Concolic Metamorphic Debugging

Hao Jin, Yanyan Jiang, Na Liu, Chang Xu*, Xiaoxing Ma* and Jian Lu
State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
Dept. of Computer Science and Technology, Nanjing University, Nanjing, China
frankjinhao@gmail.com, jiangyy@outlook.com, liunana1993@gmail.com, {changxu, xxm, lj}@nju.edu.cn

***Abstract*—Debugging is challenging and labor-intensive. Debugging programs with weak or no oracle is even more difficult due to lack of passing and failing test runs as well as their comparisons. To address these challenges, we exploit metamorphic relations to construct new programs that are enhanced with synthesized oracle, and combine concolic testing and branch-switching debugging to localize potentially faulty places in original programs. We name our approach concolic metamorphic debugging (or Comedy for short). We experimentally evaluated Comedy with real-world Java programs. The experimental results reported that Comedy successfully generated debugging report for 88.4% of 2,330 faulty programs. The average branch distance between the reported locations and the real fault places is only 1.68. Besides, 36% of the debugging reports precisely locate the fault.**

***Index Terms*—Metamorphic relation, Debugging, Fault localization**

I. Introduction

Testing and debugging are crucial for assuring the quality of computer software [1]. Testing decides whether software is faulty, whereas debugging comes afterwards to localize faulty places in software.

Unfortunately, debugging remains to be challenging and labor-intensive [2]. Automated debugging aids are desired during software development, but existing research efforts still do not work out in many application domains, despite their continuous years of progress. Existing debugging aids mostly depend on the availability of abundant passing and failing program executions. They use various sophisticated techniques to analyze and compare these program executions or traces [3]–[8], or calculate and accumulate statistics for certain entities selected from program traces [9], [10]. Based on such trace comparisons or entity rankings, these techniques can report potentially faulty places in programs. However, since these techniques use passing and failing program executions, they rely heavily on *test oracles* which decide the program pass or failure. For programs that have only weak oracles or even no oracle [11], e.g., scientific computing or algorithmic programs, their testing is non-trivial, not to mention their debugging practice.

In this paper, we present an automated fault isolation technique on top of metamorphic testing [12] to address the challenges in debugging programs that have only weak oracles or even no oracle. We name our technique $\underline{\text{Co}}$ncolic $\underline{\text{Me}}$tamorphic $\underline{\text{De}}$bugging (CoMeDy, or Comedy for short) as it integrates both metamorphic testing and concolic testing [13]–[15] with branch-switching debugging [5], [8]. Given a program, Comedy requires only one *failure-inducing test input* (either user-specified or randomly generated) that violates this program's metamorphic relation, and it can then automatically report a critical branch statement relating to the responsible fault in the program.

*Corresponding authors

Based on the failure-inducing test input, Comedy conducts systematic path exploration using concolic testing on a synthesized program which encodes the original program as well as the metamorphic relation check. The path exploration is an iterative process that searches for a pair of test inputs such that one violates the metamorphic relation while the other passes. In addition, this pair of test inputs share the longest common control-flow prefix in their executions. Comedy would report the first diverging branch in their executions as the *critical branch*. This report contains useful information for developers to understand and pinpoint the root cause of the metamorphic relation violation (or failure).

The underlying principle of Comedy is that its path exploration maximizes the common control-flow prefix, and as we explain later, this is equivalent to finding an execution trace that violates the metamorphic relation and having its suffix minimized, such that one can change this suffix by feeding a different test input to obtain a new execution satisfying the metamorphic relation. In other words, by altering the execution trace after the critical branch, one can isolate a minimal amount of code to make the failure (metamorphic relation violation) vanish. Therefore, the reported critical branch can be useful for effective debugging. Comedy is a general approach to debugging any program with metamorphic relation, and we implemented Comedy by extending Java Pathfinder [16] and Symbolic Pathfinder [17] to debug such Java programs.

We experimentally evaluated Comedy on a set of algorithmic Java programs. For a total of 2,330 mutated programs that are faulty in terms of metamorphic relation violation, Comedy successfully generated debugging reports for 88.4% of them. By measuring the branch distance on execution traces (i.e., amount of distinct branches between the reported branch and the real fault), Comedy's debugging reports are only 1.68 branches away from the places of real faults on average. Besides, 36% of the distances are 0, indicating that Comedy has precisely localized these faults (fault statements are either in the reported critical branches, or on their subsequent assignments). We also conducted a case study using server program Tomcat 6.0 with a seeded fault (a common programming mis-

```
 1 class MAXSUB {
 2   static int maxsub(int[] A, int n) {
 3     int ans = 0, cnt = 0;
 4     for (int i = 0; i < n; i ++) {
 5       cnt += A[i];
 6       if (cnt < 0)
 7         ; // should be cnt = 0;
 8       if (cnt > ans)
 9         ans = cnt;
10     }
11     return ans;
12   }
13 }
```

Figure 1: Implementation of the Kadane's algorithm solving the MAXSUB problem (Line 7 contains a missing-statement fault)

take like coding typo). Comedy precisely localized this fault, which suggests Comedy's potential usefulness in practice.

The rest of this paper is organized as follows. Section II presents background knowledge and uses a motivating example to illustrate Comedy's debugging process. Section III elaborates on our Comedy approach. Section IV introduces Comedy's implementation details. Sections V evaluate Comedy with a set of real-world Java programs. Section VI discusses related work, followed by the conclusions in Section VII.

II. Overview

A. Metamorphic Testing

Test oracle decides whether a test run passes in software testing practice. However, programs in some application domains lack oracles or own only weak oracles, and hence pose challenges to normal testing practice. Metamorphic testing [12], [18], [19] addresses this problem by reducing a program's failure to violation of this program's certain metamorphic relation. Such metamorphic relations enforce necessary conditions that must hold about inputs and outputs of the program in its several executions, rather than only one execution as in traditional program testing.

Metamorphic testing works in three steps: (1) Prepare an initial test input I for a program under test, and execute this program to obtain its output O; (2) Generate a follow-up test input I' from I and O (O is optional) based on one of this program's metamorphic relations, and execute the program to obtain its output O'; (3) Check whether (I, I', O, O') satisfies the selected metamorphic relation, and the violation indicates that the program contains a fault somewhere. This is the simplest usage of metamorphic relation (concerning two executions only). In theory, a metamorphic relation can concern arbitrarily many program executions. For ease of presentation, we assume that a metamorphic relation concerns only two program executions, but we note that our Comedy approach can apply to programs with metamorphic relations concerning more program executions.

B. Comedy Overview

To elaborate how Comedy debugs a program with metamorphic relation, consider the MAXSUB problem finding the contiguous subarray which has the largest sum (note that for an array whose elements are all negative, MAXSUB returns 0, indicating the maximum contiguous subarray is an empty array). Formally, the MAXSUB problem is to find

$$f(A) = \max_{0 \le i \le j \le n} \left\{ \sum_{i \le k < j} a_k \right\}$$

for $A = (a_0, a_1, \ldots, a_{n-1})$. Figure 1 demonstrates a Kadane's algorithm [20] implementation P to solve this problem, and P unfortunately contains a missing-statement fault at Line 7. P is difficult to test because it owns almost no oracle as one can hardly tell the expected output for a given integer array. Nevertheless, P has a straightforward metamorphic relation $f(a_0, \ldots, a_{n-1}) = f(a_{n-1}, a_{n-2}, \ldots, a_0)$. This metamorphic relation suggests that the sum of the largest contiguous subarray must equal to that of its reversed counterpart, which certainly holds for the MAXSUB problem. Assume that we find a metamorphic relation violation by testing the program with input $A = (4, -2, 1)$.

To conduct debugging, Comedy creates a new program P' that encodes the entire metamorphic testing process by user-provided code snippets. For any given test input I (an integer array), P' internally calls P with I to obtain P's output O. P' then generates a follow-up test input I' by reversing the elements in I according to the metamorphic relation, and internally calls P again with I', obtaining O'. Next, P' moves to the assertion checking part by inspecting whether O' equals to O, and finally returns this checking result (i.e., PASS or FAIL). An output of FAIL indicates a failure of P'.

In our example, the execution of P' internally calls P twice with $I = (4, -2, 1)$ and follow-up test input $I' = (1, -2, 4)$, obtaining $O = 4$ and $O' = 3$, respectively. Obviously $O \neq O'$, indicating the failure of the test. Comedy then debugs P with the failure-inducing input I by performing concolic execution [14] on P'.

Along with the concrete execution, Comedy also collects the symbolic representation of every variable and every conditional statement evaluation. For example, in the first round of the loop (`i = 0`), the symbolic representation of `cnt` is a_0 and `(cnt < 0)` is evaluated to be false, yielding a branch condition of $\neg(a_0 < 0)$. The conjunction of all branch conditions in the execution is the path condition PC such that any input satisfying PC will follow exactly the same path as executing P' with $I = (4, -2, 1)$. Remind that the execution of P' contains two pieces of P's execution (one with input I followed by the other with input I'), and therefore PC is the conjunction of path conditions of two pieces of executions[1].

[1]The complete path condition for our example is $PC = PC_I \wedge PC_{I'} = (\neg(a_0 < 0) \wedge a_0 > 0 \wedge \neg(a_0 + a_1 < 0) \wedge \neg(a_0 + a_1 > a_0) \wedge \neg(a_0 + a_1 + a_2 < 0) \wedge \neg(a_0 + a_1 + a_2 > a_0)) \wedge (\neg(a_2 < 0) \wedge a_2 > 0 \wedge a_1 + a_2 < 0 \wedge \neg(a_1 + a_2 > a_2) \wedge \neg(a_0 + a_1 + a_2 < 0) \wedge a_0 + a_1 + a_2 > a_2)$.

Starting from the path condition PC, Comedy systematically explores execution paths of P' by iteratively negating the last unprocessed branch condition in PC, and solves this newly generated path condition to obtain alternative inputs of P'. Such exploration continues until a passing execution is found. In other words, Comedy conducts an exploration of all possible program paths in a depth-first order, searching for the first one that can pass the metamorphic relation check. The depth-first ordering isolates a minimum amount of code to obtain a passing test input, and hence the point of isolation is expected to be useful for localizing the fault.

In this example, Comedy first switches the last branch statement in Line 8 at the computation of O' with $I' = (1, -2, 4)$ (whose symbolic representation is (a_2, a_1, a_0)). At this time, we have `i` $= 2$, `ans` $= a_2$ and `cnt` $= a_2 + a_1 + a_0$, and the branch condition for `if (cnt > ans)` taking the if-then branch in Line 8 is therefore $a_2 + a_1 + a_0 > a_2$. To obtain an alternative execution, Comedy replaces the selected branch's condition in PC with its negation (i.e., $\neg(a_2 + a_1 + a_0 > a_2)$, which is equivalent to $a_0 + a_1 \le 0$), and removes all subsequent branch conditions. Solving this new path condition yields an alternative input $(2, -2, 1)$, which still cannot pass the metamorphic relation check, and the iteration proceeds with this newly generated input. After two iterations (in these two iterations, the newly generated path conditions are both unsatisfiable), Comedy switches the conditional statement in Line 6 where `i` $= 1$ and `cnt` $= a_2 + a_1$ at the computation of O'. This time, switching the branch condition $(a_1 + a_2 < 0)$ yields an alternative input $(1, -1, 2)$, obtaining $O = O' = 2$, which is the first time of obtaining a passing run.

The last-time switched branch's corresponding statement, `if (cnt < 0)` in Line 6, is therefore selected by Comedy as the critical branch for debugging report. This branch is critical in the sense that by taking a different branch of this conditional statement, we can make a failing run pass. The input $(1, -1, 2)$ passes the metamorphic relation check because `cnt` never falls below zero both in the original and in the reversed array, and hence the missing-statement fault in Line 7 is never triggered. In this motivating example, Comedy successfully isolates the root cause of the fault to obtain a passing run, and precisely localizes the fault.

III. The Comedy Approach

A. Preliminaries

1) Program and Execution: Without loss of generality, we assume that a *program* consists of only assignment and if-then-else branch statements (we focus on single-threaded and deterministic programs in this paper). Given a program P and its input I, executing P yields a concrete *execution* π and π's output $P(I)$. We define execution π as a sequence of alternating assignment and branch statements manifested in the execution of P. Formally, we define

$$\pi = \alpha_1, c_1, \alpha_2, c_2, \ldots, \alpha_n, c_n, \alpha_{n+1},$$

where each α_i $(1 \le i \le n+1)$ represents zero or more assignment statements, and c_i represents the i-th branch statement executed in π. For each c_i, we use evaluation result function $V^\pi(c_i) \in \{\text{True}, \text{False}\}$ to refer to $c_i's$ evaluation result (i.e., a truth value) in π. $V^\pi(c_i) = \text{True}$ means that c_i's if-then branch is taken in π, as $V^\pi(c_i) = \text{False}$ indicates that c_i's if-else branch is taken. For convenience, we also use $S^\pi(c_i)$ to refer to c_i's corresponding source-code statement in program P.

2) Branch Condition and Path Condition: In any execution π, a branch statement c_i's evaluation result $V^\pi(c_i)$ depends only on its runtime program state, which is determined by assignment statements $\alpha_1, \alpha_2, \ldots, \alpha_i$. Values in these assignment statements ultimately depend on program P's input I. We use concolic testing [13]–[15] to collect symbolic representation of each program variable, and capture *branch condition* predicate BC_i^π for each branch statement c_i along the concrete execution of P. Intuitively, a branch condition is a constraint on program input that forces evaluation of a specific branch to be a specific truth value. Formally, for any P's input I', $BC_i^\pi(I') = \text{True}$ if and only if executing $\alpha_1, \alpha_2, \ldots, \alpha_i$ with input I' yields c_i being evaluated $V^\pi(c_i)$.

For any prefix of π, we obtain its *path condition* by conjoining each branch's branch condition. Inputs satisfying a specific path condition of π yield execution traces that share a common control-flow prefix with execution π. Formally, we define *path condition* PC_k^π for execution π's k-length prefix $\alpha_1, c_1, \alpha_2, c_2, \ldots, \alpha_k, c_k$ $(0 \le k \le n)$ as

$$PC_k^\pi = BC_1^\pi \wedge BC_2^\pi \wedge \ldots \wedge BC_k^\pi.$$

Executing P with any input satisfying PC_k^π must yield such an alternative execution $\pi' = \alpha_1', c_1', \alpha_2', c_2', \ldots, \alpha_m', c_m', \alpha_{m+1}'$ that π' follows exactly the same control flow path as in execution π for their first k branch statements, i.e., $S^\pi(c_j) = S^{\pi'}(c_j')$ and $V^\pi(c_j) = V^{\pi'}(c_j')$ for all $j \in [1, k]$.

3) Metamorphic Relation: For brevity, we restrict our formalization to binary metamorphic relations [21]. A binary metamorphic relation only involves two pieces of the original program's execution. Nevertheless, one can easily generalize our fault localization technique to metamorphic relations that involve more executions. Formally, we denote a *metamorphic relation* MR by a tuple $MR = \langle T, M \rangle$. Transform function T defines how a follow-up input I' should be generated according to a given initial input I with respect to this metamorphic relation, i.e., $I' = T(I)$ (in metamorphic testing, I' can be generated according to both I and O, but for simplicity, in this paper we assume that I' is generated only from I). Metamorphic property M specifies a condition about a program's outputs (from its two executions) that must hold if a follow-up input is generated according to T and fed to this program, i.e., if $O = P(I)$ and $O' = P(I')$, and we assert that $M(O, O')$ must hold. Our technique does not pose any restriction on T and M, as long as they can be represented by arbitrary program snippets.

B. *Comedy in a Nutshell*

Comedy is a debugging aid technique (i.e., providing useful information for reasoning the root cause of a failure).

Algorithm 1: Synthesized P' from P and $MR = \langle T, M \rangle$

```
Input: $I$
Output: Result of metamorphic relation check
$O_1 \leftarrow P(I)$;
$I_1 \leftarrow T(I)$;
$O_2 \leftarrow P(I_1)$;
if $M(O_1, O_2)$ then
    return PASS;
else
    return FAIL;
```

Algorithm 2: Finding (I_f, I_p) pair

```
Input: $P'$, $MR = \langle T, M \rangle$ and $I_{init}$
Output: A pair of $(I_f, I_p)$
$I \leftarrow I_{init}$;
$\pi \leftarrow \mathsf{ConcolicExecute}(P', I) = \alpha_1, c_1, \ldots, \alpha_n, c_n, \alpha_{n+1}$;
$R \leftarrow \{\}$; $d \leftarrow n$;
while $d > 0$ do
    if $d \notin R$ then
        $R \leftarrow R \cup \{d\}$;
        $PC \leftarrow BC_1^\pi \land BC_2^\pi \land \ldots \land BC_{d-1}^\pi \land \neg BC_d^\pi$;
        $I' \leftarrow \mathsf{SMTSolve}(PC)$;
        if $PC$ is satisfiable then
            $\pi' \leftarrow \mathsf{ConcolicExecute}(P', I') = \alpha'_1, c'_1, \ldots, \alpha'_m, c'_m, \alpha'_{m+1}$;
            if $\pi'$ passed then
                return $(I, I')$;
            $d \leftarrow m$; $I \leftarrow I'$; $\pi \leftarrow \pi'$;
    else
        $R \leftarrow R \setminus \{d\}$; $d \leftarrow d - 1$;
return FAIL;
```

Therefore, we assume that the program under test and the testing results are available at debugging time[2]. Specifically, the core algorithm of Comedy takes the following inputs: 1) a program P and P's metamorphic relation $MR = \langle T, M \rangle$ (both T and M are code snippets); 2) a failure-inducing test input I_{init} which violates MR. Comedy algorithm reports a branch statement in P which is evidently related to the program fault, helping developers narrow down the scope of finding the failure's root cause.

Starting with P, MR and I_{init}, Comedy algorithm roughly works as follows. Comedy first synthesizes a new program P' that concatenates two pieces of P's executions as well as the MR checking code, as illustrated in Algorithm 1.

Then, Comedy conducts a predicate switching [5] local search starting from I_{init}, searching for a pair of alternative inputs (I_f, I_p) that satisfies:

1) Executing P' with I_f yields a metamorphic relation violation, while executing with I_p can pass the metamorphic relation check.
2) Execution traces of executing P' with I_f and I_p are as similar as possible. Since aligning two execution traces are extremely difficult [24], our "similar" means that execution traces of I_f and I_p share a common prefix that is as long as possible.

If a pair of (I_f, I_p) is found, Comedy extracts the first diverging branch of I_f and I_p to be the critical branch, and reports this branch statement to the developer for further examination. The critical branch is "critical" in the sense that taking its if-then branch yields passing (failing) execution while taking its if-else branch yields a contrary result. The ability of hindering or exposing the failure suggests that the critical branch's related statements are possibly associated with the root cause of the bug.

In the case that multiple failure-inducing inputs are available (e.g., various failure-inducing inputs may exist for different metamorphic relations of a single program), and thus a series of different (I_f, I_p) pairs can be obtained, we choose the pair whose first diverging branch isolates the minimum amount of code to be our final reported critical branch. Formalized algorithms and detailed discussions are presented in the following subsections.

[2]There are plenty of work describing how to find failure-inducing inputs for metamorphic testing [18], [21]–[23]. Therefore we do not focus on this procedure.

C. Generating (I_f, I_p) from I_{init}

There exists many approaches that can find a passing input I_p and a failure-inducing input I_f separately [7], [8], [25]. However, to make execution traces of (I_f, I_p) share a maximal prefix length requires a systematic search strategy. We adopt the similar idea in predicate switching algorithm [5], iteratively exploring different program paths until our desired (I_f, I_p) pair is found.

Each iteration in Comedy begins with concretely executing P' with failure-inducing input I, yielding an execution trace:

$$\pi = \alpha_1, c_1, \alpha_2, c_2, \ldots, \alpha_k, c_k, \alpha_{k+1}, \ldots, \alpha_n, c_n, \alpha_{n+1}.$$

During the concrete execution, Comedy also collects branch conditions BC_i^π (note that the branch condition of MR checking is not added into PC). Then Comedy tries to switch the last branch of π to obtain a different execution trace that has not yet been explored before, by negating $V^\pi(c_k)$ for some k, at the same time keeping $V^\pi(c_i)$ unchanged for $i < k$. Suppose that Comedy is switching c_k in π for the current iteration, the branch switching operation is equivalent to solving another input I' satisfying the path constraint:

$$PC_{k-1}^\pi \land \neg BC_k^\pi = BC_1^\pi \land BC_2^\pi \land \ldots \land BC_{k-1}^\pi \land \neg BC_k^\pi.$$

If $\left(PC_{k-1}^\pi \land \neg BC_k^\pi\right)$ is unsatisfiable, we immediately start a new round of iteration. Otherwise, the constraint solver returns an alternative input I' leading to an alternative execution

$$\pi' = \alpha'_1, c'_1, \alpha'_2, c'_2, \ldots, \alpha'_k, c'_k, \alpha'_{k+1}, \ldots, \alpha'_m, c'_m, \alpha'_{m+1}$$

such that π' shares a k-length common control flow prefix with π (i.e., $\left(S^{\pi}(c_i) = S^{\pi'}(c'_i) \wedge V^{\pi}(c_i) = V^{\pi'}(c'_i)\right)$ holds for all $i \in [1, k-1]$), and the k-th branch's truth value is distinct in π and π' (i.e., $S^{\pi}(c_k) = S^{\pi'}(c'_k)$ but $V^{\pi}(c_k) \neq V^{\pi'}(c'_k)$). Since we are always switching the last branch in π, the common prefix length k is kept maximal.

If π' passes the metamorphic relation check, we immediately report (I, I') as our desired (I_f, I_p). Otherwise, π' must also be a failure-inducing trace that has not been explored before. Therefore, we naturally start a new iteration with I'. Iteration proceeds until (I_f, I_p) is returned, or all possible paths are tried. In the secondary (extreme) case that Comedy fails to find a passing execution, we conclude that the fault possibly exists in code that is inevitably executed in any path.

The systematic path exploration is realized by a depth-first concolic exploration on P' starting from I_{init}, as shown in Algorithm 2. In each iteration, Comedy negates the last unexplored branch of the current execution (Line 6) to obtain a new path condition (Line 7), and then conducts constraint solving (Line 8). If the path condition is satisfiable, Comedy checks whether this new execution passes now (Line 10–11). If not, Comedy proceeds to the next iteration (Line 15). The iteration terminates until the desired (I_f, I_p) is found, or all possible execution traces have been tried (Line 16).

D. Fault Localization from (I_f, I_p)

For a pair of inputs (I_f, I_p) and their execution traces $\pi_f = \alpha_1, c_1, \alpha_2, c_2, \ldots, \alpha_k, c_k, \alpha_{k+1}, \ldots, c_n, \alpha_{n+1}$ and $\pi_p = \alpha'_1, c'_1, \alpha'_2, c'_2, \ldots, \alpha'_k, c'_k, \alpha'_{k+1}, \ldots c'_m, \alpha'_{m+1}$ mentioned in the previous subsection, we naturally report c_k and its corresponding branch statement $S^{\pi_p}(c_k) = S^{\pi_f}(c_k)$ to be related to the root cause of the failure. The first diverging branch $V^{\pi_p}(c_k) \neq V^{\pi_f}(c_k)$ yields passing and failing executions, and this exactly explains why c_k is "critical". In other words, we have isolated the π_f's suffix $c_k, \alpha_{k+1}, \ldots, c_n, \alpha_{n+1}$ such that, if we change the evaluation result of c_k, the isolated parts are displaced and metamorphic testing passes.

Also note that we have made the common control-flow prefix length k in π_f and π_p as large as possible because we are always negating the last branch condition to obtain an alternative input. Larger k yields shorter remaining execution in π_f (i.e., smaller $n - k$), meaning that the isolated parts are kept minimal, providing a better chance of exposing the root cause of the failure.

However, fault localization is notoriously difficult and the previous reasoning still cannot provide a precise fault localization guarantee. For example, metamorphic relation may not reveal the fault even if the program has indeed manifested an erroneous state [26]. That is, though I_p passes the check of metamorphic relation, there might still contain errors in π_p. In this situation, reporting c_k and $S^{\pi_p}(c_k)$ to be the critical branch would no longer be accurate because π_p does not isolate the fault. Another unsoundness possibility comes from our depth-first exploration strategy. For scalability considerations, rather than exploring the entire input space like semi-proving [27], Comedy stops iteration immediately when a first pair of

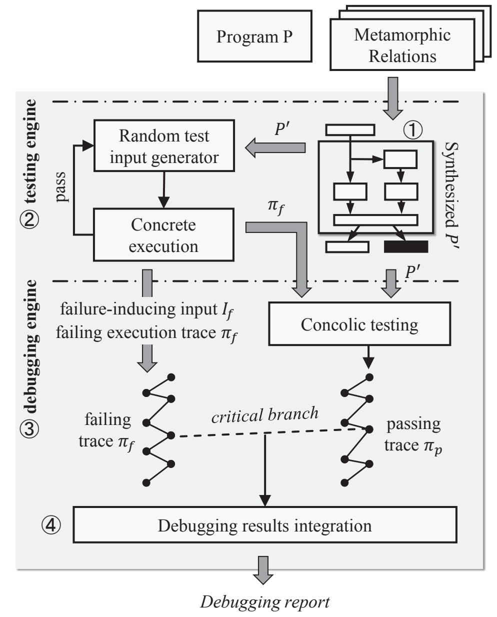


Figure 2: Comedy Tool Architecture

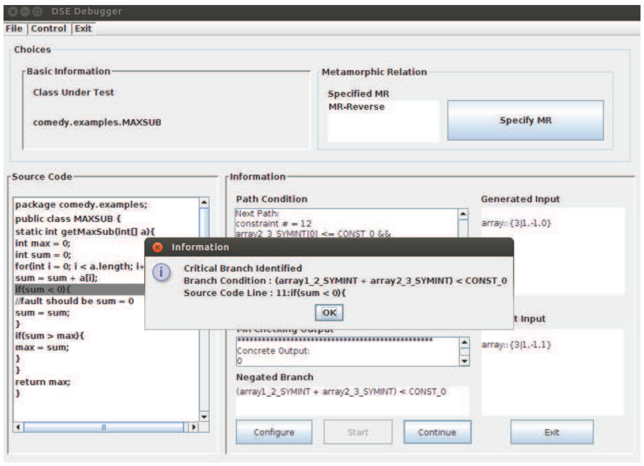
(I_f, I_p) is found, and the constraint solving procedure itself is also not completely sound. Therefore, our strategy does not necessarily produce the global failure-inducing input that has the shortest suffix, which is more likely to related to the root cause of the failure.

Nevertheless, the critical branch c_k reported by Comedy is still always related to the root cause, depending on its ability of revealing and hiding the observable fault. Also, for a specific program under test, we usually have multiple metamorphic relations. Moreover, for each metamorphic relation, a series of different (I_f, I_p) pairs are usually available. Following the intuition that we should always isolate the minimum amount of code, we naturally choose the critical branch whose failure execution π_f has a least suffix (i.e., smallest $n - k$) to be our finally reported critical branch.

IV. Implementation

We implemented our fault localization tool as a full-package solution as illustrated in Figure 2.

Overall Architecture. Figure 2 shows that Comedy first synthesizes a P' from program P for each specified metamorphic relation as is marked by ①. Synthesized P' can either be fed to Comedy's testing engine which is marked by ②, or be directly used for debugging with user-specified failure-inducing test inputs. ③ is Comedy's debugging engine which uses concolic testing to generate a pair of failing and passing executions that share a minimal common suffix. Once a passing run is obtained, the last switched branch is identified as the critical branch. Since a program may have multiple metamorphic relations and one may have multiple failure-inducing inputs for a single metamorphic relation, Comedy finally compares their isolated suffix lengths to generate a final debugging report by ④.



```
package comedy.examples;
public class MAXSUB {
static int getMaxSub(int[] a){
int max = 0;
int sum = 0;
for(int i = 0; i < a.length; i+
sum = sum + a[i];
if(sum < 0){
//fault should be sum = 0
sum = sum;
}
if(sum > max){
max = sum;
}
}
return max;
}
```

```
Next Path:
constraint # = 12
array2_3_SYMINT[0] <= CONST_0 &&
```

```
array:: {3|1,-1,0}
```

```
Critical Branch Identified
Branch Condition : (array1_2_SYMINT + array2_3_SYMINT) < CONST_0
Source Code Line : 11:if(sum < 0){
```

```
**************************************************
Concrete Output:
0
```

```
array:: {3|1,-1,1}
```

```
(array1_2_SYMINT + array2_3_SYMINT) < CONST_0
```

Figure 3: Comedy GUI

Predicate-switching debugging. Our fault localization is based on concolic testing. We extended Symbolic Pathfinder [17] to support concolic testing. We enhanced Java Pathfinder [16]'s bytecode instruction interpreter such that each variable's symbolic representation is computed and each branch instruction's branch condition is accumulated along a concrete program execution. As each branch condition is collected, we can switch any branch to obtain a new test input by invoking a constraint solver. In addition, we also extended Symbolic Pathfinder to support more input data structures and types for our concolic execution purpose.

Coping with constraint solver. Comedy relies on Choco [28] and CVC3 [29] which are included in Symbolic Pathfinder [17] for constraint solving. However, constraint solving faces significant capability and scalability issues. We thus adopted the similar technique used in DART [14] that repeatedly replace unresolvable symbolic variables with their corresponding concrete values, until constraint solving can terminate within a predefined time limit. This treatment might report a satisfiable constraint as unsatisfiable, yielding sub-optimal solutions (recall that Comedy tries to minimize the execution trace after the critical branches). Nevertheless, our evaluation results in Section V show that Comedy can still report informative critical branch for fault localization with our concolic treatment.

Putting them together. We also implemented a GUI tool for testing and debugging Java programs with metamorphic relations. Our Comedy implementation provides an almost fully automated process of debugging metamorphic relation violations. Figure 3 demonstrates how to use the Comedy GUI to debug our MAXSUB example.

V. Evaluation

Our evaluation of Comedy consists of several parts. First, we adopt techniques in mutation testing that seed mutants to our subject programs to effectively simulate real faults [30]. Then, for each mutated program, we randomly test it with metamorphic relations[3] to find failure-inducing inputs. For each failure-inducing input, Comedy produces a debugging report. We evaluate Comedy by investigating how much the debugging report can narrow down the scope for searching the fault, and how much is the cost of invoking Comedy.

A. Experimental Subjects

We selected 21 correct Java programs as our subject programs, as shown in Table I. All of them are either earlier studied subjects on metamorphic testing or algorithmic programs. Note that few of them have oracles (e.g., a sorting algorithm) but most are not. Besides, although these programs have small or medium sizes, their fault localization is extremely difficult due to the lack of oracles. (Column 4 of Table I).

We then created mutants by seeding random program changes to simulate faults made by developers using MuJava [35]. We also enhanced MuJava to inject missing-statement faults, which are typically considered as difficult for debugging [36], [37]. After filtering out unsuitable mutants (e.g., crashes or compilation failures), we obtained a total amount of 3,594 mutated programs for mutation testing (Column 5 of Table I). For each mutated (potentially faulty) program, we ran random testing for each metamorphic relation to obtain 10 failure-inducing test inputs for fault localization using Comedy. We discarded a mutated program when 100,000 randomly generated inputs still cannot fail it, either because it is an equivalent mutant, or our metamorphic relations are too weak to reveal the existence of the fault. After these two levels of filtering, we finally obtained 2,330 mutated programs that can be proved faulty by at least one metamorphic relation[4] as our fault subjects to evaluate Comedy's fault localization ability (Column 6 of Table I).

B. Experimental Setup

We study the effectiveness and efficiency of Comedy using the following methodology:

1) According to the discussions in Sections III-C and IV, Comedy does not necessarily report a critical branch statement. This is either because our concolic testing implementation is unsound, or the fault is on an inevitable path. Therefore, the success rate of Comedy poses the first question.
2) If Comedy reports a critical branch, we must evaluate how much informative it is. Therefore we evaluate Comedy's effectiveness by quantitatively measuring how close is a reported branch to its corresponding real fault.
3) Since Comedy uses concolic testing, we also evaluate its time cost on subject programs.

[3]Identification of metamorphic relation is studied in [19], [31]–[34], and therefore we adopt these standard approaches to construct metamorphic relations for our experimental subjects..

[4]We focus on evaluating Comedy's fault localization capability,and therefore care only mutated programs that can be proven faulty by a metamorphic relation. Effectiveness of metamorphic testing is a separate issue and is studied in [19], [31].

Table I: Subject Programs

Subject Name	*Source (Description)*	*LOC*	*Metamorphic Relation*	*# Mutants*	*# Killed*	*% Killed*
ClosestPair	Princeton Algorithm	370	Transpose	281	160	57%
Dijkstra	Shortest Path Algorithm	271	Transpose, Shift	214	161	75%
Edmonds-Karp	Maxium Flow Algorithm	229	Transpose, Shift	46	46	100%
MaximumRectangle	LeetCode OJ Problem	113	Transpose	172	156	91%
BoyerMoore	LeetCode OJ Problem	93	Reverse	71	59	83%
SurroundedRegion	LeetCode OJ Problem	78	Transpose	309	309	100%
DecodingWays	LeetCode OJ Problem	78	Increment	409	400	98%
LargestRectangle	LeetCode OJ Problem	77	Reverse	139	107	77%
MaxTreePathSum	LeetCode OJ Problem	74	Swap	76	76	100%
EditingDistance	Enhanced Edit Distance Algorithm	73	Reverse	223	169	76%
InterleavingString	LeetCode OJ Problem	73	Reverse	182	113	62%
HeapSort	Heap Sort Algorithm	66	Permute	112	82	73%
multi-MAXSUB	Multi-Segment MAXSUM Algorithm	61	Reverse	166	12	7%
SearchInRot	LeetCode OJ Problem	53	Reverse, Shift	208	192	92%
QuickSort	Quick Sort Algorithm	49	Permute	75	54	72%
FirstMissingPositive	LeetCode OJ Problem	40	Reverse, Shuffle	83	39	47%
MinInRot	LeetCode OJ Problem	34	Permute, Shift	71	65	92%
2D-MatrixSearch	LeetCode OJ Problem	34	Transpose	38	27	71%
DistinctSubsequence	LeetCode OJ Problem	32	Reverse	53	15	28%
MAXSUB	Kadane's MAXSUB Algorithm	25	Reverse, Add	46	21	46%
Prim	Princeton MST Algorithm	765	Transpose, Add, Shift	620	67	11%

We measure the *branch distance* between a reported critical branch and and its corresponding real fault's location on the program's failing execution trace to evaluate how "close" the reported critical branch is to the real fault. Consider a failing execution trace $\pi_f = \alpha_1, c_1, \alpha_2, c_2, \dots, \alpha_n, c_n, \alpha_{n+1}$ and c_i being reported as the critical branch. For each occurrence of the faulty statement (either a c_j or α_j[5]), its distance is measured by the amount of distinct branches between i and j. Formally, we define the distance between i and j to be:

$$d_i(c_j) = \left|\left\{S^{\pi_f}(c_k)\right\} \mid i < k \leq j \vee j \leq k < i\right|,$$

and $d_i(\alpha_j) = \begin{cases} d_i(c_j)+1, & j \leq i \\ d_i(c_j)-1, & j > i \end{cases}$. We define the *branch distance* of π_f and c_i to be the least distance among all occurrences of the faulty statement (i.e., $\min_j d_i(c_j)$ if the fault is in a branch or $\min_j d_i(\alpha_j)$ if the fault is in an assignment statement). This definition captures the intuition that a larger distance requires more amount of code to be inspected to localize the fault by a study of the execution trace starting from the critical branch. Furthermore, we consider a fault to be *precisely located* if the branch distance is 0 (i.e., the faulty statement is directly reachable from the critical branch, or the critical branch itself contains a fault). This kind of bug reports are considered extremely helpful since programmers can directly find the bug by examining the code near the critical branch.

C. Evaluation Results

We conducted our evaluation on a desktop machine with a dual-core i5 CPU and 4GB memory running Ubuntu Linux. Evaluation results are shown in Table II.

First, we study for how many faulty programs can Comedy generate critical branches for. The amounts of faulty programs with critical branches identified are listed in Column 3 of Table II. Comedy finds critical branch for 88.4% among the 2,330 faulty subject programs with a median of 95.7%. For 15 out of 21 subject programs, our success rate is above 90%, and only three are below 80%. These results show that Comedy can mostly generate successful debugging reports.

[5]The faulty statement must appear in π otherwise this test passes the metamorphic relation check.

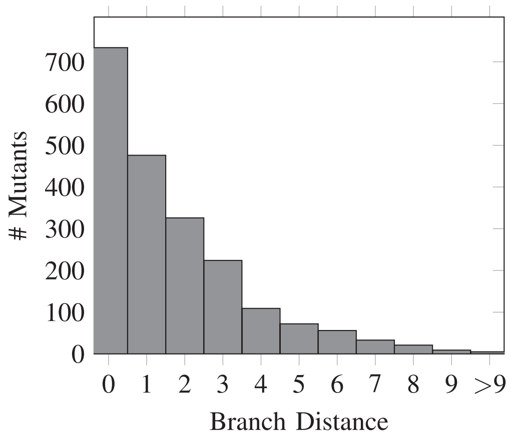


Figure 4: Distribution of Branch Distances

Second, we study the precision of Comedy's reported critical branches. We measure the branch distance for each reported critical branch. Summarizing the Column 4 of Table II, the average branch distance is 1.68, and histogram in Figure 4 shows that most critical branches reported by Comedy are close to the real fault. For almost 90% of the reported branches, the branch distance is no more than 3, and our critical branch is precise for 36% among all reported critical branches (Column 5). These results suggest that developers only need to examine a few parts of the program to find the real fault using the Comedy's debug report.

Table II: Evaluation Results

Subject Name	*# Killed*	*# Succeeded*	*Average Branch Dis.*	*# Precisely Located*	*Average Time*
ClosestPair	160	145(90.1%)	1.641	68 (46.9%)	81.83s
Dijkstra	161	154(95.7%)	3.580	40 (26.0%)	58.46s
Edmonds-Karp	46	43(93.5%)	2.441	7 (16.3%)	108.22s
MaximumRectangle	156	153(98.1%)	2.072	41 (26.8%)	131.17s
BoyerMoore	59	49(83.1%)	1.490	21 (42.9%)	74.05s
SurroundedRegion	309	250(80.9%)	2.224	45 (18.0%)	52.72s
DecodingWays	400	287(71.8%)	1.864	64 (22.3%)	34.70s
LargestRectangle	107	105(98.1%)	2.133	25 (23.8%)	114.76s
MaxTreePathSum	76	67(88.2%)	0.286	41 (61.2%)	133.29s
EditingDistance	169	169(100%)	0.766	103 (60.9%)	102.63s
InterleavingString	113	113(100%)	1.3	34 (30.0%)	100.03s
HeapSort	82	61(74.4%)	0.295	43 (70.5%)	48.98s
multi-MAXSUB	12	12(100%)	0.5	8 (66.7%)	17.00s
SearchInRot	192	192(100%)	1.522	46 (24.0%)	41.36s
QuickSort	54	41(75.9%)	0.24	31 (75.6%)	55.96s
FirstMissingPositive	39	38(97.4%)	0.842	16 (42.1%)	11.22s
MinInRot	65	63(96.9%)	0.381	44 (69.8%)	34.79s
2D-MatrixSearch	27	27(100%)	0.704	14 (51.9%)	39.83s
DistinctSubsequence	15	14(93.3%)	0.429	12 (85.7%)	24.12s
MAXSUB	21	21(100%)	0.286	15 (71.4%)	10.40s
Prim	67	61(91.0%)	2.344	16 (26.2%)	106.56s

Finally, we collect each debugging run's time, as summarized in Column 6. The average debugging time is 65.81 seconds with a maximum of 133.29 seconds. Note that our evaluation is conducted on a commodity desktop computer, and this time cost is considered acceptable for an ordinary developer.

D. Case Study

We also conducted a preliminary case study on the widely-used Web server Tomcat 6.0. We demonstrate the debugging practice of Comedy by localizing an injected error. We simulated a common program mistake of using the wrong constant `SP` instead of `HT` at Line 422 in `InternalInputBuffer` class which should be "`if (buf[pos]==Constants.SP || buf[pos]==Constants.HT){`", as illustrated in Figure 5. We exploited a metamorphic relation stating that removing redundant whitespace characters in the requested URI does not change the contents returned, suggesting that the request string "`Get␣\t/uri␣Http/1.1`" should return the same page as "`Get␣/uri␣Http/1.1`" does. However, this metamorphic relation is violated because of the fault in Line 422. Then we use Comedy to debug debug Tomcat with the failure-inducing test input "`Get␣\t/uri␣Http/1.1`".

We mocked the library classes so that our Comedy tool can be applied to debug the `InternalInputBuffer` class. Comedy first synthesized a program that contains the two pieces of execution and a metamorphic relation check, and used our input to obtain a failure-inducing trace, and then iteratively generated new executions by concolic testing. When the branch condition "`buf[5]`$\neq$32 (32 is the ASCII code of space, or `Constants.SP`)" was switched, Comedy solved a path constraint containing "`buf[5]`=32" and obtained an alternative input (an HTTP request string "`Get␣␣/uri␣Http/1.1`"), and this input can pass the metamorphic relation check. Therefore

```
415 ...
416 while ( space ) {
417   // Read new bytes if needed
418   if (pos >= lastValid) {
419     if (!fill())
420       throw new EOFException(sm.getString("iib.eof.
        error"));
421   }
422   if (buf[pos]==Constants.SP || buf[pos]==Constants.
        SP) {/* Injected fault, the second SP should be
        HT */
423     pos++;
424   } else {
425     space = false;
426   }
427 }
428 ...
```

Figure 5: Faulty Code of `InternalInputBuffer` Class

Comedy reported the statement corresponding to the switched branch condition "`if (buf[pos]==Constants.SP || buf[pos]==Constants.SP){`" to be the critical branch, which exactly contains the fault (Line 422 in Figure 5).

E. Potential Threat

The subjects selected in our experiments are mostly small programs which may not be representative enough. Indeed to further validate the debugging effectiveness we should experiment with larger programs. We plan to address this threat in our future work.

VI. Related Work

Automated debugging techniques are proposed because debugging is labor-intensive and resource-consuming. Slicing-based debugging [3], [4], [38] analyzes control or data dependencies in a concrete execution to isolate a small portion of program code that is considered relevant to fault. Slicing narrows down the scope of a fault in a sound way, i.e., the root cause of a failure is guaranteed to be inside the slicing. On the other hand, soundness also brings a problem that sliced execution traces are often too large for manual inspection [38].

Statistical debugging [9], [10], [39] addresses the manual inspection problem by exploiting a simple observation that, if a pattern frequently occurs in failing executions but not in passing executions, it is more likely to be faulty (called suspicious). By extending this idea, statistical debugging analyzes samples from large amounts of execution traces to vote for suspicious faults. Statistical debugging is strong in localizing faults, but is less informative and barely tells a reason why a localization result is relevant to a target fault [8].

Delta debugging [6]–[8], [40], on the other hand, is orthogonal to the approaches mentioned above. It compares passing and failing execution traces and narrows the difference between them to isolate the statements causing the error. Such techniques aim to find a critical point in the execution that altering it would lead to different testing verdicts (pass or fail). Zeller [6] showed how delta debugging systematically narrows the difference of program states by comparing a passing and a failing runs. He also concentrated on program input and proposed a delta debugging algorithm to simplify and isolate fail-causing input [7]. Delta debugging usually offers more information for developers to find root causes for target failures [8].

The idea of Comedy is inspired by a series of seminal research work. DARWIN [5] isolates a critical branch between two program versions using concolic testing. Wang [25] introduced a technique that can automatically generate a successful run similar to the failing run according to a distance metric and compare these two runs to localize the likely root cause. Ball [41] presented an algorithm which exploits the existence of a set of successful traces and uses them to discover the likely defects in a failing trace. Besides, concolic testing [13]–[15] has been investigated these years. Some studies combine concrete and symbolic execution to localize faults in a buggy program.

However, these automated debugging techniques do not directly apply to our problem since our target programs may only have weak or even no oracle. To address the test oracle issue, metamorphic testing [12] has been studied by a large amount of research [18], [23], [31] and our experimental results suggest that metamorphic relations do exist and can be easily obtained for many real-world programs. For example, for those machine learning [19], graph theory [31], numerical programs [31], [34] and some applications like web search engine [33], compiler [32], there exist various necessary properties and verifying against such metamorphic relations is effective at revealing the existence of program faults.

Still, the violation of a metamorphic relation would involve multiple executions of a program, and normally one is not able to distinguish which is/are actually causing the corresponding failure, presenting a challenge to fault localization. Hence we extend existing debugging techniques to cope with metamorphic relation violations in this paper. Our Comedy belongs to delta debugging and has been inspired by the pioneering work [5], [6], [8], [40], [42]. Comedy provides an analytical approach to porting these ideas to debugging metamorphic relation violations.

There are several pieces of metamorphic-testing based work closely relating to our Comedy. For example, Xie [26] used a new concept called metamorphic slicing which extends statistical debugging to cope with metamorphic testing, and refines debugging results by program slicing. Chen [27] presented semi-proving that uses model checking to verify whether a given metamorphic relation is a tautology in a program. If a counterexample can be found, the equivalent class of failure-inducing inputs would be reported to developers for debugging purposes. Different from these techniques which address fault removal problem, Liu [43] applied metamorphic testing to fault tolerance. Compared with these pieces of existing work, our Comedy has its distinct advantages: it (1) analyzes and integrates debugging results from multiple metamorphic relations, (2) provides more informative debugging reports that contain control-flow information for localizing faults in a program, and (3) scales better than a model checker like semi-proving.

VII. Conclusion

In this paper, we present Comedy, a novel fault localization technique specifically designed for debugging metamorphic relation violations. Comedy adopts the key insight of branch-switching debugging, and uses concolic testing to report a critical branch that is related to the fault. By evaluation on real-world algorithmic programs and a case study, we demonstrated that our Comedy can provide informative debugging aids for programmers in localizing faults. In our future work, we aim to combine our control flow metamorphic debugging technique with data flow analysis (e.g., program slicing) to provide more precise debugging reports.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China.

References

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: Software technologies, engineering processes, and business practices*, 2003.

[3] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI ’90, 1990, pp. 246–256.

[4] M. Weiser, “Program slicing,” in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE ’81, Piscataway, NJ, USA, 1981, pp. 439–449.

[5] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, “Darwin: An approach to debugging evolving programs,” *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 3, pp. 19:1–19:29, Jul. 2012.

[6] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10, 2002, pp. 1–10.

[7] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, Feb 2002.

[8] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 272–281.

[9] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29, Washington, DC, USA, 1996, pp. 46–57.

[10] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 34–44.

[11] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[12] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next test cases," *Department of Computer Science, Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01*, 1998.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, Berkeley, CA, USA, 2008, pp. 209–224.

[14] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005, pp. 213–223.

[15] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005, pp. 263–272.

[16] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[17] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: Symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, 2010, pp. 179–180.

[18] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, 2009, pp. 189–200.

[19] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[20] J. L. Bentley, *Programming pearls*, 2000.

[21] G. Dong, C. Nie, B. Xu, and L. Wang, "An effective iterative metamorphic testing algorithm based on program path analysis," in *Quality Software, 2007. QSIC'07. Seventh International Conference on*. IEEE, 2007, pp. 292–297.

[22] P. Wu, "Iterative metamorphic testing," in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, vol. 1. IEEE, 2005, pp. 19–24.

[23] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *Software Engineering, IEEE Transactions on*, vol. 40, no. 1, pp. 4–22, Jan 2014.

[24] K.-M. Chao, R. C. Hardison, and W. Miller, "Recent developments in linear-space alignment methods: A survey," *Journal of Computational Biology*, vol. 1, no. 4, pp. 271–291, 1994.

[25] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, 2005, pp. 347–351.

[26] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, "Metamorphic slice: An application in spectrum-based fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 866–879, 2013.

[27] T. Y. Chen., T. H. Tse, and Z. Q. Zhou, "Semi-proving: An integrated method for program proving, testing, and debugging," *Software Engineering, IEEE Transactions on*, vol. 37, no. 1, pp. 109–125, Jan 2011.

[28] N. Jussien, G. Rochart, X. Lorca *et al.*, "Choco: An open source java constraint programming library," in *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008, pp. 1–10.

[29] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590, Jul. 2007, pp. 298–302, berlin, Germany.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[31] Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004, pp. 346–351.

[32] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 270–279.

[33] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, 2012.

[34] T. Y. Chen, J. Feng, and T. Tse, "Metamorphic testing of programs on partial differential equations: A case study," in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. IEEE, 2002, pp. 327–333.

[35] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: A mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 827–830.

[36] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '96, 1996, pp. 121–134.

[37] X. Wang, S.-C. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 45–55.

[38] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04, 2004, pp. 94–106.

[39] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07, 2007, pp. 5–15.

[40] W. N. Sumner and X. Zhang, "Comparative causality: Explaining the differences between executions," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, Piscataway, NJ, USA, 2013, pp. 272–281.

[41] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03, 2003, pp. 97–105.

[42] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10, 2010, pp. 49–60.

[43] H. Liu, I. I. Yusuf, H. W. Schmidt, and T. Y. Chen, "Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 420–423.