Improving Failure Detection by Automatically Generating Test Cases Near the Boundaries

Min Zhou, Xi Cheng, Xinrui Guo, Ming Gu
School of Software
Tsinghua University
Beijing, 100084, China
Email: zhoumin03@gmail.com

Hongyu Zhang
Microsoft Research
Beijing, 100084, China
Email: honzhang@microsoft.com

Xiaoyu Song
Electrical and Computer Engineering
Portland State University
Portland, 97207, USA
Email: song@ece.pdx.edu

***Abstract*—Boundary value analysis is a typical conventional testing technique. However, manually identifying input regions and writing test cases are labor-intensive and time-consuming. In this paper, we propose a search-based random testing approach, which automatically generates test data along the boundaries of semantic regions of the input domain. The experiments on mutated programs confirm the effectiveness and efficiency of the proposed approach. Furthermore, our approach significantly outperforms the conventional ART (Adaptive Random Testing) methods, which sample test cases evenly across the input regions. Our approach also outperforms EvoSuite, a state-of-the-art tool that generates test cases satisfying certain coverage criterion.**

***Keywords*—boundary value; random testing; memetic;**

I. Introduction

Software testing is a primary approach to ensure the quality of software systems. However, the current testing practice is still largely labor-intensive and time-consuming. Huge amount of manual efforts are required to design and execute test cases in order to reveal code defects[1]. Over the years, various automatic test data generation techniques have been proposed to facilitate software testing. For example, the Random Testing (RT) technique [2], [3], which generates purely random test inputs, have been successfully applied to testing software such as Java Just-In-Time (JIT) compilers [4] and Windows NT applications[5]. Adaptive Random Testing (ART) [6], [7], [8] aims to improve the efficiency of random testing by reducing the size of the test suite while still triggering test failures. Although the effectiveness of ART is questioned by some researchers[9], there are results show that ART improves RT in certain scenarios by detecting the first failure using 30% to 50% fewer test cases[7]. Many search-based techniques have also been proposed to automatically generate test cases that satisfy certain coverage criterion [10], [11], [12], [13].

Boundary value analysis [14], [15] is a typical conventional testing technique. Technically, a program unit can be viewed as a function. Test data can be viewed as sample points in the input domain. The input domain is partitioned to regions that consist of test inputs which produce the same execution trace. It has long been recognized that by testing boundary values of the input region, failure detection performance could be improved[15]. However, the existing boundary value based testing techniques require testers to manually identify input regions and generate test cases, which is very labor-intensive and time-consuming, especially when there is no formal specification of the program unit.

In this paper, we show that better testing performance can be achieved by automatically generating test cases along the boundaries of input regions. First, we locate the boundaries and then generates test cases close to each boundary. We guide the test case generation process using the memetic algorithm [16], [17], which is an effective evolutionary algorithm that combines genetic and local search. The fitness function is constructed by two distance metrics, namely branch distance (which reflects both the structural distance in a control flow graph and closeness of the current execution to a target branch) and boundary distance (which is the distance from a test input to a certain boundary).

To evaluate our approach, we have implemented a prototype in Java and compared it with various existing approaches, including ART and a branch coverage based method. We also compare our prototype against EvoSuite[13], a state of the art search-based test case generating tool which generates test suites towards satisfying a coverage criterion. The subject programs include common search and sorting algorithms, numerical functions, and real-world code used in a CAD system. We adopt the mutation analysis techniques as proposed by Offutt[18] to generate mutants of these subject programs. Each mutant can be considered as a faulty version. We then apply the proposed approach to generate test cases that are able to kill these mutants (i.e., causing the mutant programs to fail). The experimental results show that, with the same number of test cases, on average 82.77% mutants can be killed by our approach, while only 45.48% - 60.85% mutants can be killed by the conventional ART techniques. Compared with EvoSuite, our approach generates test suites of comparable branch coverage but capable of killing more mutants.

The major contributions of this paper include:

1) We propose to automatically sample test cases near the boundaries of regions in order to achieve a higher failure detection rate.
2) We design and implement an evolutionary algorithm that generates inputs near the boundary. Experimental results show that its failure detection outperforms that of the related methods.

The rest of the paper is organized as follows: we introduce

Min Zhou, Xi Cheng, Xinrui Guo and Ming Gu are with Key Laboratory for Information System Security (Chinese Ministry of Education) and Tsinghua National Laboratory for Information Science and Technology (TNList)

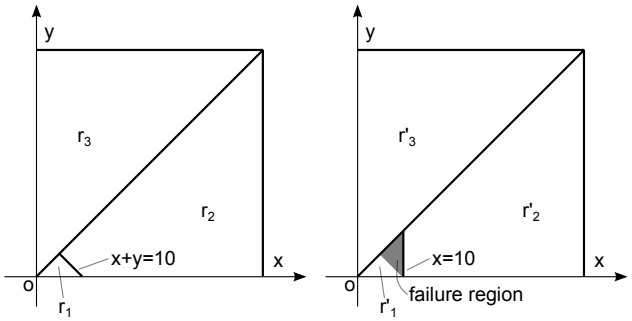
a motivating example in Section II and preliminaries in Section III. Section IV gives an overview of the proposed test data generation method. Section V describes the memetic algorithm and the fitness function in detail. Experimental results and analysis are presented in Section VI. Section VII summarizes the related work and Section VIII concludes the paper.

II. A Motivating Example

Consider the code snippet in Figure 1. The function `foo` contains two nested `if` statements (with the condition `x > y` and `x + y >= 10` respectively). The return value depends on the branches that are taken in execution. Such a branching structure is very common in programming.

```
1  public int foo(int x, int y) {
2      if (x > y) {
3          if (x + y >= 10) {
4              return 1;
5          } else {
6              return 0;
7          }
8      } else {
9          return x;
10     }
11 }
```

Fig. 1. Example (correct version)



(a) Correct program (b) Faulty program

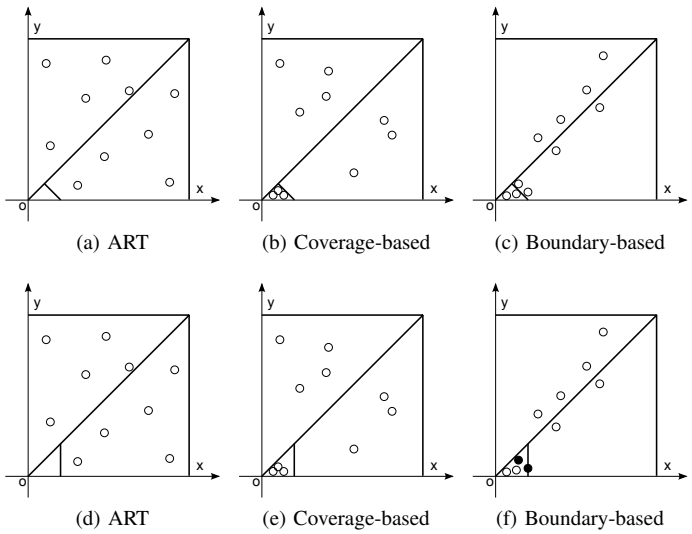
Fig. 2. Regions of the example

We assume that both x and y are integers range from 0 to 100, i.e., the input domain is $D = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x, y \leq 100\}$. The function `foo` partitions D into 3 regions $R = \{r_1, r_2, r_3\}$, as show in Figure 2a, i.e.:

$$\begin{aligned} r_1 &= \{(x, y) \mid x, y \geq 0 \wedge x > y \wedge x + y < 10\} \\ r_2 &= \{(x, y) \mid x > y \wedge x + y \geq 10 \wedge y \geq 0 \wedge x \leq 100\} \\ r_3 &= \{(x, y) \mid 0 \leq x \wedge x \leq y \wedge y \leq 100\} \end{aligned}$$

Inputs within the same region follows the same control flow. The regions are defined implicitly by the conditions and operations along the control flow path. Now, suppose that the program is modified during maintenance and the condition `x + y >= 10` is modified to `x >= 10` by mistake. The partition of regions is thus changed accordingly as in Figure 2b. We investigate how random testing helps us discover this fault.

A test suite is a set of test cases (input + expected output). The generated test cases scatter across the input domain,



(a) ART (b) Coverage-based (c) Boundary-based

(d) ART (e) Coverage-based (f) Boundary-based

Fig. 3. Distribution of the generated test inputs (the first row is for the correct program while the second row is for the faulty program.)

following certain distributions that depend upon the generation strategy. Figure 3 shows the distribution of test cases under various strategies for the regions in Figure 2. Each test suite contains exactly 10 test cases. If the ART strategy is used, the test cases are sampled evenly in the input domain with respect to certain distance metric. Figure 3a shows an example of using the ART strategy with Euclidean distance metric. It is discussed in [7] that evenly distributed test cases are helpful in revealing failures. If test cases are generated with respect to some coverage criterion, for instance the branch coverage criterion, they may distribute as what is shown in Figure 3b, where test cases spread over all regions. The main idea of this paper is that *inputs closer to the boundaries are more likely to reveal failures*. Traditionally, boundary value analysis requires testers to manually identify input regions and generate test cases. Our strategy is to automatically generate test cases along the boundaries. Figure 3c shows the test cases generated following our strategy.

A test case is said to reveal a failure if the program under test produces an unexpected output when fed the input. The results of applying the generated test suites to the faulty version are shown in Figure 3d, 3e and 3f. The faulty program passes the test suites generated under the former two strategies. However, the test suite generated by our strategy is able to cause a failure. The intuition is that, if the program is wrong, the partition of regions must be different at some boundary (for instance, the failure region in Figure 2b). Test cases closer to the boundaries are sensitive to such disturbance and are likely to distinguish the correct and faulty versions.

In this paper, we extend this basic idea and propose a method that can efficiently detect failures by automatically generating test cases close to the boundaries of regions.

III. Preliminaries

The Control Flow Graph (CFG) of a function f is a directed graph $G_f = (N, E)$ where N is a finite set of nodes and E is a finite set of edges. Each node n_a corresponds to a

statement a in f. If a statement a' is executed immediately after a, an edge $(n_a, n_{a'})$ is added. Every CFG contains two special empty nodes n_{in} and n_{out} (called the *entry* and *exit* respectively). For the first statement a_1 in f, we add an edge (n_{in}, n_{a_1}). An edge $(n_{a'}, n_{out})$ is added for each node $n_{a'}$ that is associated with a statement after which the control flow leaves the function. Specially, the control flow of an empty function is $N = \{n_{in}, n_{out}\}$ and $E = \{(n_{in}, n_{out})\}$. A path in the CFG is a sequence of nodes "$n_{i_1}, n_{i_2}, \ldots, n_{i_k}$" such that $(n_{i_j}, n_{i_{j+1}}) \in E$ for all $1 \leq j < k$.

There are two kinds of nodes in the CFG, the decision nodes and the action nodes. Each decision node corresponds to a branch statement (such as `if`) or the decision point of a loop statement (such as `while`). It has an associated condition (denoted by cond_n) and two successors. Two edges originated from the same decision node are labeled `true` or `false` and are called the true-branch or false-branch respectively. The *branch condition* of $e = (n, n')$ is denoted by:

$$\mathsf{cond}_{(n,n')} = \begin{cases} \mathsf{cond}_n & \text{if } (n, n') \text{ is the true-branch} \\ \neg\mathsf{cond}_n & \text{if } (n, n') \text{ is the false-branch} \end{cases}$$

Two branches from the same decision node have complementary branch conditions. Action nodes are basic statements that contain assignments or function invocations. Since each of them has exactly one successor, consecutive action nodes are usually grouped into a basic block. Each basic block is a maximal sequence of statements $\langle n_{i_1}, n_{i_2}, \ldots, n_{i_k} \rangle$ such that n_{i_j} is the only predecessor of $n_{i_{j+1}}$ and $n_{i_{j+1}}$ is the only successor of n_{i_j}. For simplicity, basic blocks are represented by a single action node in the CFG.

Throughout the paper, the program unit under test is denoted by p . Its formal parameters are denoted by $(x_1, x_2, \ldots, x_n)$. An input (or a test case) c is a map from the formal parameters to concrete values. An execution trace is a path from n_{in} to n_{out} (we assume the execution always terminates). Use $\pi_p(c)$ to denote the execution trace of p under the input c. The subscript $\cdot_p$ is omitted if no ambiguity is caused.

There are several coverage criteria used in practical testing, e.g., the statement coverage and branch coverage. The former one requires that each statement is executed at least once by a test suite while the latter one requires each branch be taken. A test target of branch coverage is a branch in the CFG. We use $B = \{b_1, b_2, \ldots, b_n\}$ to denote all test targets. A target $b = (n, n')$ is said to be covered by c if $(n, n') \sqsubseteq \pi(c)$, i.e., (n, n') is a subsequence of $\pi(c)$.

```
1  public int factorial(int x) {
2        if (x >= 0) {
3              int r = 1;
4              for (int i = 1; i <= x; i++)
5                    r = r * i;
6              return r;
7        } else {
8              return 0; // error
9        }
10 }
```

Fig. 4. Factorial function

Definition 1 (Region): A region is a set of inputs whose execution traces share the same set of CFG nodes. Two inputs c_1 and c_2 are in the same region r if and only if $n \in \pi(c_1) \iff n \in \pi(c_2)$ holds for all $n \in N$.

Boundaries are the areas where two or more regions join. The partition of regions depends not only on the syntax but also on the semantics. For the motivating example, its regions are given in Figure 2. Inputs who share the same execution trace are in the same region. Moreover, take the example shown in Figure 4, which computes the factorial of x. The execution traces of $x = 6$ and $x = 7$ are in the same region because they cover the same set of statements (lines 2, 3, 4, 5, 6) although they differ in the number of iterations. But $x = 0$ and $x = 6$ are in different regions because the inside of the loop (line 5) is not reached by $x = 0$.

IV. Overview of Our Approach

In our approach, test cases are automatically generated in each region, near the boundary but not exactly on it. For this purpose, we need to deal with two subproblems: (1) Locate the boundaries and (2) Generate test cases close to each boundary.

A. Locate Boundaries

Input domain is partitioned based on the program structure and decision conditions. Points in two adjacent regions are test cases that satisfy or falsify the branch condition corresponding to the boundary. The distance from a test case to the boundary is quantified by the degree of satisfiability of branch condition. For instance, $p_2(80, 50)$ is farther from b_1 than $p_1(60, 50)$. Such can be verified by checking the degree of satisfiability of `x > y`. The degree is quantified as 10 $(60 - 50)$ for p_1 while it is quantified as 30 $(80 - 50)$ for p_2. We prefer test cases that are closer to the boundary, i.e., p_1.

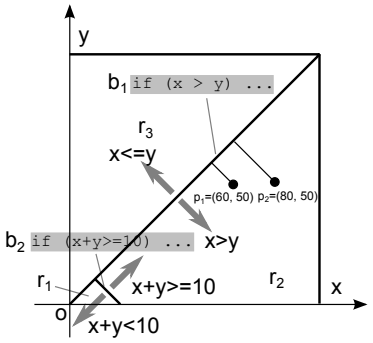


Fig. 5. Regions and boundaries of the motivating example

B. Test Case Generation

To generate test cases (on both sides) along the boundary, inputs that cover both branches of the decision node are generated. This part is similar to existing works that aim at achieving high branch coverage [10], [11], [12], [19]. However, branch coverage is not enough for efficient failure detection. We further measure *how well the branching condition is satisfied* and push the candidates towards boundary.

Algorithm 1 is a very high-level description of the test suite generation algorithm. Technically, it (1) generates inputs that cover a certain branch target, (2) optimizes the boundary distance to push them towards the boundary, (3) selects a

Input: $B \leftarrow$ the set of all branch targets
Output: a test suite S
foreach $b \in B$ **do**
  $P \leftarrow$initPopulation(b);
  for $i \leftarrow 1$ *to* MaxIteration **do**
    $P \leftarrow$search(b, P);
    $ts(b) \leftarrow$ top ($ts(b)$, P);
  end
end
$S \leftarrow$reduce($\cup_{b \in B} ts(b)$);
return S;

Algorithm 1: Test suite generation

fixed number of inputs as the test suite. (1) and (2) are optimization problems with complex objectives, thus can be solved by an evolutionary search algorithm. We use a memetic algorithm which combines genetic search and local search. "search" represents an iteration of the memetic algorithm. It tries to improve the fitness of the current population. The fitness function measures whether a test case covers a certain branch target and how close it is to the boundary. The "top" function selects the best test cases from the current test suite and the population. The "reduce" function reduces the size of population by selecting a fixed number of inputs. There are various selection strategies[20] and we use a simple one that selects test cases that lie in cold regions (i.e., regions that contain fewer test cases).

V. The Search Algorithm

The core of out approach (i.e., the "search" function) is explained in this section. The essences of the algorithm are (1) the fitness function that is designed to guide the search towards the boundary, (2) a set of memetics to manipulate the input candidates.

The algorithm is explained with the example shown in Figure 6, which is a function that accepts the length of three sides of a triangle and returns its type (either `Invalid`, `Acute`, `Obtuse` or `Right`). The input domain, as well as all regions, are subsets of $\mathbb{Z}^3$. Note that the boundaries are not planes because of the non-linear expression at line 18.

In general, randomly covering all branch targets is not easy. In this example, the branch at line 25 is only covered by a triple (x, y, z) that forms a valid right triangle. Uniformly generated random (x, y, z) (each from 16-bit unsigned integers $[0, 65535]$) have the probability 2.49×10^{-9} to satisfy $x^2 + y^2 = z^2$, which is extremely low.

Evolutionary search algorithm is often used to generate test cases with high branch coverage. The search algorithm is guided by a fitness function which measures *how good a certain input is*. For instance, assume that we want to cover the branch at line 25 and three inputs are randomly generated: $c_1 = (0, 1, 1)$, $c_2 = (6, 8, 193)$ and $c_3 = (6, 8, 11)$. A proper fitness function would produce the following rank $c_3 > c_2 > c_1$. c_1 is considered to be the worst since it fails at the very beginning. Between c_2 and c_3, the latter one is better because they both fail at the condition $x^2 + y^2 = z^2$ but c_3 is closer.

The major difference between our algorithm and previous work is that, the search algorithm does not stop when it finds a test case that covers a certain branch, it further pushes the test case towards a boundary so that it may reveal more defects. This is also guided by a fitness function which measures the distance from a test case to a certain boundary. For example, assume that we want to push towards the boundary introduced by line 15 and three inputs are found to cover the branch: $c_4 = (1, 4, 10)$, $c_5 = (1, 4, 7)$ and $c_6 = (1, 4, 6)$. Although they all take the true-branch on line 15, c_6 is closest to the boundary because it falsifies `x + y <= z` at a minimal degree.

```
1  public byte classify(int x, int y, int z) {
2     if (x <= 0 || y <= 0 || z <= 0) {
3        return INVALID_TRIANGLE;
4     } else {
5        if (x > y) {
6           int t = x; x = y; y = t;
7        }
8        if (y > z) {
9           int t = y; y = z; z = t;
10       }
11       if (x > z) {
12          int t = z; z = x; x = t;
13       }
14       // now, x <= y <= z
15       if (x + y <= z) {
16          return INVALID_TRIANGLE;
17       } else {
18          long delta = (x*x + y*y - z*z);
19          if (delta > 0) {
20             return ACUTE_TRIANGLE;
21          } else if (delta < 0) {
22             return OBTUSE_TRIANGLE;
23          } else {
24             // delta == 0
25             return RIGHT_TRIANGLE;
26          }
27       }
28    }
29 }
```

Fig. 6. A triangle classifier

A. Memetic Algorithm

Let $B = \{b_1, \ldots, b_{|B|}\}$ be the set of branch targets. As shown in Algorithm 2, we iteratively search for test cases that are near the boundary introduced by a branch target b.

The outlined algorithm is a typical memetic algorithm framework[21]. It is called by Algorithm 1 iteratively. Genetic and local search are performed at each iteration to produce new candidates. Selection is applied at the end of each iteration. It terminates when good enough candidates are produced or the maximal number of iterations is reached.

B. The Fitness Function

The fitness function significantly affects performance of the search algorithm. For our particular problem, given a target b and a candidate c, the fitness of c is evaluated based on the trace $\pi(c)$. The fitness value measures *how close it is to the boundary*. It is defined by two metrics:

- Branch distance: It consists of two parts: (1) the minimal number of control dependent nodes from

Input: $b \leftarrow$ a branch target, $C \leftarrow$ init population.
Output: optimized population C
$C_1 \leftarrow \emptyset$, $C_2 \leftarrow \emptyset$;
// Genetic phase
for $i = 1$ **to** CrossoverSize **do**
  pick $x, y \in C$;
  $z \leftarrow$crossover (x, y);
  mutate(z);
  $C_1 \leftarrow C_1 \cup \{z\}$;
end
// Local search phase
for $i = 1$ **to** LocalSearchSize **do**
  pick $x \in C$;
  $z \leftarrow$localSearch(x);
  $C_2 \leftarrow C_2 \cup \{z\}$;
end
// Selection
$C \leftarrow$select($C \cup C_1 \cup C_2$);
$iteration \leftarrow iteration + 1$;
return C;

Algorithm 2: The search function - an iteration of the memetic algorithm

every node in the trace to the decision node. (2) the distance for the expected branch to be taken at the decision node. Branch distance is first minimized so that the execution trace covers the specific branch.

- Boundary distance: It measures how far is it from the boundary. It is applied after the branch is covered.

Technically, given $G = (N, E)$, a node n' is reachable from another one n, denoted by $n \rightsquigarrow n'$, iff there is a path from n to n'. All reachable nodes of n are $\mathsf{Reach}(n) = \{n' \in N \mid n \rightsquigarrow n'\}$. In a CFG, all successors of a node are denoted by $\mathsf{Post}(n) = \{n' \in N \mid \exists(n, n') \in E\}$ and all predecessors are $\mathsf{Pred}(n) = \{n' \in N \mid \exists(n', n) \in E\}$. Notice that the concept of reachability is defined regardless of conditions on the edges. Obviously, $\mathsf{Post}(n) \subseteq \mathsf{Reach}(n)$ holds for all nodes in N. Without loss of generality, we assume that all nodes are reachable from n_{in} and n_{out} is reachable from all nodes.

Let $\mathsf{fit}_b(\cdot)$ be the fitness function. Let b be a branch target $(t_s, t_d) \in E$, c be an input. Let $\pi_p(c) = n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ be the execution trace produced by c. Assume that the valuation at the node n_{i_j} is v_j (initially, $v_1 = c$), then $\mathsf{fit}_b(c)$ is:

- If b is not covered (t_s may be in $\pi(c)$), the fitness is:

$$\mathsf{fit}_b(c) = \max_{1 \le j < k} \{-\mathsf{dist}_{\prec(t_s)}(n_{i_j}) + \mathsf{dist}_\beta(v_j, n_{i_j} \rightharpoonup n_{i_{j+1}})\}$$

- Otherwise there exists a j such that $n_{i_j} = t_s$ and $n_{i_{j+1}} = t_d$ (i.e., b is covered), then

$$\mathsf{fit}_b(c) = \mathsf{dist}_\delta(v_{i_j}, t_s \rightharpoonup t_d)$$

1) Branch Distance: Branch distance consists of two parts. The first part $\mathsf{dist}_{\prec(n')}(n)$ is the number of isolation nodes from n to n'. The second part $\mathsf{dist}_\beta(v, n_1 \rightharpoonup n_2)$ is the satisfiability degree of the branch condition for the edge (n_1, n_2). The branch distance mentioned in this paper extends similar ideas of the approach level in [22] and the branch evaluation expression in [23].

First part: Given a graph $G = (N, E)$ and a pair of nodes $n, n' \in N$. n is an isolation node for n' if and only if: (1) $n \neq n' \wedge n' \in \mathsf{Reach}(n)$. (2) $\exists v \in \mathsf{Post}(n).\ (n' \notin \mathsf{Reach}(v))$. Intuitively, n is an isolation node for n' if and only if n' is reachable from n but n' will no longer be reachable if an unwanted decision is made at n. $\mathsf{Isolation}(n')$ denotes the set of isolation nodes for n'. $\mathsf{dist}_{\prec(n')}(n)$ denotes the minimal number of isolation nodes over all paths from n to n'. It is $+\infty$ if $n' \notin \mathsf{Reach}(n)$. Notice that $\mathsf{dist}_{\prec(n')}(n)$ is asymmetric.

Given a CFG $G = (N, E)$ and $n \in N$. We call $\mathsf{dist}_{\prec(n)}(\cdot)$ the *isolation distance* and CFG labeled with $\mathsf{dist}_{\prec(n)}(\cdot)$ the *isolation graph*. They can be constructed efficiently using an algorithm which is similar to the construction of dominant graph. Tdime complexity is $O(|E| \cdot \log(|N|))$.

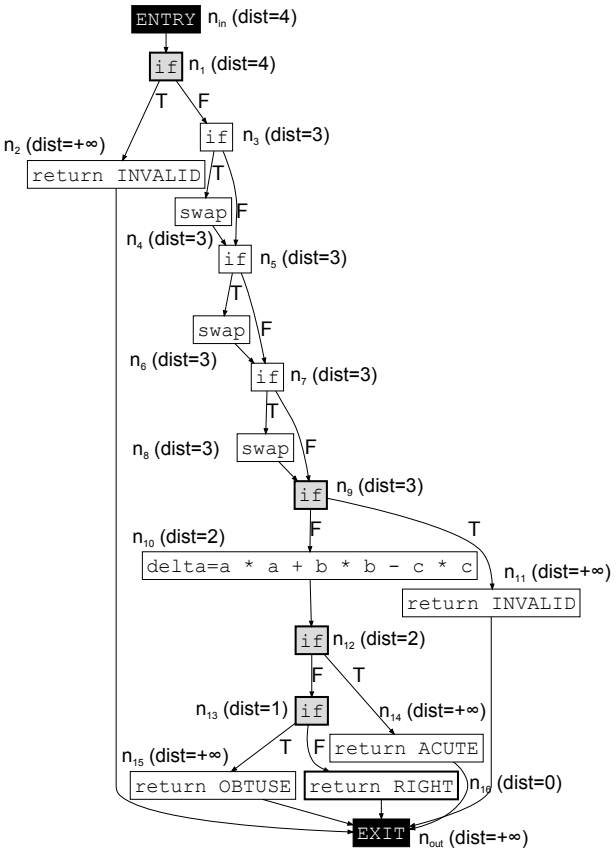


Fig. 7. Isolation graph of the example

Take the `TriangleClassifier` example, the isolation graph for the node n_{16} (`return RIGHT_TRIANGLE;`) is shown in Figure 7. The isolation nodes for n_{16} are colored in gray ($\mathsf{Isolation}(n_{16}) = \{n_1, n_9, n_{12}, n_{13}\}$) and the isolation distance to n_{16} is labeled on each node. From n_1, at least 4 desired decisions should be made (at n_1, n_9, n_{12}, n_{13}), to n_{16}, i.e., $\mathsf{dist}_{\prec(n_{16})}(n_1) = 4$. But only 2 desired decisions are required from n_{12} to n_{16}, thus $\mathsf{dist}_{\prec(n_{16})}(n_{12}) = 2$. Notice that the decision nodes n_3, n_5, n_7 are not isolation nodes because they all potentially lead to n_{16} no matter which branch is taken.

Second part: When the decision node of a branch is reached but the execution goes along the other branch, the distance for taking a desired branch is measured. It is defined based on the satisfying distance of condition expressions. Each

TABLE I. ϕ: SATISFYING DISTANCE FOR ATOMIC FORMULAS

f	Type	$\phi(v, f)$
$l > r$	disc.	$\mu(v(l) - v(r) - 1)$
$l \geq r$	disc.	$\mu(v(l) - v(r))$
$l < r$	disc.	$\mu(v(r) - v(l) - 1)$
$l \leq r$	disc.	$\mu(v(r) - v(l))$
$l \geq r$	cont.	$\mu(v(l) - v(r))$
$l > r$	cont.	$\mu(v(l) - v(r) - \epsilon)$
$l \leq r$	cont.	$\mu(v(r) - v(l))$
$l < r$	cont.	$\mu(v(r) - v(l) - \epsilon)$
$l = r$	any	$-\mu(\lvert v(r) - v(l) \rvert)$
Other Bool. expr. e	Bool.	$v(e)?1 : -1$

decision node is associated with a condition expression, which is a Boolean combination of atomic formulas. Each atomic formula is either a Boolean variable, a predicate ("$=, <, \leq$", etc.) or a function invocation which returns a Boolean value. For instance, "$f := \neg(b \vee (x_1 = x_2) \wedge \mathsf{contains}(s, x))$". A valuation v is a map from variables to concrete values. The value of f under v is denoted by $v(f)$. We write $v \vDash f$ if $v(f)$ is `true`, otherwise we write $v \nvDash f$.

Definition 2 (Satisfying distance): Given a formula f and a valuation v. The satisfying distance of v is:

$$\mathsf{sd}(v, f) = \begin{cases} \min\{\mathsf{sd}(v, f_i)\} & \text{if } f := f_1 \wedge f_2 \wedge \ldots \wedge f_n \\ \max\{\mathsf{sd}(v, f_i)\} & \text{if } f := f_1 \vee f_2 \vee \ldots \wedge f_n \\ -\mathsf{sd}(v, f') & \text{if } f := \neg f' \\ \phi(v, f) & \text{if } f \text{ is atomic} \end{cases}$$

The satisfying distance measures how well f is satisfied by v. It is defined recursively and based on the ϕ function which evaluates the satisfying distance for atomic formulas. Let $\mu : \mathbb{R} \mapsto [-1, 1]$ be any monotonically increasing function that maps real numbers to a bounded interval and $\mu(0) = 0$.

Table I gives the definition of ϕ that is used in this paper. ϕ is computed depending on the structural pattern and data types in the expression. In the table, "discr." are discrete data types such as `int`, `byte`, `char` and "cont." are continuous data types such as `float` and `double`. ϵ is a small positive value which is chosen according to the precision of the corresponding data type. In this paper, we use $\mu(x) = 2 \cdot \mathsf{arctan}(x)/\pi$.

From the definition, it holds that $v \vDash f$ iff $\mathsf{sd}(v, f) \geq 0$, $v \nvDash f$ iff $\mathsf{sd}(v, f) < 0$. Then $\mathsf{dist}_\beta(v, n \rightharpoonup n')$ is defined as:

$$\mathsf{dist}_\beta(v, n \rightharpoonup n') := \mathsf{sd}(v, \neg \mathsf{cond}_{(n,n')})$$

Given a branch b, the search algorithm is guided by branch distance to obtain an input c that covers b.

2) Boundary Distance: If the target $b = (n, n')$ is covered by an input c, then c must be already in the region that corresponds to b. For efficient failure detection, we want c to be near the boundary. The satisfying distance is used to measure *how far is it from c to the boundary introduced by b.*

First of all, since $b = (n, n')$ is already covered by c, we know that (n, n') is in the trace. Assume that v is the valuation at the node n, then it must hold that $v \models \mathsf{cond}_{(n,n')}$. Therefore $0 \leq \mathsf{sd}(v, \mathsf{cond}_{(n,n')}) \leq 1$. The boundary distance $\mathsf{dist}_\delta(\cdot, \cdot)$ is:

$$\mathsf{dist}_\delta(v, b) = 1 - \mathsf{sd}'(v, b)$$

where sd' is a modified satisfying distance. The intuition is that we want values near the boundary but not exactly on it. If we use the satisfying distances$\mathsf{d}$ directly, then the values on the boundary will have better fitness, which is not desired. The definition of sd' differs from that of sd in the way that the computation for atomic formulas (ϕ) is replaced with a new version ϕ' which has a barrier that prevents a value from being exactly on the boundary. For inequalities such as $l \leq r$, the computation of $\mu(v(r) - v(l))$ is modified to $\mu(barrier(v(r) - v(l)))$. The $barrier(x)$ function ensures that test cases at a certain close distance are preferred. The fitness reduces when it is too close or too far.

3) Example: Take the example in Figure 6, assume the current branch target b is the branch from line 19 to 20. Then the candidates in Table II are ranked: $c_1 < c_2 < c_3 < c_4 < c_5$.

TABLE II. EXAMPLES OF FITNESS EVALUATION

Input	Fitness	Note
$c_1 = \{0, 4, 4\}$	< -3	fails early at line 2
$c_2 = \{1, 2, 10\}$	< -2	fails early at line 15
$c_3 = \{3, 4, 6\}$	< -1	fails at line 19
$c_4 = \{3, 3, 3\}$	$[-1, 0]$	$-\mu(barrier(3^2 + 3^2 - 4^2))$
$c_5 = \{3, 4, 3\}$	$[-1, 0]$	$-\mu(barrier(3^2 + 3^2 - 3^2))$

C. Memetics

Memetics are a group of heuristics that are used to modify a test case[21]. We implemented the following memetics:

- `random()`: generates a random value in the domain.
- `randist()`: generates a random value following certain distribution (e.g. $p(x) = 1/(x + 1)$).
- `extreme()`: generates an extreme value in the domain (e.g. 255 for `byte`).
- `special()`: generates a special value in the domain (e.g. $\{0, 1, -1\}$ for `int`).
- `neighbour(`x`)`: generates a neighbor of an existing value x (used in genetic search).
- `neighbours(`x`)`: generates all neighbors of a value x (used in local search).

The memetics are implemented differently for elementary types. For dependent types such as lists (`List<T>`) or arrays (`T[]`), their memetics are implemented upon those of the underlying their basic type `T`. Basic operations that manipulates a list (or array) are adding, modifying and removing an element.

D. Implementation Issues

Due to the *short-circuit evaluation* feature of programming language[1], the evaluation of a condition expression may not be completed. Thus we use partial satisfying distance $\mathsf{sd}^*(\cdot, \cdot)$ instead of the satisfying distance $\mathsf{sd}(\cdot, \cdot)$.

$$\mathsf{sd}^*(v, f) = \begin{cases} \mathsf{sd}(v, f_q) & \text{short-circuit happens} \\ \mathsf{sd}(v, f) & \text{otherwise} \end{cases}$$

f_q is the subset of f that have been evaluated. It can be proven that $|\mathsf{sd}^*(v, f)| \leq |\mathsf{sd}(v, f)|$. Furthermore, $\mathsf{sd}^*(v, f) \geq 0 \iff \mathsf{sd}(v, f) \geq 0 \iff v \vDash f$ and $\mathsf{sd}^*(v, f) < 0 \iff \mathsf{sd}(v, f) < 0 \iff v \nvDash f$ hold. Thus the partial satisfying distance is a weaker estimation for the satisfying distance. But the former one is practically computable.

[1] http://en.wikipedia.org/wiki/Short-circuit_evaluation

VI. EXPERIMENTS

A. Experiment Design

To evaluate the performance of our method in failure detection, we developed a prototype in Java. Figure 8 depicts the experiment procedure. Given a program unit, we (1) extract the CFG from the source code, (2) instrument necessary code to evaluate the fitness, (3) run the memetic algorithm to generate a test suite. CFG is obtained by analyzing Abstract Syntax Tree (AST) of the Java code with JDT. The program is instrumented upon AST, which is later exported to well-formed Java code and compiled to byte codes. Then the memetic algorithm is used to generate test cases and a test suite is obtained. On the other hand, the original program is mutated to a group of mutants. Mutation is performed using μJava [18], which mutates Java programs using the rules in Table III. Finally, we feed the test suite to both the original program and mutants and compare their outputs. A mutant is *killed* if its output is different from that of the original program (i.e., the mutant program fails).

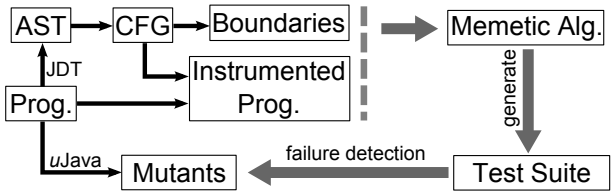


Fig. 8. Experiment design

TABLE III. MUTATION OPERATORS IN μJAVA

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Experiments were performed on a dozen of Java programs, some of which are commonly used routines of the Apache Commons Project. Others include some program units extracted from a real-world CAD system[2] whose McCabe cyclomatic complexity[24] is relatively large. Furthermore, since we found that well written Java code usually consists of small units, we add some artificial programs with complicated control flows. Each program is measured by the number of statements and lines of code. Details are presented in Table IV. The "#stat" column shows the number of statements and the "C-C" column shows the cyclomatic complexity values.

The failure detection performance of our approach is compared with several related methods:

- ART1: An ART method that measures the distance of inputs based on the Euclidean distance in the input domain[7].
- ART2: An ART method that measures the Jaccard distance of the set of statements in its execution trace[25]. In this way, ART is used as a white-box testing method. The Jaccard distance[3] of two sets A and B is defined as $D(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$.
- BCS: As branch coverage is an important coverage criterion in structural testing, this method tries to generate a set of test cases that achieve high branch coverage. We also implement this method by a memetic algorithm without considering the boundary distance. Although test cases are generated from regions, they are not necessarily near the boundaries.

As reported in related work [7] and [8], ART techniques perform better than pure random testing (RT) techniques, therefore in this paper we do not compare our method with RT techniques.

B. Results on Failure Detection

To evaluate the performance of our method in failure detection, we examine the hypothesis that the proposed method kills more mutants than ART1, ART2 and BCS. The number of killed (or/and alive) mutants is compared.

Since ART1, ART2 and BCS are all implemented by evolutionary algorithms, the number of iterations should be bounded. For a fair comparison, the upper bound is the bound for our algorithm multiples the size of the population, i.e., it is $|population| \times \mathsf{maxIteration}_{memetic}$. For the memetic algorithm, the population size is set to 50. In each iteration, 5 new candidates are obtained by crossover and mutation. The local search is restricted to be in a small neighborhood which is reachable within 3 steps from the current value. In the local search, at most 2 input parameters for the unit could be modified simultaneously. The setting of parameter is obtained empirically and fixed prior to the experiments for all cases.

TABLE V. FAILURE DETECTION RESULTS

Prog.	Total Mutants	Alive			
		ART1	ART2	BCS	Ours
mot	91	86	86	29	**17**
triangle	253	189	165	64	**55**
array	94	73	13	24	**7**
binary	109	88	64	22	**21**
bubble	84	**3**	**3**	**3**	**3**
gcd	285	79	79	40	**35**
heap	350	54	53	**50**	52
arcsinh	371	251	30	48	**16**
pow	37	37	30	7	**4**
prime	67	61	40	**9**	**9**
sec	177	101	101	68	**64**
quad	405	224	218	135	**80**

TABLE VI. OVERALL EVALUATION

Total=2323	ART1	ART2	BCS	Ours
Alive	1246	882	499	**363**
Kill	1077	1441	1824	**1960**
Mutant Score	45.48%	60.85%	77.03%	**82.77%**

[2]http://gems8.com/

[3]http://en.wikipedia.org/wiki/Jaccard_index

TABLE IV. PROGRAMS UNDER TEST

Prog.	LOC	#stat	C-C	Note
mot	11	7	4	The motivating example.
triangle	33	22	8	Classifies triangles to `Invalid`, `Acute`, `Obtuse` or `Right` by their sides.
array	37	23	10	Checks if an array is monotonically increasing or decreasing.
binary	21	11	4	The binary search algorithm of Apache Commons.
bubble	11	10	4	The bubble sort algorithm.
gcd	55	26	10	The binary greatest common divisor algorithm of Apache Commons.
heap	67	41	13	The in-place heap sort algorithm.
arcsinh	25	16	6	Inverse hyperbolic sine function of Apache Commons.
pow	12	5	3	A tricky algorithm that check if a number is a power of 2.
prime	22	15	6	Checks if a number is a prime number.
sec	29	12	6	Solves a second-order equation, from a CAD system.
quad	47	25	12	Checks the type of quadratic shape, from a CAD system.

The results are shown in Table V and Table VI. It is obvious that our proposed method outperforms the related methods. Our method wins ART1 and ART2 in almost all cases and only ties in one case (the "bubble" program). Out of all the 2323 generated mutants, the ART-based algorithm kills at most 1486 (60.85%) mutants in all cases while our method kills 2005 (82.77%) mutants. Compared with BCS, our method only loses in one case (the "heap" program), ties in two cases (the "bubble" and "prime" programs), and wins in all the other cases. For numeric programs such as "quad" and "arcsinh", our method achieves much better results. This is because the proposed method utilizes the satisfying distance that is smooth for numeric programs. Thus the search algorithm is more sensitive to the input and yields better performance. We also perform pairwise t-test to check the statistical significance of the comparisons. The results confirm that our method outperforms the related methods at the significance level 0.05.

C. Compare with EvoSuite

EvoSuite[4] is a tool that automatically generates test cases for Java classes. EvoSuite applies an evolutionary algorithm to generate and optimize whole test suites towards satisfying a coverage criterion. It further suggests possible oracles by adding small and effective sets of assertions. Since it was ranked first in the Unit Test Generation competition in SBST 2013[26], we believe that EvoSuite is a qualified representative of state-of-the-art test case generation tools.

EvoSuite and our method both employ evolutionary algorithms. However, EvoSuite aims to generate test cases that satisfy certain coverage criterion but our method further pushes the cases towards the boundaries. By comparing these two approaches, we may further reveal how utilizing the input region information improves the quality of the generated test suites.

The comparison is conducted as follows. We collect 20 program units[5] including those used in the previous experiment as well as more common routines. We require that the experiment subjects operates on data of elementary types, strings, arrays or lists[6], with no further constraints. We first use EvoSuite to generate a test suite for each of the program unit. For a fair comparison, we then use our approach to generate test suites of the same size as EvoSuite does correspondingly.

[4]http://www.evosuite.org/

[5]http://pan.baidu.com/share/link?shareid=1150704595&uk=2167922693

[6]Our method could be extended to a rich set of data types such as Set, Map, by providing corresponding memetics.

The comparison results are presented in Table VII. The first column lists the abbreviation of subject programs' names. Column "#TC" counts the number of test cases in each generated test suite and the number of mutants for each program is denoted by "#M". Column "Coverage" reports the number and ratio of mutants covered by test suites generated by both tools, i.e., mutants whose mutated statement is covered by the test cases. The last column "Killed" reports the number and ratio of killed mutants.

According to the table, the two tools achieved comparable mutant coverage on all program units. In 11 of 20 programs, the test suites generated by both tools covered 100% mutants, and in 7 more programs this coverage is above 90%. However we noticed an interesting phenomenon as we inspect the statistics on mutants killed. In 14 programs our tool killed no less mutants than EvoSuite. However, in programs such as "gcd", "heap", "arcsinh" and "taxcalc", we observe a prominent improvement in this number as 10-75 more mutants of each programs are killed by our tool. A straightforward illustration is presented in Figure 9 and 10. This is a strong indication that even satisfying the same coverage goal, pushing test data to the input region boundaries certainly improves test suites' ability to distinguish failing programs from the correct ones.

An interesting observation is that our method gains better performance when the number of mutations grows larger, i.e., when the program size becomes larger.

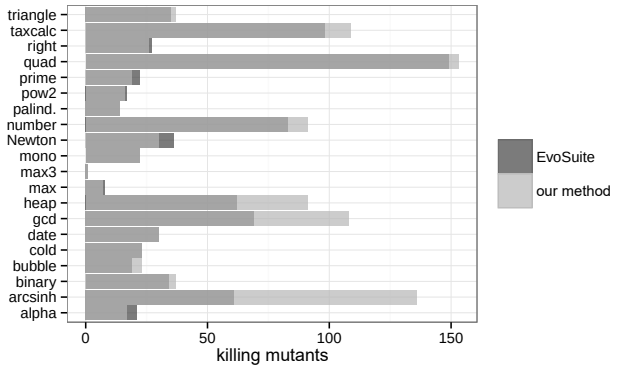


Fig. 9. Comparison with EvoSuite

D. Threats to Validity

The proposed method is mainly designed for unit testing thus the experiment subjects are small programs units. Also,

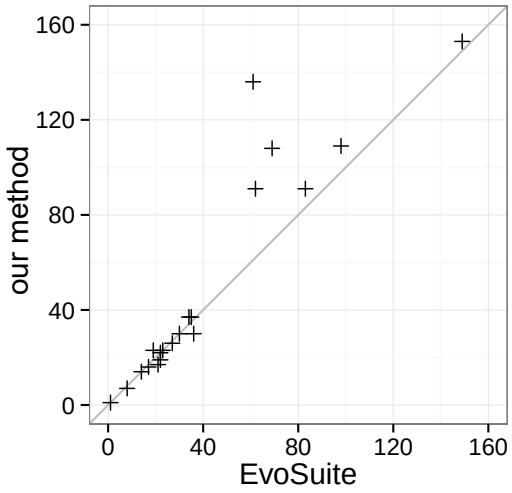


Fig. 10. Comparison with EvoSuite

TABLE VII. DETAILED COMPARISON WITH EVOSUITE

Algo.	#TC	#M	Coverage		Killing	
			Ours	EvoSuite	Ours	EvoSuite
alpha	5	28	28(1.00)	28(1.00)	17(0.60)	21(0.75)
mono	5	42	42(1.00)	42(1.00)	22(0.52)	22(0.52)
binary	3	43	43(1.00)	43(1.00)	37(0.86)	34(0.79)
bubble	1	45	29(0.64)	29(0.64)	23(0.51)	19(0.42)
cold	3	32	32(1.00)	32(1.00)	23(0.72)	23(0.72)
date	5	37	37(1.00)	37(1.00)	30(0.81)	30(0.81)
max	2	20	20(1.00)	20(1.00)	7(0.35)	8(0.40)
gcd	4	155	153(0.99)	143(0.92)	108(0.70)	69(0.45)
heap	2	160	128(0.80)	129(0.80)	91(0.57)	62(0.39)
arcsinh	5	311	311(1.00)	311(1.00)	136(0.44)	61(0.20)
pow2	3	21	21(1.00)	21(1.00)	16(0.76)	17(0.81)
max3	3	6	6(1.00)	6(1.00)	1(0.17)	1(0.17)
Newton	3	47	43(0.92)	45(0.96)	30(0.64)	36(0.77)
palind.	3	20	20(1.00)	20(1.00)	14(0.70)	14(0.70)
right	6	49	49(1.00)	49(1.00)	26(0.53)	27(0.53)
prime	4	30	29(0.97)	30(1.00)	19(0.63)	22(0.73)
quad	16	197	187(0.95)	187(0.95)	153(0.78)	149(0.76)
number	20	268	253(0.94)	255(0.95)	91(0.34)	83(0.31)
taxcalc	7	121	119(0.98)	109(0.90)	109(0.90)	98(0.81)
triangle	6	76	69(0.91)	69(0.91)	37(0.49)	35(0.46)

in the current experiment, our method performs better on numeric programs since their fitness functions are smooth. In the future, we will apply it to a broader class of programs, including larger and more diverse programs. More specifically, the current target is program units but we would like to deal with object-oriented programs using inter-procedural analysis, like what Ciupa et al. did [27]. Furthermore, in this paper the effectiveness of the proposed method is evaluated using randomly generated mutants. In the future, we will evaluate it in real-world software testing practice.

VII. RELATED WORK

Adaptive Random Testing (ART) technique [6], [7], [8] aims to improve the efficiency of random testing by reducing the size of the test suite while still triggering test failures. The idea of ART is to evenly sample random inputs in the input domain. The assumption is that two nearby inputs have a large probability of detecting the same failure pattern. It is pointed out that the failure-causing inputs tend to be clustered within the input domain[7]. Thus, evenly chosen test cases are likely to detect different possible failures. Ciupa et al.[27] also proposed to apply the concept of ART to object-oriented software by defining new distance measures. As discussed in Section IV, our proposed method chooses test cases along the regional boundary. Because our method utilizes the control flow information extracted from source code, it is not surprising that it can outperform the traditional ART method (which is largely a black-box testing method).

Automatic generation of test suite with boundary value analysis was also studied in [28] and [29]. Their methods are based on dynamic symbolic execution (DSE). This paper shares the same spirit in the motivation but differs in the technical roadmap. The generality of DSE is restricted by the power of underlying SMT solvers. For instance, many SMT solvers can not solve path conditions that contain triangular functions. Our approach, however, relies on concrete execution and applies to a border range of programs. The work of Shahbazi[30] is an improvement of RT/ART/QRT by using the centroids of Voronoi regions to improve the effectiveness of the test case production. Their work relies on boundaries but does not prefer test data along boundaries.

In our work, we focus on the program units at the intra-procedure level. In object-oriented (OO) unit testing, generating unit tests often involves the generation of method sequences and the generation of method arguments[31]. Xie et al. proposed Symstra, a framework that achieves both test generation tasks using symbolic execution of method sequences with symbolic arguments[31]. Korat[32] is a novel framework for automated testing of Java programs. Given a formal specification for a method, Korat uses the method precondition to automatically generate all test cases. However, formal specification is not commonly available and symbolic execution is often expensive.

Wegener et al. proposed the metric of approach level in [22], which is similar to the isolation distance in this paper. In Korel's paper[23], a metric similar to branch distance was also proposed. In our work, the notion of boundary distance is introduced to utilize the semantic structure of regions. Our idea is that test data near the boundaries are more likely to detect bugs. In this paper, we extend this basic idea and propose a method that can efficiently detect failures by automatically generating test cases near the boundaries of regions.

Our approach is essentially a Search-based Software Engineering (SBSE) approach [33], [34]. SBSE seeks to reformulate software engineering problems as "search problems" using a variety of techniques from the metaheuristic search, operations research, to evolutionary computation paradigms. SBSE has been applied to solve many different software engineering problems such as optimizing design decisions[35]. Search-based techniques have been also used to generate a test input to drive the execution along a chosen path or branch [10], [22], [11], [33], [12], [19], [13]. Our work applies the essential idea of SBSE to search for test data along the region boundaries.

Our work can be applied to regression testing. In practice, regression testing is performed to ensure that new changes to the code do not introduce new faults. As it is time-consuming to run the aggregated tests, many test selection and prioritization techniques [20], [36] have been proposed to reduce the number of tests used in regression testing. Some of these techniques also apply search-based algorithms

(such as greedy algorithms[20] and genetic algorithms[37]), to optimize test cases with respect to certain coverage criteria (such as statement coverage and branch coverage). We apply an evolutionary algorithm called memetic algorithm to generate test suites for effective regression testing.

VIII. Conclusion

In this paper, we propose a search-based approach for automatically generating test cases that are close to the boundaries of regions. A prototype tool is implemented and experimental results have shown that our method can generate better test suites for failure detection. The results also show that our approach outperforms the ART-based and coverage based test case generation methods. Our approach can reduce the manual work required by the existing boundary value analysis techniques and can be put in to practice in the future.

Acknowledgments

This research is sponsored in part by NSFC Program (No. 91218302, 61402248), and MIIT IT funds (Research and application of TCN key technologies) of China.

References

[1] B. Beizer, "Software testing techniques. 1990," *New York, ISBN: 0-442-20672-0*.

[2] G. Renfer, "Automatic program testing," in *Proceedings of the 3rd Conference of the Computing and Data Processing Society of Canada*, 1962.

[3] R. Hamlet, *Random Testing*. Wiley Online Library, 2002.

[4] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for java JIT compiler test system," in *Quality Software, 2003. Proceedings. Third International Conference on*. IEEE, 2003, pp. 20–23.

[5] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th USENIX Windows System Symposium*, 2000, pp. 59–68.

[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, pp. 1001–1010, 2004.

[7] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*. Springer, 2005, pp. 320–329.

[8] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[9] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 265–275.

[10] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[11] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 12, pp. 1085–1110, 2001.

[12] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 226–247, 2010.

[13] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[14] B. Legeard, F. Peureux, and M. Utting, "Automated boundary testing from z and b," in *FME 2002: Formal MethodsGetting IT Right*. Springer, 2002, pp. 21–40.

[15] S. C. Reid, "An empirical analysis of equivalence partitioning, boundary value analysis and random testing," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*. IEEE, 1997, pp. 64–73.

[16] P. Moscato, "On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms," *Caltech concurrent computation program, C3P Report*, vol. 826, p. 1989, 1989.

[17] X. Chen, Y.-S. Ong, M.-H. Lim, and K. C. Tan, "A multi-facet survey on memetic computation," *Evolutionary Computation, IEEE Transactions on*, vol. 15, no. 5, pp. 591–607, 2011.

[18] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[19] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1098–1105.

[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)*, vol. 2, no. 3, pp. 270–285, 1993.

[21] P. Moscato, C. Cotta, and A. Mendes, "Memetic algorithms," in *New optimization techniques in engineering*. Springer, 2004, pp. 53–85.

[22] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[23] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 16, pp. 870–879, 1990.

[24] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[25] B. Jiang, Z. Zhang, W. Chan, and T. Tse, "Adaptive random test case prioritization," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 233–244.

[26] G. Fraser and A. Arcuri, "Evosuite at the sbst 2013 tool competition." in *ICST Workshops*, 2013, pp. 406–409.

[27] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 71–80.

[28] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[29] K. Jamrozik, G. Fraser, N. Tillman, and J. De Halleux, "Generating test suites with augmented dynamic symbolic execution," in *Tests and Proofs*. Springer, 2013, pp. 152–167.

[30] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal voronoi tessellations-a new approach to random testing," *Software Engineering, IEEE Transactions on*, vol. 39, no. 2, pp. 163–183, 2013.

[31] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 365–381.

[32] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 123–133.

[33] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[34] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 342–357.

[35] W. Heaven and E. Letier, "Simulating and optimising design decisions in quantitative goal models," in *Requirements Engineering Conference (RE), 2011 19th IEEE International*. IEEE, 2011, pp. 79–88.

[36] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE, 2004, pp. 106–115.

[37] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 1, pp. 19–34, 1999.