages 362–372, 2014.

[67] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proc. ICSE*, pages 272–281. ACM, 2006.