Software Quality Journal
https://doi.org/10.1007/s11219-023-09659-9

RESEARCH

Optimal test case generation for boundary value analysis

Xiujing Guo[1] · Hiroyuki Okamura[1] · Tadashi Dohi[1]

Accepted: 21 December 2023
© The Author(s) 2024

Abstract
Boundary value analysis (BVA) is a common technique in software testing that uses input values that lie at the boundaries where significant changes in behavior are expected. This approach is widely recognized and used as a natural and effective strategy for testing software. Test coverage is one of the criteria to measure how much the software execution paths are covered by the set of test cases. This paper focuses on evaluating test coverage with respect to BVA by defining a metric called boundary coverage distance (BCD). The BCD metric measures the extent to which a test set covers the boundaries. In addition, based on BCD, we consider the optimal test input generation to minimize BCD under the random testing scheme. We propose three algorithms, each representing a different test input generation strategy, and evaluate their fault detection capabilities through experimental validation. The results indicate that the BCD-based approach has the potential to generate boundary values and improve the effectiveness of software testing.

Keywords Software testing · Boundary value analysis · Test input generation · Boundary coverage · Random testing

1 Introduction

Software testing is the execution of software systems for the purpose of detecting defects, which is of great importance for verifying the reliability of software systems. The main activity of software testing is to generate test cases consisting of test inputs and their expected behavior (outputs) of the software. In a testing phase, the test cases are executed, i.e., we obtain the actual outputs of the implemented software with the test inputs and compare them with the expected ones, so that the behavior of the software is validated. To

Hiroyuki Okamura and Tadashi Dohi contributed equally to this work.

✉ Xiujing Guo
guoxiujing6@gmail.com

Hiroyuki Okamura
okamu@hiroshima-u.ac.jp

Tadashi Dohi
dohi@hiroshima-u.ac.jp

[1] Graduate School of Advanced Science and Engineering, Hiroshima University, 1 Chome-2 Kagamiyama, Hiroshima 739-8511, Japan

ensure the reliability of the software, it is important to design a set of test cases that can find all the inherent bugs in the software.

There are several approaches to test case design. Test coverage plays the most important role in test case design. Test coverage is one of the criteria to measure how much the software execution paths are covered by the set of test cases. For example, statement coverage, called C0 coverage, is defined as the percentage of program statements executed by the test cases overall program statements. Test coverage criteria (Zhu et al., 1997), such as statement coverage (SC), path coverage (PC), branch coverage (BC), decision coverage (DC), decision/condition coverage (D/CC), and modified condition/decision coverage (MC/DC) (Chilenski & Miller, 1994), are commonly used in software testing. And the test cases are designed to achieve a test coverage that reaches a predefined level. On the other hand, boundary value analysis (BVA) is also commonly used for the design of effective test cases. The BVA is to identify the boundaries of program paths, i.e., the values that change the program paths with a small amount of input changes, and to design the test cases using the boundary values. Since it is empirically known that software bugs exist around the boundaries (Dobslaw et al., 2020), the test cases with BVA are those capable of finding bugs with high probability.

Traditionally, BVA follows a manual approach where a human developer analyzes a specification to identify partitions and their associated boundary values. Test cases are then written to ensure correct behavior at these boundaries (Reid, 1997). Several authors have contributed to the development and improvement of boundary value analysis and testing since its initial introduction and extension by White and Cohen (1980) and Clarke et al. (1982). Jeng and Forgacs (1999) proposed a semi-automatic method that mixed dynamic search and algebraic manipulation of boundary conditions to more efficiently generate test data for boundary value testing (BVT). Zhao et al. (2010) focused on the handling of string inputs and proposed an innovative approach to automatically generate test points to identify problems at the boundaries of code with string predicates. In addition, Ali et al. (2016) extended their search-based test data generation method to model-based testing, using a solver to automatically generate boundary values based on a set of heuristics. More recently, Feldt and Dobslaw (2019) applied the idea of derivatives in mathematical parlance to detect the maximum "change" region by combining the input and output distances, known as the detection boundary. This method uses program derivatives as a fitness function in search-based software testing for automated BVA. BVA was originally used in black-box testing. Recently, researchers have proposed several approaches to apply BVA in white-box test generation (Pandita et al., 2010; Jamrozik et al., 2013; Zhang et al., 2015; Guo et al., 2023). Zhang et al. (2015) proposed a new definition for boundary conditions based on mutation. The authors use a constraint to constrain the input to the boundary and use a constraint solver to solve it. The study in Guo et al. (2023) presents a test case generation approach that focuses on BVA in white-box testing. It combines a machine learning approach to bounds detection and MCMC (Markov chain Monte Carlo) to generate test inputs.

BVA is a systematic testing technique used in software testing to design test cases that focus on the boundaries of input domains. The primary goal of BVA is to identify potential defects or errors that may occur at or near the edges or limits of the input space. To make BVA more effective, proposing a boundary coverage metric is important. By defining boundary coverage metrics, we can measure how thoroughly we test these critical areas, evaluate the quality of the test suite, and ensure we do not miss critical test scenarios. Moreover, BVA is particularly useful for uncovering off-by-one errors, boundary-related exceptions, and other issues that often escape less focused testing. A boundary coverage metric allows to quantify the coverage of these high-risk areas, reducing the chances of releasing software with critical defects. At the same time, testing can be time-consuming and resource-intensive. By establishing boundary coverage metrics, we can prioritize testing efforts. Focusing on achieving full coverage of the boundary during testing while conducting less exhaustive testing in non-boundary areas helps optimize resource allocation and testing efficiency. In summary, proposing a boundary coverage metric for boundary value analysis in software testing is essential for enhancing the precision, efficiency, and effectiveness of the testing process.

Evaluating the test coverage intending BVA is difficult since the boundaries are defined as continuous values. Li and Miao (2012) propose a series of model-based logic boundary coverage criteria, combining the boundary coverage criteria and the logic coverage criteria. Kosmatov et al. (2004) focus on the development of boundary coverage criteria for test generation from formal models and formalize the boundary coverage criteria, the rational for the (BZ-TT) method, and toolset. Utilizing the aforementioned techniques, it is necessary to have the specification that clearly states the boundary formally. However, many software development projects lack formal specifications in their development process. Identifying the exact boundaries within the input space of a software system can be complex.

Software applications often operate within multidimensional input domains. These domains might involve intricate data structures and interactions between various inputs. The complexity of these structures and interactions complicates the identification of boundaries, as they may not always align with conventional notions of limits or edges. Software behavior is not always static. Dynamic behavior can lead to shifting or evolving boundaries, making it challenging to define and cover them adequately. The dynamic nature of software adds an extra layer of complexity in identifying and testing boundaries effectively. Complex data structures, interactions between inputs, and dynamic software behavior can complicate boundary identification.

Additionally, creating a comprehensive set of test cases to cover all boundaries can be resource-intensive. As the number of input variables or dimensions increases, the number of potential boundary combinations can grow exponentially. This results in a combinatorial explosion of test cases, requiring significant time and resources to create and execute. Achieving complete boundary coverage might not always be the most efficient or cost-effective strategy. Finding a balance between comprehensive testing and resource constraints is a challenge.

Overall, the lack of formal specifications, navigating the complexities of software behavior, multidimensional input spaces, and the trade-offs between thorough testing and resource constraints pose significant challenges in proposing effective boundary coverage metrics and associated test generation algorithms. Finding solutions involves a balance between the ideal of comprehensive testing and the practicalities of resource management in software testing endeavors.

In this paper, we attempt to define alternative measures, called boundary coverage distance (BCD). BCD introduces a metric that evaluates test inputs' quality concerning boundary coverage in BVA. It focuses on distance measurements and considerations of lower and upper limits to gauge test inputs' proximity to the boundaries, ensuring comprehensive boundary coverage for more reliable software testing. In addition, based on BCD, we consider the optimal test input generation to minimize BCD under the random testing scheme. This method addresses the resource-intensive nature of generating exhaustive boundary tests by efficiently selecting critical test cases that contribute the most to the BCD metric. In summary, the contributions of this paper include (i) the definition of the boundary coverage distance to measure how much the test cases cover the boundary and (ii) developing test generation algorithms based on BCD.

This paper is organized as follows. In Section 2, we introduce concepts associated with BVA. Section 3 is a definition of boundary coverage distance (BCD). In Section 4, we illustrate our method for generating test input. Section 5 is devoted to experiences with real programs. Finally, in Section 6, we summarize this paper and discuss future work.

2 Boundary value analysis

Boundary value analysis (BVA) is one of the most popular approaches to generating test inputs. It is empirically known that the probability of introducing bugs is higher around the program path changes, i.e., the boundary. BVA finds the boundary from specifications or programs and generates test inputs around the boundary.

Consider the formal definition of boundary. Let f and I be a software under test and an input domain of software, respectively. The output of software is defined as $O := \{y \mid y = f(x),\ x \in I\}$. Suppose that the output of software is divided into several categories. Let $O_1, \ldots, O_m$ be m-categorized outputs of software where $O = \cup_{i=1}^{m} O_i$ and $O_i \cap O_j = \phi$ for all $i \neq j$. The outputs in a category are considered to be equal in a sense. Then, the input domain can be divided by the equivalent partitions, i.e.,

$$I_i := \{x \in I \mid y = f(x),\ y \in O_i\}. \tag{1}$$

Intuitively, the boundary is the input of software that crosses two equivalent partitions.

To define the boundary, we consider the functions that change the input of software. Let g be the bijection function from the input domain to the input domain: $g : I \to I$. Let G be a set of the functions with the following properties:

- For any $g \in G$, $g^{-1} \in G$ where g^{-1} is the inverse function of g.
- There exists at least one composite function of G from any $x \in I$ to any $y \in I$.

A function in G is regarded as a minimal operation that changes inputs. The first property corresponds to the existence of an inverse operation. The second one means that any input in I can be generated by a chain of the operations. Therefore, the boundary of equivalent partitions is given by

$$B_i := \{x \in I_i \mid \text{There exists } g \in G \text{ such that } y = f(g(x)),\ y \notin O_i\}. \tag{2}$$

Consider the software for judging whether the English examination passed. The English examination has two kinds of scores: listening and reading. Each of listening and reading is scored out of 100. The conditions to get the credit are as follows: (i) both listening and reading scores exceed 50, and (ii) the total of listening and reading scores exceeds 120. The input of software is a pair of the listening and reading scores for a student, and the output is one of (1) “the input is invalid,” (2) “the student gets the credit,” and (3) “the student does not get the credit.”

For such software, we consider four functions to represent four operations changing inputs: (g_1) increasing the listening score by one, (g_2) decreasing the listening score by one, (g_3) increasing the reading score by one, and (g_4) decreasing the reading score by one. In function set $G = \{g_1, g_2, g_3, g_4\}$, $(g_1)^{-1} = g_2$. The boundary sets are

$$\begin{aligned} B_1 := \{&(-1,0), \ldots, (-1,100), \\ &(101,0), \ldots, (101,100), \\ &(0,-1), \ldots, (100,-1), \\ &(0,101), \ldots, (100,101)\}, \end{aligned} \tag{3}$$

$$\begin{aligned} B_2 := \{&(100,50), \ldots, (70,50), \\ &(50,100), \ldots, (50,70), \\ &(100,51), \ldots, (100,100), \\ &(51,100), \ldots, (99,100), \\ &(69,51), (68,52), \ldots, (52,68), (51,69)\}, \end{aligned} \tag{4}$$

$$\begin{aligned} B_3 := \{&(0,0), (0,1), \ldots, (0,100), \\ &(100,0), \ldots, (100,49), \\ &(1,0), \ldots, (100,0), \\ &(0,100), \ldots, (49,100), \\ &(99,49), \ldots, (70,49), \\ &(49,99), \ldots, (49,70), \\ &(69,50), (68,51), \ldots, (51,68), (50,69)\}, \end{aligned} \tag{5}$$

where (x, y) means the scores for listening and reading, respectively. Figure 1 shows the input domain and the boundaries of this software. The x-axis and y-axis correspond to the listening and reading scores, respectively. There are three equivalent partitions, I_1, I_2, and I_3, and the boundaries are located on the edges of equivalent partitions. It should be noted that the input $(-1, -1)$ is not the boundary, because the input $(-1, -1)$ cannot be the input belonging to I_2 or I_3 even if we apply any operations.

Remark 1 Any operations can be used if it holds the two properties. For example, we can add the operations: (v) increasing both the listening and reading scores by one and (vi)

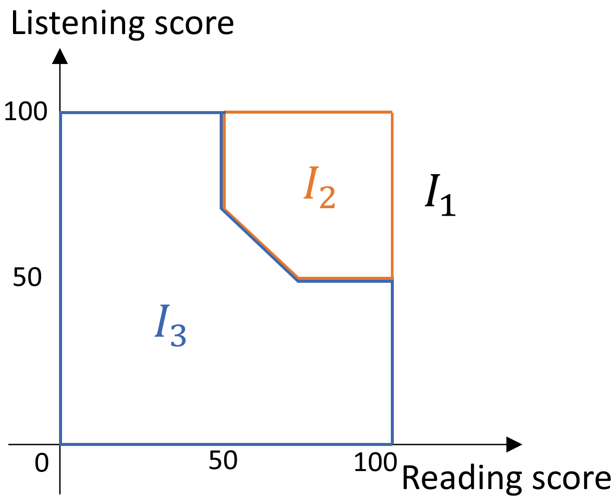


Fig. 1 The input domain and the boundaries of the `English` software

decreasing both the listening and reading scores by one. In this case, the input $(-1, -1)$ becomes the boundary because (0, 0) is generated by the operation (v) from the input $(-1, -1)$. In other words, the boundaries depend on the definitions of operations in the formal definition.

Remark 2 The outputs of software can be defined arbitrarily. If we focus on the execution paths on the program, the inputs (49, 50) and (50, 49) can be the different execution paths. In this example, we define the equivalent partitions only from the result of the judgment for the English examination by ignoring the execution paths, which is like a black-box approach. On the other hand, if the output of software, the behavior of software, is defined by the execution paths, as in a white-box approach, the equivalent partitions are changed. That is, the boundaries are also changed.

3 Boundary coverage distance

3.1 Definition

In this paper, we propose a metric to evaluate the quality of test inputs from the perspective of BVA, called boundary coverage distance (BCD). First, we define the boundary coverage.

Definition (boundary coverage) Boundary coverage is the percentage of boundary values that are executed by a test suite.

Ideally, the boundary coverage is also achieved by a test suite to ensure highly reliable software. However, it is not easy to cover all the boundary values, because there are a huge number of boundary values and sometimes the number of boundary values becomes infinite. For example, even in the example of English examination, although it is a simple program, there are 963 boundary values.

Instead of the boundary coverage, we consider the distance from a given test suite to the test suite that achieves the boundary coverage. Let $d(x, y)$ be the function that returns the distance from x to y. Based on the formal definition of boundary, the distance is defined by the number of minimum operations from x to y, i.e.,

$$d(x, y) = \min_n \{n \geq 0 \mid y = g_1(g_2(\cdots g_n(x))),\ g_1, \ldots, g_n \in G\}. \tag{6}$$

For example, in English program mentioned in Section 2, the number of minimum operations from input $x = (-1, -1)$ to input $y = (0, 0)$ is 2.

Suppose that T_i is the test input (test suite) belonging to the equivalent partition I_i. The basic idea is the expansion of test inputs. The expansion means that each test input covers the test inputs that are within a given distance. The distance from T_i to I_i is defined as the minimum expansion distance for the test inputs in T_i until they cover all the boundary values B_i.

$$d(T_i, B_i) = \max_{y \in B_i} \min_{x \in T_i} d(x, y), \tag{7}$$

where $d(T_i, B_i) = \infty$ when T_i is empty.

Finally, the distance from the suite $S = \cup_{i=1}^m T_i$ to $B = \cup_{i=1}^m B_i$ is given by

$$d(T, B) = \max_{i=1,\ldots,m} d(T_i, B_i). \tag{8}$$

We call this distance the boundary coverage distance (BCD). If BCD is small, the test inputs are placed close to the boundaries.

3.2 Computation of BCD

To compute BCD, we need to obtain all the boundary values. As mentioned before, it is not easy to get all the boundary values. Here, we consider the lower and upper limits of BCD. Let $\underline{B}_i$ be a set of test inputs which are placed on B_i, i.e., $\underline{B}_i \subseteq B$. In this paper, $\underline{B}_i$ is called the boundary points on I_i. From Eq. (21), it is clear that

$$d(T_i, \underline{B}_i) \leq d(T_i, B_i). \tag{9}$$

Therefore, we have the lower limit of BCD as follows.

$$\underline{\mathrm{BCD}}(T) = \max_{i=1,\dots,m} d(T_i, \underline{B}_i). \tag{10}$$

Next, we consider the upper limit of BCD. This paper focuses on components of the boundary, called the boundary components. For example, if the boundary is represented by a line segment, the boundary components are line segments that are shorter than the original boundary. If the boundary is a plane, the boundary components are triangles that cover the plane. We assume that each boundary component is defined by a set of points. A line segment is represented by two start and end points. In the case of a triangle, it is defined by a set of three points. In general, $\overline{B}_i := \{P_1, \dots, P_k\}$ is a set of boundary components $P_1, \dots, P_k$ that cover the boundary B_i where each boundary component consists of a set of points $P_i := \{x_1, \dots, x_h\}$. The distance from a point x to a boundary component P_i is given by

$$d(x, P_i) = \max_{i=1,\dots,h} d(x, x_i). \tag{11}$$

This gives us the distance from T_i to $\overline{B}_i$:

$$d(T_i, \overline{B}_i) = \max_{p \in \overline{B}_i} \min_{x \in T_i} d(x, p). \tag{12}$$

Since Eq. (11) takes the maximum of points on the boundary, the distance of the boundary components is greater than the exact distance, i.e.,

$$d(T_i, B_i) \leq d(T_i, \overline{B}_i). \tag{13}$$

The upper limit of BCD becomes

$$\overline{\mathrm{BCD}}(T) = \max_{i=1,\dots,m} d(T_i, \overline{B}_i). \tag{14}$$

Since both $\underline{B}_i$ and $\overline{B}_i$ consist of the points that are the subset of B_i, they can be computed without extracting all the boundary values.

For instance, we select the following subsets of B_1, B_2, and B_3 in the example:

$$\begin{aligned} \underline{B}_1 := \{&(-1,0), (-1,100), (101,0), (101,100), \\ &(0,-1), (100,-1), (0,101), (100,101)\}, \end{aligned} \tag{15}$$

$$\underline{B}_2 := \{(100,50),(70,50),(50,100),(50,70),(100,100)\}, \tag{16}$$

$$\underline{B}_3 := \{(0,0),(0,100),(100,0),(49,100),(100,49),(49,70),(70,49)\}. \tag{17}$$

$$\begin{aligned}\overline{B}_1 := \{&\{(-1,0),(-1,100)\},\{(101,0),(101,100)\},\\ &\{(0,-1),(100,-1)\},\{(0,101),(100,101)\}\},\end{aligned} \tag{18}$$

$$\begin{aligned}\overline{B}_2 := \{&\{(100,50),(70,50)\},\{(50,100),(50,70)\},\{(100,50),(100,100)\},\\ &\{(50,100),(100,100)\},\{(70,50),(50,70)\}\},\end{aligned} \tag{19}$$

$$\begin{aligned}\overline{B}_3 := \{&\{(0,0),(0,100)\},\{(0,0),(100,0)\},\\ &\{(0,100),(49,100)\},\{(100,0),(100,49)\},\\ &\{(49,100),(49,70)\},\{(100,49),(70,49)\},\\ &\{(49,70),(70,49)\}\}.\end{aligned} \tag{20}$$

It should be noted that $\underline{B}_i = \cup_{c\in\overline{B}_i} c$. Figure 2 shows the boundary points, and Fig. 3 shows the boundary components (segments).

4 Test input generation

In this section, we consider the test input generation that minimizes BCD. Suppose that the boundary points $\underline{B}_i$ and the boundary components $\overline{B}_i$ are given. In this situation, we obtain additional n test inputs minimizing BCD. Ideally, when the BCD is reduced to a minimum value, the n test data to be optimized move to the boundary point or to the center of the boundary component, since a large number of test cases cause a large amount of cost on software testing. In this study, we first empirically analyze all possible boundary values as boundary points. Then, the boundary region to be tested is selected by minimizing the value of the BCD of the n test data.

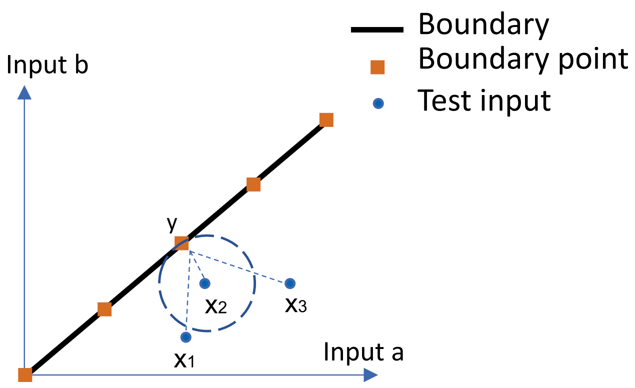


Fig. 2 An example for covering a boundary point (compared with x_1 and x_3, the test input x_2 can cover the boundary point y with the minimum distance $d(x_2, y)$)

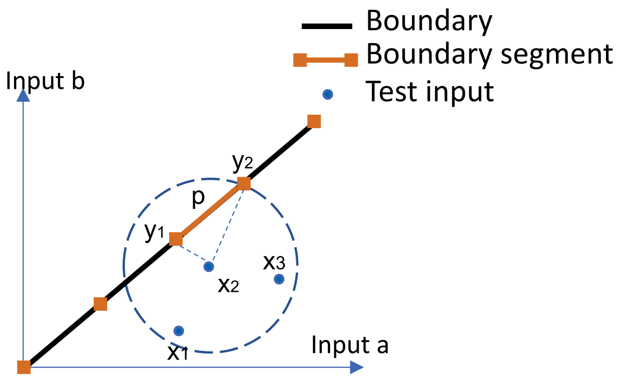


Fig. 3 An example for covering a boundary component (segment) (the test input x_2 can cover the boundary segment p with a distance of $d(x_2, y_2)$)

We propose three test input generation algorithms that are very similar to the MCMC (Markov chain Monte Carlo) method (Chib & Greenberg, 1995).

The optimization process is shown in Algorithm 1, Algorithm 2, and Algorithm 3, respectively.

Algorithm 1 first randomly generates n test inputs from the input domain as an initial test set. Then, the initial test set is optimized by reducing the BCD value (lines 2–10). During the optimization process, the algorithm 1 first calculates the BCD of the initial test set. Then, it randomly selects a test input t from the initial test set and generates a new candidate t' according to the proposal distribution, provided that t is given $t' \sim Q(t';t)$. If replacing t with the candidate t' can reduce the value of BCD, the candidate t' is accepted and replaces t in the initial data set; otherwise, the candidate t' is rejected. After the optimization process performs a fixed number of iterations, the initial test data is moved to the boundary. In the Algorithm 1, BCD can be computed as $\underline{BCD}$ or $\overline{BCD}$ mentioned in Section 3.2, or even as the mean of $\underline{BCD}$ and $\overline{BCD}$, denoted as $BCD\_mean = mean(\underline{BCD} + \overline{BCD})$.

Algorithm 1 only accepts candidates that can reduce the maximum distance among the minimum distances between the boundary point (or boundary line segment) and the test input. This means that the optimization goal of each iteration is only a boundary point or a boundary component, and the optimization process is slow.

Algorithm 1 An algorithm to generate boundary test inputs by reducing BCD

```
1:  $Initial\_test\_set \leftarrow$ {randomly generate n test data from the input domain}
2:  while $j \leq Iter$ do
3:      $BCD \leftarrow BCD(Initial\_test\_set)$
4:      $t \leftarrow randomly\ select\ a\ test\ data\ from\ Initial\_test\_set$
5:      $t' \leftarrow Q(t';t)$
6:      $Candidate\_test\_set \leftarrow$ {Replace the t in the Initial_test_set with the candidate
        t'}
7:      $BCD' \leftarrow BCD(candidate\_test\_set)$
8:      if $BCD' < BCD$ then
9:          $Initial\_test\_set \leftarrow candidate\_test\_set$
10:     end if
11:     $j \leftarrow j + 1$
12: end while
```

Algorithm 2 An algorithm that considers each boundary point or each boundary component

```
1: $Initial\_test\_set \leftarrow$ {randomly generate n test data from the input domain}
2: while $j \leq Iter$ do
3:     $Calculate\ d(T_i, y)$
4:     $t \leftarrow randomly\ select\ a\ test\ data\ from\ Initial\_test\_set$
5:     $t' \leftarrow Q(t'; t)$
6:     $Candidate\_test\_set \leftarrow$ {Replace the t in the Initial_test_set with the candidate
       t'}
7:     $Calculate\ d'(T_i, y)$
8:     if $b\_decrease > 0$ and $b\_increase = 0$ then
9:         $Initial\_test\_set \leftarrow candidate\_test\_set$
10:    end if
11:    $j \leftarrow j + 1$
12: end while
```

With this in mind, we judged whether to accept the candidate by directly comparing the coverage distance $d(T_i, y)$ for each boundary point or boundary component.

$$d(T_i, y) = \min_{x \in T_i} d(x_i, y) \quad for \;\; y \in B_i. \tag{21}$$

Let *b_decrease* be the number of boundary points or boundary components whose coverage distance is decreased by the candidate test set. Similarly, *b_increase* denotes the number of boundary points or boundary components whose coverage distance is increased by the candidate test set. We then get two algorithms for generating boundary test inputs, shown in Algorithm 2 and Algorithm 3 respectively. Algorithm 2 accepts a candidate when the candidate can reduce the coverage distance of one or more boundary points (or one or more boundary components) without increasing the coverage distance of any others. However, in Algorithm 3, even if the candidate increases the coverage distance of some boundary points (or some boundary components), it may be accepted with a certain probability. In Algorithms 2 and 3, *b_decrease* and *b_increase* can be calculated with $\overline{B_i}$ or $\underline{B_i}$. Combining the three algorithms and the three BCD calculation methods, seven methods can be obtained, as shown in Table 1.

Algorithm 3 Accept with probability

```
1: $Initial\_test\_set \leftarrow$ {randomly generate n test data from the input domain}
2: while $j \leq Iter$ do
3:     $Calculate\ d(T_i, y)$
4:     $t \leftarrow randomly\ select\ a\ test\ data\ from\ Initial\_test\_set$
5:     $t' \leftarrow Q(t'; t)$
6:     $Candidate\_test\_set \leftarrow$ {Replace the t in the Initial_test_dataset with the
       candidate t'}
7:     $Calculate\ d'(T_i, y)$
8:     $Generate\ a\ uniform\ random\ number\ U$
9:     if $\frac{b\_decrease}{b\_decrease + b\_increase} > U$ then
10:        $Initial\_test\_set \leftarrow candidate\_test\_set$
11:    end if
12:    $j \leftarrow j + 1$
13: end while
```

Figures 4 and 5 demonstrate the ideal solutions generated by our algorithms when applying the $\underline{BCD}(\underline{B_i})$ and $\overline{BCD}(\overline{B_i})$ criteria, respectively. For the purpose of illustration, we assume a scenario where five boundary points are evenly spaced along a linear boundary, with a unit distance between each pair of adjacent points. The ideal solution varies based on the number of boundary points and test inputs. Figure 4a represents a scenario where the number of test inputs is fewer than the boundary points. Here, one test input is positioned at the central boundary point, while the remaining test inputs are placed midway between adjacent boundary points. This arrangement results in an optimal $\underline{BCD}$ value of 0.5. Conversely, as shown in Fig. 4b, when the number of test inputs exceeds the boundary points, each test input aligns with a boundary point, achieving an optimal $\underline{BCD}$ value of 0. Figure 5 illustrates the ideal solutions under the $\overline{BCD}$ criterion, which shows a different pattern compared to the $\underline{BCD}$ criterion. If the number of boundary points is greater than the test inputs, some test cases are likely to align with the boundary points themselves. However, when there are more test inputs than boundary points, all test inputs tend to be the middle of the boundary segments. The average outcomes of the $\underline{BCD}$ and $\overline{BCD}$ methods demonstrate these intermediate tendencies. These scenarios represent ideal states in a highly simplified boundary

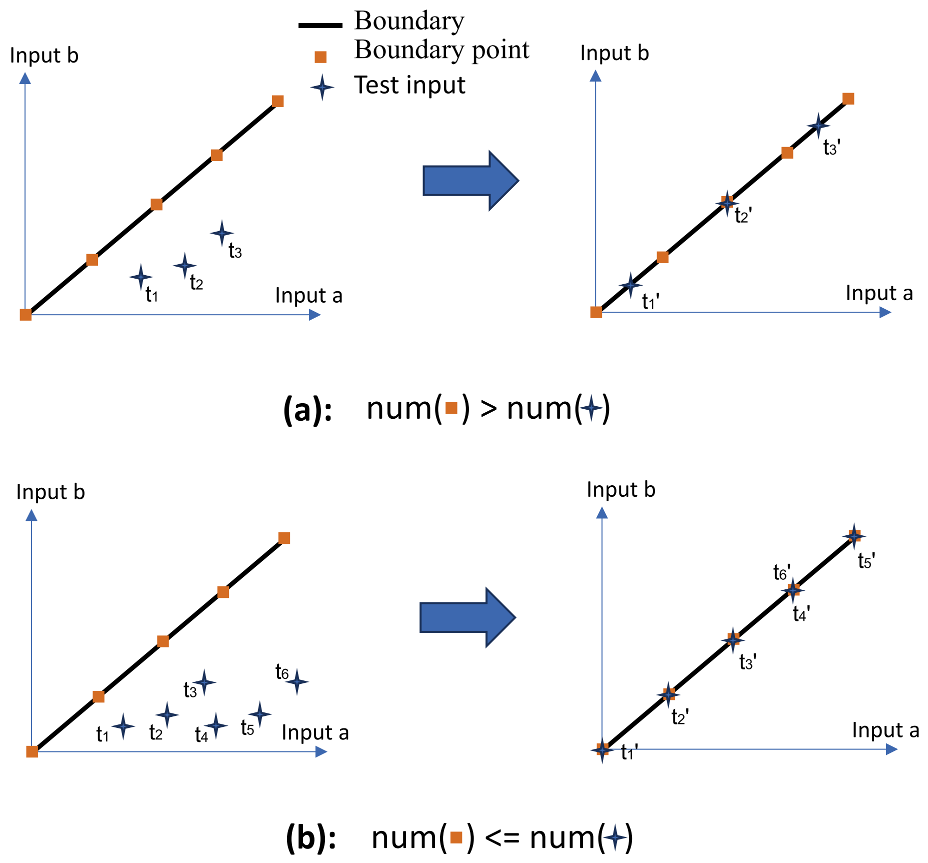


Fig. 4 The ideal solutions generated by our algorithms when applying the $\underline{BCD}(\underline{B_i})$ criteria

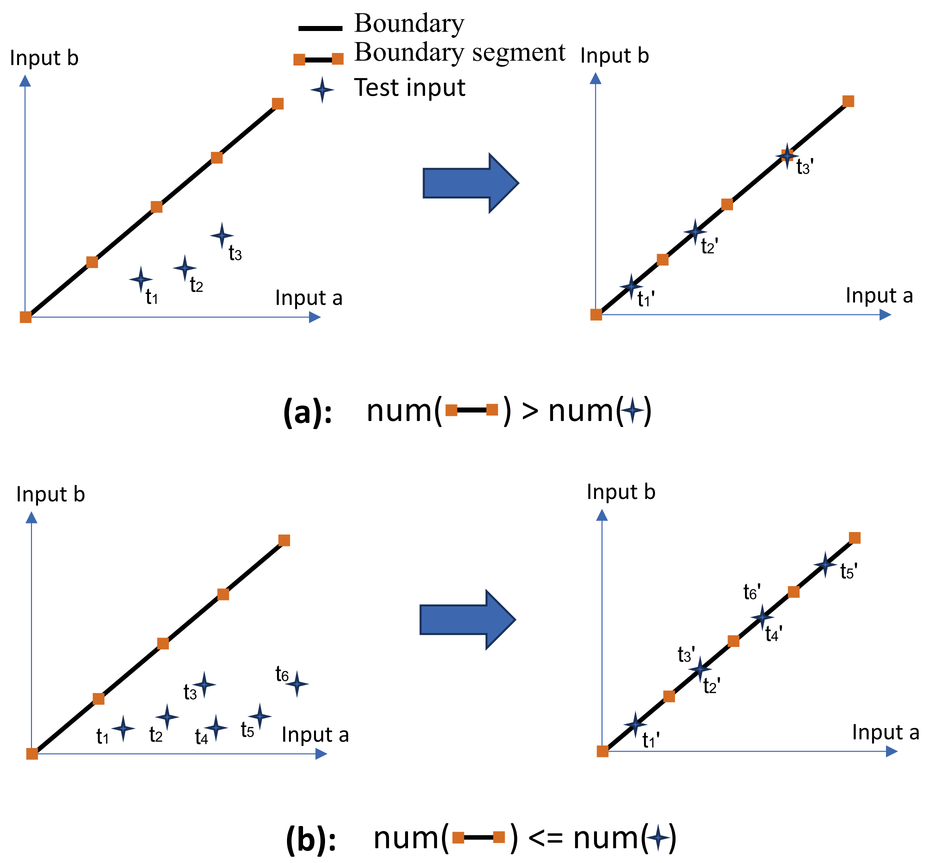


Fig. 5 The ideal solutions generated by our algorithms when applying the $\overline{BCD}(\overline{B_i})$ criteria

context. It is important to note that in practical applications with more complex boundaries, the arrangement of test cases is likely to be more intricate.

5 Experiment

This section presents experiments to investigate the fault detection capabilities of the three algorithms of the BCD approach and compare them with RT, ART, and concolic testing. We conducted experiments to generate test inputs for the previously mentioned English examination program and four real programs.

Table 1 Combining the three algorithms with the three methods of BCD calculation results in seven approaches

Methods \ BCD / Algorithms	$\underline{BCD}(\underline{B_i})$	$\overline{BCD}(\overline{B_i})$	BCD_mean
Algorithm1	A1-$\underline{BCD}$	A1-$\overline{BCD}$	A1-BCD_mean
Algorithm2	A2-$\underline{B_i}$	A2-$\overline{B_i}$	—
Algorithm3	A3-$\underline{B_i}$	A3-$\overline{B_i}$	—

5.1 Fault detection ability

To investigate the effectiveness of test sets in detecting faults, we employ mutation testing. This approach involves modifying specific statements in the source code, known as mutations, to evaluate whether the existing test cases are capable of identifying these code errors. In our experiment, we deliberately introduce faults into the program by seeding them. Each seeded fault creates a faulty version of the program. For every test set generated during the experiment, we execute the entire set on each faulty version and tally the number of mutations that are detected or "killed." We calculate the kill rate using Eq. (22).

$$kill\_rate = \frac{the\ number\ of\ killed\ mutations}{total\ number\ of\ mutations} \tag{22}$$

The program under test is subjected to five types of injected faults. One category is the off-by-one bugs (OBOB), which occur when a computation process utilizes an incorrect value that is either one more or one less than the intended value. Boundary faults commonly fall into this category (Zhang et al., 2015). The remaining faults comprise four commonly used mutation operators (Jia & Harman, 2010): relational operator replacement (ROR), arithmetic operator replacement (AOR), scalar variable replacement (SVR), and logical operator replacement (LOR). To determine if a mutation is killed, we analyze the execution path. Specifically, if at least one test input exhibits a different execution path from the correct version, the mutation is killed.

In this experiment, the execution path is defined as the combination of the execution state of each branch in the program under a given input. Here, we refer to each atomic Boolean expression in the path condition as a predicate. Predicates can include Boolean variables and comparison predicates ($>, >=, <, <=, =, \neq$) and should not contain any Boolean operators such as $\wedge$, $\vee$, and $\neg$ (Zhang et al., 2015). Each comparison predicate consists of two branches, and each branch has three states, denoted as "`1, 0, -1`." For example, consider the comparison predicate ($b < 0$), which has two branches: ($b < 0$) and ($b >= 0$).

We assign different states to a branch based on its execution and whether it is taken. If the branch is executed and taken (satisfied) one or more times, we label the state of this branch as "`1`." If the branch is executed but not taken, we label its state as "`0`." If the branch is not executed at all, we label its state as "`-1`." For conditional branches within a program loop, regardless of the number of iterations, we mark the state of the branch that triggers the loop as "`1`." This means that when a branch is taken multiple times, the actual execution path differs based on the number of times it is taken. However, to prevent path explosion from causing too many equivalence partitions, we disregard the number of times the branch is taken and simply consider it as "`1`" if it is taken.

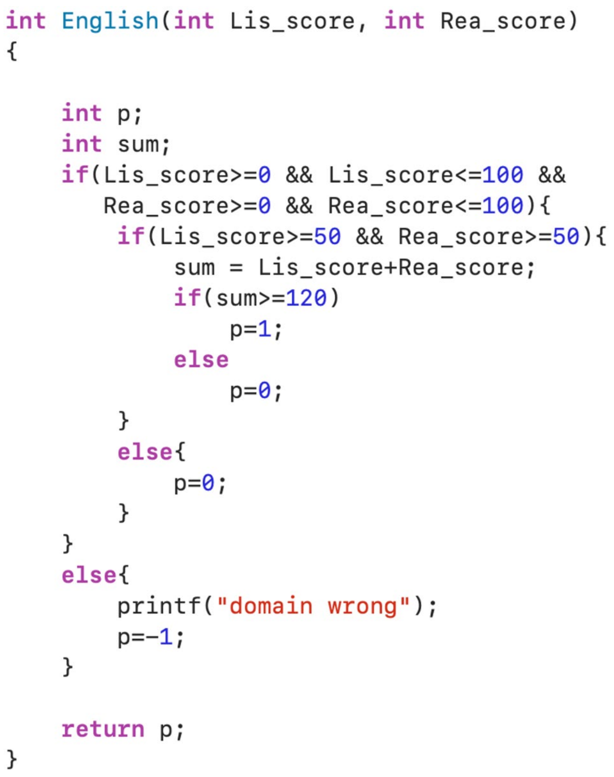
We annotate the source code to add instrumentation and use the Gcov tool to extract the branch execution information. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html is a source code coverage analysis tool that works in conjunction with GCC to implement statement coverage and branch coverage testing of C/C++ files. By adding instrumentation, Gcov accurately records the number of executions for each statement in the program. In our work, we use Gcov to write the branch execution frequency to an output file during program testing. We then extract the branch information from the file to obtain the execution path.

For example, Fig. 6 shows a simple C program we mentioned earlier to determine whether the English examination was passed. Figure 7 demonstrates the result of Gcov on running the `En_testing.c.gcov` program with the input values $a = 45$ and $b = 65$. We convert “`taken a (a>0)`” to “`1`” to indicate that the branch is taken at least once, convert “`taken 0`” to “`0`” to indicate that the branch is not taken, and convert “`never executed`” to “`-1`” to indicate that the branch is not executed. The execution path of the program `En_testing.c` with the input values $a = 55$ and $b = 45$ is the combination of all branch executions: `1,0,1,0,1,0,1,0,1,0,0,1,-1,-1`.

5.2 RT and ART

In the RT approach, we randomly generate a test set consisting of n test inputs and execute each mutated program with this test set to study the fault detection ability.

The execution of ART follows the algorithm outlined in Chen et al. (2004). In the traditional ART algorithm, the executed set is incrementally updated by selecting elements from the candidate set until a failure is uncovered. However, to ensure comparability between the experimental results of ART and the BCD-based method, we modified the experimental stopping condition of ART to incrementally generate n test cases. For each mutation, the test case generation process stops if the injected fault is detected during the generation of the i-th ($i \leq n$) test input, and the generated test set successfully kills the mutation. Conversely, if none of the n generated test inputs identify the fault, it implies that the generated test set was unable to kill the mutation.

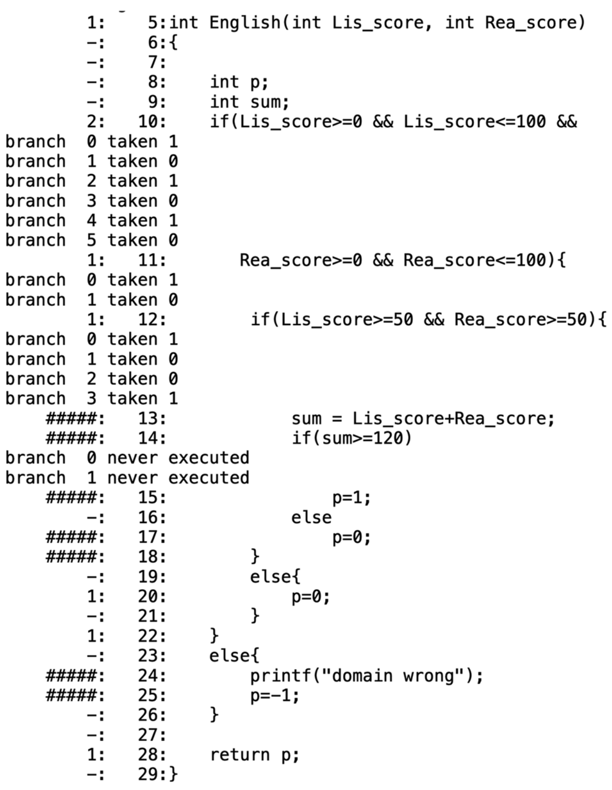


```
int English(int Lis_score, int Rea_score)
{

    int p;
    int sum;
    if(Lis_score>=0 && Lis_score<=100 &&
       Rea_score>=0 && Rea_score<=100){
        if(Lis_score>=50 && Rea_score>=50){
            sum = Lis_score+Rea_score;
            if(sum>=120)
                p=1;
            else
                p=0;
        }
        else{
            p=0;
        }
    }
    else{
        printf("domain wrong");
        p=-1;
    }

    return p;
}
```

Fig. 6 The source code of `En_testing.c`



```
        1:    5:int English(int Lis_score, int Rea_score)
        -:    6:{
        -:    7:
        -:    8:    int p;
        -:    9:    int sum;
        2:   10:    if(Lis_score>=0 && Lis_score<=100 &&
branch  0 taken 1
branch  1 taken 0
branch  2 taken 1
branch  3 taken 0
branch  4 taken 1
branch  5 taken 0
        1:   11:       Rea_score>=0 && Rea_score<=100){
branch  0 taken 1
branch  1 taken 0
        1:   12:        if(Lis_score>=50 && Rea_score>=50){
branch  0 taken 1
branch  1 taken 0
branch  2 taken 0
branch  3 taken 1
    #####:   13:            sum = Lis_score+Rea_score;
    #####:   14:            if(sum>=120)
branch  0 never executed
branch  1 never executed
    #####:   15:                p=1;
        -:   16:            else
    #####:   17:                p=0;
    #####:   18:        }
        -:   19:        else{
        1:   20:            p=0;
        -:   21:        }
        1:   22:    }
        -:   23:    else{
    #####:   24:        printf("domain wrong");
    #####:   25:        p=-1;
        -:   26:    }
        -:   27:
        1:   28:    return p;
        -:   29:}
```

Fig. 7 Gcov writes the branch execution frequency to the output file `En_testing.c.gcov`

5.3 Concolic testing

Concolic testing combines symbolic execution with concrete execution paths to improve software verification. Symbolic execution replaces normal inputs with symbolic values during program execution, allowing for the maintenance of constraint sets for each execution path. Constraint solvers are then employed to resolve the constraints and identify the inputs that lead to the execution. The primary goal of this approach is to maximize code coverage. To achieve high program coverage and automatic test case generation, KLEE (Cadar et al., 2008), a dynamic symbolic execution tool based on the LLVM compilation framework, is used as a comparative approach to our proposed BCD-based approach.

5.4 Programs under testing

The method we have proposed is designed for generating boundary test cases in software testing by optimizing a set of test cases to move towards boundary points. This approach is suitable for programs where inputs can transition from one state to another through measurable operations. The distance between two test cases, in terms of the number of operational steps required for transformation, determines the applicability of this method.

Fig. 8 DIMACS CNF format

```
p cnf 5 3

1 1 2 0
-1 2 3 0
3 -4 5 0
```

We select four programs used in the existing literature and English examination program to evaluate the effect of our approach. The descriptions of these subject programs are shown in Table 2. And the details about all five programs are shown in Tables 3 and 4, such as the dimensional number of program inputs, the range of input domain, the number of patterns, line of code (LOC), mutation faults information, and the number of boundary points (num_Bpoint).

For programs other than miniSAT, we define the minimum operation as plus one and its opposite operation as minus one. In miniSAT (Eén & Sörensson, 2003) experiment, we tested the solver.cc module in miniSAT version 2.2. The problem (test cases) is fixed as a 3-SAT problem with five variables and three clauses. This is represented in the DIMACS CNF format as Fig. 8. There are a total of 10^9 possible test patterns for this problem. In this problem, we define two operations: Change(x, y) and Pos(x, y). The operation "Change(x, y)" allows for increasing the numbers x and y (ranging from 1 to 5) in the clauses. The operation "Pos(x, y)" is used to make the y-th number in the x-th clause a positive value, for example, converting −1 to 1. There are a total of 18 types of operations, consisting of nine "Change" operations and nine "Pos" operations. Each operation corresponds to how many times it is applied. For example, the vector representing the number of operations needed to change from test case A to test case B might look like this:

$$[0, 1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] \tag{23}$$

In this vector, the "−1" signifies the inverse operation, which means either "decrease the variable number by one" or "make it negative." The absolute value of −1 corresponds to the number of times the operation is applied. In this context, the "distance" between test

Table 2 Experimental programs

Prog name	Description
triType (Williams et al., 2005)	The type of a triangle
nextDate (Awedikian et al., 2009)	Calculate the following date of the given day
findMiddle (Ghani, 2009)	Find the middle number among three numbers
English	Judge whether the English examination passed
miniSAT	Minimalistic SAT solver

Table 3 Details of subject programs

Program	Dim	Input domain		Test patterns	Size (LOC)
		From	To		
findMiddle	3	(−10, −10, −10)	(10, 10, 10)	9.26×10^3	36
English	2	(0,0)	(100, 100)	1.02×10^4	26
triType	3	(0, 0, 0)	(50, 50, 50)	1.32×10^5	41
nextDate	3	(0, 0, 0)	(2018, 12, 31)	8.39×10^5	90
miniSAT	9	(1,1,1,1,1,1,1,1,1) (-1,-1,-1,-1,-1,-1,-1,-1,-1)	(5,5,5,5,5,5,5,5,5) (-5,-5,-5,-5,-5,-5,-5,-5,-5)	1.0×10^9	1069

cases A and B is determined by the total number of operations needed to transform one into the other. For example, if there are two "−1" operations, the distance between test case A and test case B is 2.

In this paper, boundary points are obtained by manual analysis of the source code. First, the input domain is divided into *m* equivalent partitions based on the output. Then, in each equivalent partition, boundary points are generated based on the definition in Section 2.

5.5 Design parameters

In the experiment, we investigate the fault detection capabilities of RT, ART, BCD-Algorithm1 (A1), BCD-Algorithm2 (A2), and BCD-Algorithm3 (A3). First, the input domains of programs triType, nextDate, findMiddle, English, and miniSAT are divided into 5, 15, 3, 3, and 2 equivalent partitions, respectively. Then, the RT randomly generates test inputs for each equivalent partition. Specifically, ten test inputs are generated for each partition of triType, three for each partition of nextDate, ten for each partition of findMiddle, ten for each partition of English, and 15 for each partition of miniSAT. In total, 50 test inputs are generated for the program triType, 45 for nextDate, 30 for findMiddle, 30 for English, and 30 for miniSAT.

We then use A1, A2, and A3 to optimize the RT-generated test set. To generate the candidate, we use the uniform distribution as the proposal distribution. And the number of iterations for the optimization process is set to 10, 000.

Table 4 Mutation faults information and the number of boundary points

Program	Fault types					Total faults	num_Bpoint
	OBOB	ROR	AOR	SVR	LOR		
triType	6	6		3	6	21	420
nextDate	16	7			8	31	685
findMiddle	15	14			5	34	1092
English	14	7	2		2	25	283
miniSAT	8	16	2		9	35	279

5.6 Result

Table 5 shows the kill rates for the test sets generated by different methods. KLEE generates n test inputs for each of the four programs, as shown in Table 5. In Algorithm 1, BCD can be computed as $\underline{BCD}$, $\overline{BCD}$, and BCD_mean. In the definition of the BCD calculation, we use the *max* operation to calculate the boundary coverage distance. In this experiment, we also used an alternative method to replace all *max* operations in the BCD calculation process with *mean* operations, so that each boundary point affects the BCD calculation results. Under the *max* operation or *mean* operation, we denote the BCD calculation method of Algorithm 1 (A1) as A1_$\underline{BCD}_{max}$, A1_$\underline{BCD}_{mean}$, A1_$\overline{BCD}_{max}$, A1_$\overline{BCD}_{mean}$, A1_BCD_mean_{max}, and A1_BCD_mean_{mean}, respectively. In Table 5, the A2_$\underline{B_i}$ method uses Algorithm 2 to optimize the test set based on each boundary point, and the A2_$\overline{B_i}$ method optimizes the test set based on each boundary segment.

From the results, it can be seen that most BCD-based methods have better fault detection ability than RT and ART, and the kill rate of test inputs generated by BCD-based methods is better than that of concolic testing, because concolic testing is not good at detecting OBOB bugs. In the experiment of using the KLEE tool to generate test cases for the miniSAT program, we used KLEE to provide a symbolic file, and KLEE only generated one test case. It can be seen that it is difficult to test miniSAT using symbolic methods. Meanwhile, test inputs generated by A1_$\underline{BCD}_{mean}$ can kill more mutations than other methods. For the same number of iterations, A1 with BCD_{mean} accepts more candidates than A1 with BCD_{max}, as shown in Table 6. Therefore, reducing the BCD computed with the mean operation is more helpful for test set optimization than the max operation. However, the acceptance rate of A2 and A3 is slightly higher than that of A1_BCD_{mean}, but due to the large number of boundary points, judging the coverage distance of each boundary point does not give a good result.

Table 5 Kill rates for the test sets generated by different methods

Method	Kill rate				
	triType ($n = 50$)	nextDate ($n = 45$)	findMiddle ($n = 30$)	English ($n = 30$)	miniSAT ($n = 30$)
RT	0.66	0.54	0.61	0.36	0.65
ART	0.61	0.51	0.63	0.44	0.64
A1_$\underline{BCD}_{max}$	0.8	0.7	0.61	0.44	0.65
A1_$\underline{BCD}_{mean}$	0.95	1	0.97	0.8	0.94
A1_$\overline{BCD}_{max}$	0.8	0.7	0.73	0.44	0.65
A1_$\overline{BCD}_{mean}$	0.9	0.93	0.88	0.68	0.68
A1_BCD_mean_{max}	0.61	0.74	0.94	0.52	0.59
A1_BCD_mean_{mean}	0.71	0.96	0.94	0.71	0.65
A2_$\underline{B_i}$	0.85	0.96	0.76	0.52	0.85
A2_$\overline{B_i}$	0.8	0.9	0.85	0.59	0.62
A3_$\underline{B_i}$	0.76	0.93	0.55	0.44	0.62
A3_$\overline{B_i}$	0.76	0.83	0.94	0.4	0.59
KLEE	0.7 ($n = 14$)	0.9 ($n = 56$)	0.88 ($n = 13$)	0.6 ($n = 8$)	0 ($n = 1$)

Table 6 Accept rate during optimization of each method

Method	Accept rate				
	triType	nextDate	findMiddle	English	miniSAT
A1_$\underline{BCD}_{max}$	0.0026	0.0006	0	0.0003	0.0001
A1_$\underline{BCD}_{mean}$	0.0123	0.0115	0.0075	0.0081	0.00075
A1_$\overline{BCD}_{max}$	0.0025	0.0009	0.0009	0.0021	0.0003
A1_$\overline{BCD}_{mean}$	0.0151	0.0101	0.0075	0.0083	0.0073
A1_BCD_mean_{max}	0.0035	0.0011	0.0024	0.004	0.0006
A1_BCD_mean_{mean}	0.0133	0.0098	0.007	0.0076	0
A2_B_i	0.0043	0.0039	0.0016	0.0022	0.0023
A2_$\overline{\overline{B}}_i$	0.0053	0.0031	0.0012	0.0017	0.0035
A3_B_i	0.1807	0.2369	0.4172	0.3815	0.135
A3_$\overline{\overline{B}}_i$	0.1655	0.195	0.4125	0.3875	0.0748

Figure 9 shows the distribution of the data generated by RT and the optimization results of A1_$\underline{BCD}_{mean}$ on the test set generated by RT in the English experiment. From the distribution graph, it can be seen intuitively that most of the data in the optimized test set moved to the boundary.

Tables 7 and 8 show the time cost of the experiment, including the time to generate the test input (optimization time) and the mutation time. Since in the ART method, the detection of injected faults (mutation testing) is performed during the test input generation process, the time cost of the ART method is not presented in a table. the ART time costs of programs triType, nextDate, findMiddle, English, and miniSAT are 248, 572, 193, 253, and 996 s, respectively. Although the time cost of the BCD-based method is higher than other methods, the BCD-based method can generate better quality test inputs and detect more faults.

Table 7 Time cost of test input generation

Method	Test input generation time (sec)				
	triType	nextDate	findMiddle	English	miniSAT
RT	0.002	0.002	0.002	0.001	0.002
A1_$\underline{BCD}_{max}$	253	769	597	236	134
A1_$\underline{BCD}_{mean}$	242	799	295	154	142
A1_$\overline{BCD}_{max}$	432	1163	921	319	359
A1_$\overline{BCD}_{mean}$	421	1194	1043	316	382
A1_BCD_mean_{max}	442	1559	966	338	410
A1_BCD_mean_{mean}	442	1500	1004	336	404
A2_B_i	244	544	303	144	151
A2_$\overline{\overline{B}}_i$	390	541	864	298	365
A3_B_i	241	510	270	158	146
A3_$\overline{\overline{B}}_i$	389	534	830	309	387
KLEE	1	1	1	1	1

Table 8 Time cost of mutation testing

Method	Mutation time (sec)				
	triType	nextDate	findMiddle	English	miniSAT
RT	291	388	256	196	1232
A1_$\underline{BCD}_{max}$	282	440	242	195	1133
A1_$\underline{BCD}_{mean}$	287	420	265	201	1324
A1_$\overline{BCD}_{max}$	295	392	269	196	1102
A1_$\overline{BCD}_{mean}$	284	388	297	189	1309
A1_BCD_mean_{max}	285	420	273	198	1190
A1_BCD_mean_{mean}	296	405	291	198	1114
A2_$\underline{B_i}$	287	457	259	198	1266
A2_$\overline{\overline{B}}_i$	302	399	250	197	1308
A3_$\underline{B_i}$	284	346	246	198	1146
A3_$\overline{\overline{B}}_i$	284	359	243	212	1125
KLEE	68	469	67	56	43

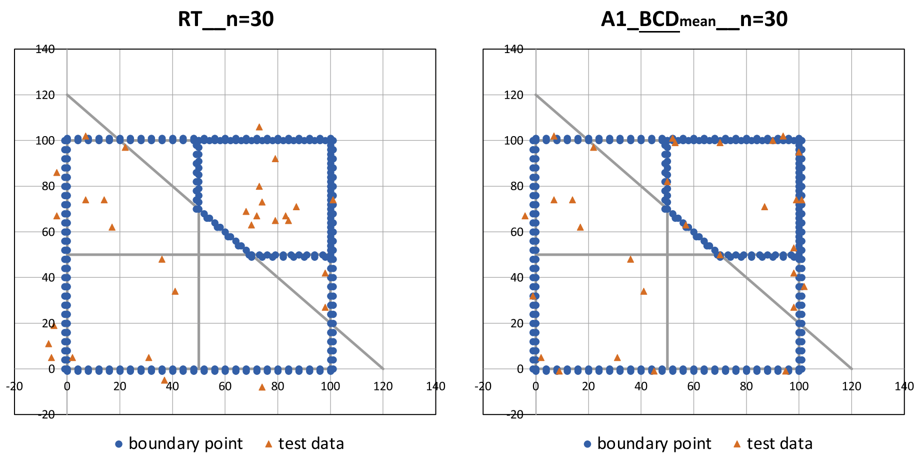


Fig. 9 The data distribution generated by RT and A1_$\underline{BCD}_{mean}$ in the English experiment

6 Conclusion

In this paper, we proposed a new boundary coverage metric called boundary coverage distance (BCD). Then, a set of randomly generated test inputs is optimized based on BCD to generate boundary values. We conducted the experiments on five programs to exhibit the performance of the BCD-based method. Our results showed that the BCD-based method can generate test inputs close to the boundary and can increase the fault detection rate of a randomly generated test set.

The selection of the boundary points significantly affects the performance of the BCD-based method. In this paper, we analyze the boundaries of several programs to obtain boundary points. However, in the case of large programs or complex numerical computation programs, the identification of the boundary becomes difficult and poses a significant problem. In this study, we manually analyze and obtain the boundary points. To improve the automation of our method, we will explore possibilities such as integrating existing analysis tools or using deep learning techniques to predict boundary points.

To generate test input for software testing, it is important to consider different types of input. If the program input is numerical data, such as continuous data, we can use Gaussian distribution or similar techniques to infer neighboring points. Conversely, for discrete data, simple addition and subtraction operations can be used to determine adjacent points. However, when the program input consists of non-numerical data, such as an array (e.g., in a sorting problem) or a string of letters, the definition of boundaries and adjacent inputs requires further consideration for future optimization.

In summary, the primary limitations of our proposed work involve the reliance on manual analysis for boundary point identification, posing challenges for larger or intricate programs. Additionally, when dealing with non-numerical data like arrays or strings, defining boundaries and adjacent inputs becomes a more complex task. In future work, we will work on overcoming these limitations.

Author contribution Conceptualization: X.G., H.O. and T.D.; Methodology: X.G., H.O. and T.D.; Formal analysis and investigation:, X.G., H.O. and T.D.; Writing—original draft preparation: X.G.; Writing—review and editing, H.O. All authors reviewed the manuscript.

Funding Open Access funding provided by Hiroshima University. This work was supported by JST, the establishment of university fellowships towards the creation of science technology innovation, grant number JPMJFS2129.

Data availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Code availability The code is available from the corresponding author on reasonable request.

Declarations

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Ali, S., Yue, T., Qiu, X., et al. (2016). Generating boundary values from OCL constraints using constraints rewriting and search algorithms[C]//*2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 379–386.

Awedikian, Z., Ayari, K., & Antoniol, G. (2009). MC/DC automatic test input data generation[C]//*Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 1657–1664.

Cadar, C., Dunbar, D., & Engler, D. R. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs[C]//*OSDI*, *8*:209–224.

Chilenski, J. J., & Miller, S. P. (1994). Applicability of modified condition/decision coverage to software testing[J]. *Software Engineering Journal, 9*(5), 193–200.

Chen, T. Y., Leung, H., & Mak, I. K. (2004). *Adaptive random testing[C]//Annual Asian Computing Science Conference* (pp. 320–329). Berlin, Heidelberg: Springer.

Chib, S., & Greenberg, E. (1995). Understanding the Metropolis-Hastings algorithm[J]. *The American Statistician, 49*(4), 327–335.

Clarke, L. A., Hassell, J., & Richardson, D. J. (1982). A close look at domain testing[J]. *IEEE Transactions on Software Engineering, 4*, 380–390.

Dobslaw, F., de Oliveira Neto, F. G., & Feldt, R. (2020). Boundary value exploration for software analysis[C]//*2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 346–353.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver[C]//*International conference on theory and applications of satisfiability testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 502–518.

Feldt, R., & Dobslaw, F. (2019). *Towards automated boundary value testing with program derivatives and search[C]//International Symposium on Search Based Software Engineering* (pp. 155–163). Cham: Springer.

Ghani, K. (2009). Searching for test data[J]. Ph. D Thesis.

Guo, X., Okamura, H., & Tadashi, D. (2023). Towards high-quality test suite generation with ML-based boundary value analysis[C]//*2023 IEEE 10th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE.

Jamrozik, K., Fraser, G., Tillman, N., et al. (2013). Generating test suites with augmented dynamic symbolic execution[C]//Tests and Proofs: 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings 7. Springer Berlin Heidelberg, 152–167.

Jeng, B., & Forgács, I. (1999). An automatic approach of domain test data generation[J]. *Journal of Systems and Software, 49*(1), 97–112.

Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing[J]. *IEEE Transactions on Software Engineering, 37*(5), 649–678.

Kosmatov, N., Legeard, B., Peureux, F., et al. (2004). Boundary coverage criteria for test generation from formal models[C]//*15th International Symposium on Software Reliability Engineering*. IEEE, 139–150.

Li, L., & Miao, H. (2012). Model-based boundary coverage criteria for logic expressions[J]. *Applied Mathematics, 6*(1S), 31S-34S.

Pandita, R., Xie, T., Tillmann, N., et al. (2010). Guided test generation for coverage criteria[C]//*2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

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

White, L. J., & Cohen, E. I. (1980). A domain strategy for computer program testing[J]. *IEEE Transactions on Software Engineering, 3*, 247–257.

Williams, N., Marre, B., Mouy, P., et al. (2005). *PathCrawler: Automatic generation of path tests by combining static and dynamic analysis[C]//European Dependable Computing Conference* (pp. 281–292). Berlin, Heidelberg: Springer.

Zhang, Z., Wu, T., & Zhang, J. (2015). Boundary value analysis in automatic white-box test generation[C]//*2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 239–249.

Zhao, R., Lyu, M. R., & Min, Y. (2010). Automatic string test data generation for detecting domain errors[J]. *Software Testing, Verification and Reliability, 20*(3), 209–236.

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy[J]. *ACM Computing Surveys (CSUR), 29*(4), 366–427.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

XIUJING GUO (M'21) received the B.S.E. degree in engineering from Dalian Jiaotong University, Dalian, China, in 2018, and the M.S. degree in engineering from Hiroshima University, Higashihiroshima, Japan. She is currently pursuing the D.Eng degree in engineering with Hiroshima University, Higashihiroshima, Japan. Her research interests include software testing, dependable computing, and machine learning.

Hiroyuki Okamura (M'03) received the B.S.E., M.S., and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 1995, 1997, and 2001, respectively. In 1998, he joined Hiroshima University as an Assistant Professor, where he has been an Associate Professor with the Department of Information Engineering, Graduate School of Engineering, since 2003. He is now a Processor with the Department of Information Engineering, Graduate School of Engineering, since 2018. His research interests include performance evaluation, dependable computing, and applied statistics. Dr. Okamura is a member of the Association for Computing Machinery, the Operations Research Society of Japan, the Institute of Electrical, Information and Communication Engineers, the Information Processing Society of Japan, the Reliability Engineering Association of Japan, and the Institute of Electrical and Electronics Engineers.

TADASHI DOHI (M'00) received the B.S.E., M.S., and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 1989, 1991, and 1995, respectively. In 1992 and 2000, he was a Visiting Researcher with the Faculty of Commerce and Business Administration, University of British Columbia, Canada, and the Hudson School of Engineering, Duke University, USA, respectively, on the leave from Hiroshima University. Since 2002, he has been a Full Professor in Hiroshima University. He has been appointed as the Dean in the School of Informatics and Data Science, Hiroshima University, since 2022. His research interests include reliability engineering, software reliability, and dependable computing. Dr. Dohi is a member of the Operations Research Society of Japan, the Institute of Electrical, Information and Communication Engineers, the Information Processing Society of Japan, the Reliability Engineering Association of Japan and IEEE. He is also an Associate Editor of the IEEE TRANSACTIONS ON RELIABILITY.