VNU Journal of Science:
Computer Science and Communication Engineering
Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

A Hybrid Method for Test Data Generation for Unit Testing of C/C++ Projects

Tran Nguyen Huong[1,2], Do Minh Kha[1], Hoang-Viet Tran[1,*], Pham Ngoc Hung[1]

[1]*VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam*
[2]*National College for Education, 387 Hoang Quoc Viet, Cau Giay, Hanoi, Vietnam*

Received 04 March 2022
Revised 24 June 2022; Accepted 12 August 2022

Abstract*:* In recent years, automated test data generation from source code has gained a significant popularity in software testing. This paper proposes a method, named Hybrid, to generate test data for unit testing C/C++ projects. The method is a combination of two test data generation methods named IIBVTG and WCFT. In IBVTG method, the source code is analyzed to find simple conditions. Then, bases on these conditions, IBVTG generates test data for boundary values without having to solve test paths constraints. This makes the method faster than BVTG method when generating test data. In Hybrid method, while generating test data using WCFT, simple conditions are collected for boundary values test data generation. Test data generated by Hybrid are able to ensure both high source code coverage and error detection ability. In addition, Hybrid is capable of finding infeasible execution paths and dead code. Experimental results with some popular unit functions show that Hybrid outperforms STCFG method in terms of test data generation time and boundary values related error detection. IBVTG is superior to BVTG in term of test data generation time whilst its boundary values related error detection ability depends on the number of simple conditions inside each unit function.

Keywords: Unit testing, test data generation, concolic testing, weighted CFG, boundary value analysis.

1. Introduction

Software testing is important to enhance the quality and reliability of software products. There are two main approaches for this purpose which are black-box and white-box testing.

Black-box testing does not rely on the internal structure of the unit under test. It relies on the unit requirement to generate test data and expected outputs. On the other hand, white-box testing bases on the internal structure of the unit

* Corresponding author.
E-mail address: thv@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.354

under test to generate test data. Currently, automated test data generation has been considered a standard approach for software quality assurance thanks to its fully automated testing ability. There are two approaches in automated test data generation known as static testing and dynamic testing. Both approaches use the symbolic execution method [1-2] to generate test data by solving the constraint expressions using an SMT-Solver. Static testing uses the analysis of the source code to generate test data. This paper focuses on dynamic testing which is the combination of source code analysis and program execution [3-7]. One of the well-known dynamic testing methods is concolic testing whose idea was firstly mentioned in DART [6]. The name concolic testing was firstly proposed in CREST [3], CUTE [8]. Later, concolic testing was improved in PathCrawler [7], CAUT [9] and SDART [10]. The main idea of concolic testing is to generate a new test data based on the previous test data execution information.

To generate test data, concolic testing follows steps below. The source code of the unit under test is converted to a control flow graph (CFG). Then, the method finds execution paths from this CFG. At the beginning, some random test data are generated and executed to find initial test paths coverage information. From this coverage information, we can find test paths which are not covered. In the next iteration, a new test data which covers one uncovered test path is generated using SMT-Solver and executed to retrieve the new coverage information. The process is repeated until no new test data can be generated. Although concolic testing gains high coverage result, it is a slow process due to the high usage of SMT-Solver.

To improve the speed of the test data generation process, many methods have been proposed. These methods focus on improving compilation process [3, 6, 10]; symbolic execution [8, 9, 11-13]; constraints optimization [4, 5, 8, 9, 14]; SMT-Solver selection and optimization [4, 5, 15]; path selection strategies [3, 6-10, 16].

In 2016, Nguyen et al., proposed a method (hereby named STCFG) to improve the existing methods [17]. However, the speed of test data generation is still slow when running with large scale projects. In our previous paper, we proposed a method named WCFT which based on a weighted control flow graph for faster test data generation process [18]. The initial experimental results show that WCFT significantly improved the test data generation time in comparison with STCFG method.

In addition to a high source code coverage and a fast test data generation process, when testing software projects in practice, the error detection ability of test data is of high importance. One of the well-known testing methods for this purpose is the boundary values testing. The reason is that errors often come from boundary related values where software testers tend to forget when testing the application.

There are three main approaches for generating boundary values test data. The first approach is to analyze the relationship between input parameters [19-21]. Initially, the method is done by a sequence algorithm. Later, a function tree method was proposed which can be applied to functions with more than two parameters. The second approach is to use mutants for comparison predicates and apply combinatorial testing [22]. This method can cover all types of boundary values and reduce the number of test data. The third approach is named Boundary Value Exploration (BVE). This method can deal with functions whose specifications are not complete, inconsistent, not clear, or even not exists [23]. However, the above approaches have high cost due to the fact that they all employ SMT solvers.

In our previous paper, we proposed a method (named BVTG) which generates boundary values related test data based on the CFG of the unit under test [18]. From another view point, boundary values should come from business definition of the software requirements. These values reside in simple conditions implemented inside source code of the unit under test. As a result, boundary values should come from

simple conditions, not from the whole test path solutions. This can make the boundary values test data generation process much faster than the method which bases on test paths constraints solving process.

This paper proposes a hybrid test data generation method (named Hybrid) which combines WCFT and the new boundary value test data generation method mentioned above (named IBVTG). This new method has a higher error detection ability while maintaining a fast test data generation time in comparison with STCFG. The key idea of the method is to use the weighted CFG of the unit under test to generate test data. In this process, a new test data is generated for the uncovered test path with the greatest weight. While the CFG is traversed to find test paths, simple conditions are collected and the corresponding boundary values test data are generated. For this method of generating test data, Hybrid can generate test data in an almost the same speed as WCFT method while having higher error detection ability. We have implemented Hybrid method in a tool named HybridCFT4Cpp to evaluate the effectiveness of Hybrid method.

The rest of this paper is organized as follows. At first, Section 2 gives the background concepts about test data generation. Then, Section 3 presents the method to generate test data from weighted control flow graph (WCFT method). Next, Section 4 shows IBVTG method to generate test data from boundary values. After that, Section 5 explains Hybrid method in detail. We give an example about the Hybrid method in Section 6. This example demonstrates the test data generation process of IBVTG and WCFT methods. Then, Section 7 presents the experimental results and discussions about the proposed methods. Related works to our methods are presented in Section 8. Finally, we conclude the paper in Section 9.

2. Background

In this section, we present some basic concepts which will be used in this paper.

Definition 1 (Control Flow Graph - CFG). *Given a unit function, we have its corresponding CFG. This is a directed graph* $G = (V, E)$. *In which,* $V = \{v_0, v_1, .., v_n\}$ *is a set of vertices representing the set of statements of the unit.* $E = \{(v_i, v_j) \mid (v_i, v_j) \in V\}$ *is a set of directed edges. Each edge* (v_I, v_j) *denotes the corresponding state from* v_i *to* v_j.

Test path is an important concept which is a sequence of vertices from the first vertex to the end vertex of a CFG. Formally, it is defined as follows.

Definition 2 (Test path). *Given a CFG* $G = (V, E)$, *a test path is a path* $\{v_0, v_1,.., v_n \mid (v_i, v_{i+1}) \in E\}$, *where* $0 \le i \le n - 1$, v_0 *and* v_n *are corresponding to the initial vertex and end vertex of the given CFG.*

Path is another important concept which is used in this paper. *Path* is a part of a test path in which all of its vertices are adjacent to each other.

Definition 3 (Path). *Given a CFG* $G=(V, E)$, *a path is a sequence of vertices* $\{v_p,v_{p+1}, ..,v_{p+k} \mid (v_i ,v_{i+1}) \in E, 0 \le p < k \le n\}$, *where n is the number of vertices of G.*

The existence of dead code often means that the project is not implemented well. However, finding dead code in a given project is a hard problem in software engineering. In this paper, we propose a method that can find dead code whilst generating test data for a given unit.

Formally, a dead path and a dead code is defined as follows.

Definition 4 *(Dead path). For a given CFG, a dead path is a path which cannot be covered by any test data.*

Definition 5 (Dead code). *A dead code is a part of source code which cannot be covered by any test data.*

In this paper, we use simple conditions to generate boundary values related test data. For example, the following two conditions $x > 5$, $x <= 10$ are simple conditions. Formally, a simple condition is defined as follows.

Definition 6 (Simple condition). *A simple condition is an expression of the form* $x\ \theta\ k$, *where* $\theta \in \{>;>=;<;<=;==,!=\}$ *is a relational operation, x is one input parameter of a function and k, called boundary value, is a specific value.*

In addition to simple condition, a condition which contains two or more input parameters is called a complex condition. For example, to check if the given three numbers a, b, c are valid values for three edges of a triangle, we use the expression $a+b>c \&\& a+c>b \&\& c+b>a$. This expression and its partial expressions $a+b>c$, $a+c>b$, and $c+b>a$ are complex conditions.

Let $F(x_1, .., x_n)$ be a unit function which takes n primitive input parameters x_1 , .., x_n. In addition, x_i has a valid value range represented by following value ranges: $a_i \le x_i \le b_i$ where a_i , b_i are domain related values ($1 \le i \le n$). We call a_i and b_i boundary values of x_i . In the test data generation for boundary values, we use another predefined number called *step* to specify the *upper boundary* and *lower boundary* values. Normally, for integer boundary values, we use *step* = 1. For float or double boundary values, *step* depends on the need of the project under test. For each boundary value *a* of one input variable *x* which has the maximum (*max*) and minimum (*min*) domain valid values, the corresponding upper boundary and lower boundary values of *a* are *a + step* and *a - step*, respectively. When generating test data for boundary values, we generate the following test values for *x*: *min*, *min + step*, *a - step*, *a*, *a + step*, *max - step* and *max*. In addition to these values, we have a special value called *norm* for *x* for combining with values of other variables to create test data for *F*. These values are shown in Figure 1.

Figure 1. Boundary points for x

For example, consider the case where the function *F* has two input parameters, x_1 and x_2. Domain values of x_1 is [min_1, max_1] and the domain values of x_2 is [min_2, max_2]. Let a_1, b_1, and $norm_1$ be test values for x_1 and a_2 , b_2 , and $norm_2$ be test values for x_2. We generate following test data set for *F*: {(min_1, $norm_2$); (a_1, $norm_2$); (b_1, $norm_2$); (max_1, $norm_2$); ($norm_1$, min_2); ($norm_1$, a_2); ($norm_1$, b_2)}

3. Generate Test Data from Weighted Control Flow Graph

This section presents WCFT method which generates test data based on weighted control flow graph [18]. The key idea of the method is that to generate test data which cover statement coverage (denoted by C1), branch coverage (denoted by C2), or sub-conditions (or modified condition/decision coverage - MC/DC) coverage (denoted by C3), we can use the CFG of the unit under test. From the CFG, we can generate a test data by solving the corresponding test path constraints which follow a predefined test coverage standard. However, the process of solving the test path constraints is a slow process due to the test path constraints solving time. Furthermore, there are test paths in which some of the statements cannot be covered because the corresponding test path constraints do not have any satisfied solution. In this case, we call these statements dead code. We proposed a method named WCFT which can improve the speed of the test data generation process and find out dead code in a given CFG[18]. The overview of WCFT method is shown in Figure 2.

Given a unit function and a coverage criteria, WCFT starts by generating the corresponding CFG using the method proposed by Nguyen et al., [17] (step 1). Then, the method initializes the weight for the generated CFG and marks all test paths to be not visited (step 2). In step 3, the method checks if there is any test path which is not visited. If the test path does not exist, the method comes to step 8. Otherwise, it continues with step 4. In step 4, the method chooses the test path which has the greatest weight to process. Then, the method generates test data by solving the corresponding test path constraints by using an SMT-Solver named Z3 and marks the test path to be visited (step 5). If the corresponding solution exists (step 6), the method stores the solution and updates weights of the CFG (step 7). After that, the method comes back to

step 3 and starts the process again. In step 8, we have the weight updated CFG (UCFG). From this UCFG, we can find dead paths as follows. The first vertex of a dead path is a condition statement of the CFG which makes the test path not feasible. The remaining vertices (except the last one) of the test path are corresponding to dead code. Details of WCFT method are shown in sections below.

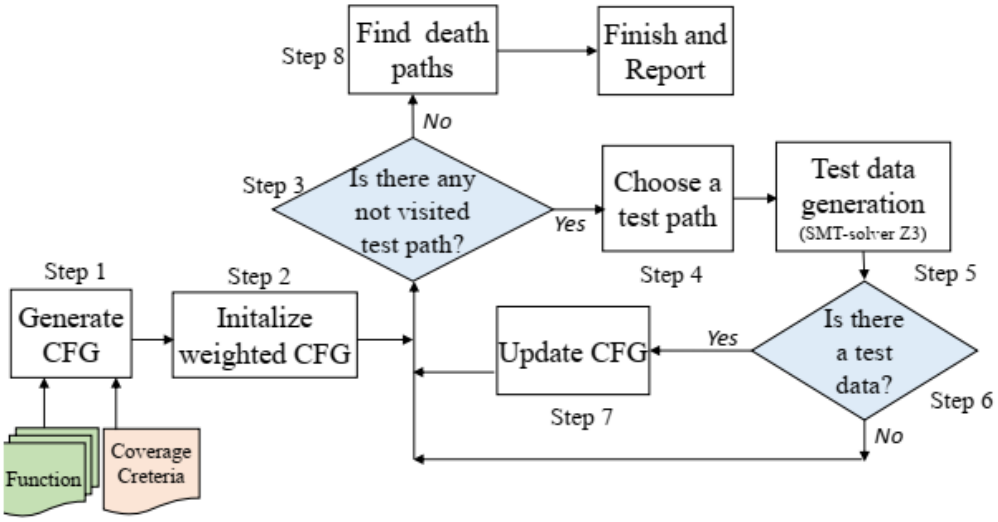

Figure 2. An overview of WCFT method.

3.1. Generate CFG for a Given Unit Function

The first step in WCFT method is to generate the CFG of the unit under test. Details of the step are shown in Algorithm 1. The algorithm accepts an abstract syntax tree (AST) block of a function (*currentBlockAST*), the required CFG (*CFG*), and a coverage criterion (*t*) as its inputs. Its output is the needed CFG. At the beginning (before calling the algorithm), CFG and *currentBlockAST* are assigned the block of AST corresponding to the function under test. The algorithm recursively breaks *currentBlockAST* into smaller blocks (until each block represents a statement) while CFG contains the result. Initially, *partial_AST* is the corresponding AST of *currentBlockAST* (line 1); *B* is the list of blocks of *partial_AST* (line 2); link_blocks is the graph which is created by linking all blocks in *B* to each other (line 3). The algorithm updates CFG by replacing *currentBlockAST* with *link_blocks* (line 4). For each block *M* in *B*, the algorithm recursively calls Algorithm 1 to divide *M* to smaller blocks (line 5 to line 9).

Algorithm 1: Generate the CFG of a unit function

input : *currentBlockAST*: source code
CFG: the required CFG
t: coverage criterion
output : *CFG*: the required CFG

1: *partial_AST* = create the *AST* of *currentBlockAST*
2: *B* = break *partial_AST* into blocks specified by *t*
3: *link_blocks* = create links between blocks in *B*
4: Update *CFG* by replacing *currentBlockAST* with *link_blocks*
5: **for** each block *M* in *B* **do**
6: **if** block *M* can be divided into smaller blocks **then**
7: **call** Algorithm 1 (*M*, *CFG*, *t*)
8: **end if**
9: **end for**

3.2. Generate Test Paths from a CFG

From the CFG generated from Algorithm 1, we can find test paths. Details of this process are presented in Algorithm 2. The algorithm takes three inputs which are the first vertex (*v*) of the CFG, the number of a loop included in the test path (*depth*), and a global variable (*path*) containing a test path being generated from the algorithm. The output of the algorithm is the list of all feasible test paths which are stored in *P*. The algorithm checks if *v* is *NULL* or the end vertex (line 1). If yes, it adds *path* to *P* (line 2). Otherwise, it checks if the number of occurrences of *v* in *path* is less than or equals to *depth* (line 3). If yes, it adds *v* to the end of *path* (line 4). After that, for all adjacent vertices *u* of *v*, the algorithm recursively calls Algorithm 2 to find all remaining vertices of the test path *path* (line 5 to 7). Line 8 is to make sure we do not add the last vertex twice to *path*.

Algorithm 2: Generate test paths from a CFG

input : *v*: The first vertex of the CFG corresponding to C3 coverage
depth: the maximum number of iterations for a loop
path: a global variable to store a test path
output : *P*: a list of feasible test paths

```
1: if v == NULL or v is the end vertex then
2:     Add path to P
3: else if the occurrence number of v in path <=depth then
4:     Add v to the end of path
5:     for each adjacent vertex u of v do
6:         call Algorithm 2 (u, depth, path)
7:     end for
8:     Remove the latest vertex added in path from it
9: end if
```

3.3. Update Weight for a CFG and Generate Test Data

After having the CFG and test paths from previous algorithms, the key steps of WCFT method is to update weight for the CFG and generate required test data. This step is performed in Algorithm 3. The algorithm takes a CFG (*CFG*) and the list of generated test paths (*TestPaths*) as inputs and generates the list of test data (*S*) as its output. The weight updated CFG (*UCFG*) is the second output of the algorithm.

Algorithm 3: WCFT-Update weight for a CFG and generate required test data

input : *CFG*: a given CFG
TestPaths: test paths of the given *CFG*
output : *S*: generated test data
UCFG: The weight updated CFG

```
1: Initialize weight for every edge of the CFG to be 1 and set all test paths to be not visited
2: while there exists a not visited test path do
3:     t ← test path which has the greatest sum of weights and is not visited
4:     constraint ← constraint expression generated from t
5:     solution = solution of constraint from SMT-Solver Z3
6:     if (solution is not null) then
7:         S.append(solution)
8:         Update test path t in CFG by adding 1 to all edges
9:     end if
10:    Mark t to be visited
11: end while
12: UCFG ← CFG
```

At the beginning, all edges of CFG are initialized with the weight of 1 and all test paths are marked as *not visited* (line 1). While there exists a test path which is not visited, the algorithm chooses a test path *t* which has the greatest sum of weights and is not visited (line 3). Then, the algorithm finds the solution (*solution*) for the constraints corresponding to *t* (line 4 to line 5). If *solution* exists, the algorithm generates a test data from *solution* and adds it to *S* (line 7) and updates *t* by adding 1 to every edge (line 8). After that, it marks *t* as *visited* and comes back to line 2 to consider other test paths.

When the algorithm finishes, CFG is the updated CFG (*UCFG*) (line 12).

3.4. Collect Dead Paths

Algorithm 4: Collect dead paths

```
input  : UCFG: a given UCFG
output : paths: dead paths if exist
 1: for (Each testPath in UCFG) do
 2:     deadPath ← ∅
 3:     for (Each edge ∈ testPath) do
 4:         if (edge.getWeight() == 1) then
 5:             deadPath.append(edge)
 6:         else
 7:             if (deadPath <> ∅) and deadPath ∉
                paths then
 8:                 paths.append(deadPath)
 9:             end if
10:             deadPath ← ∅
11:         end if
12:     end for
13:     if (deadPath <> ∅) and (deadPath ∉ paths)
        then
14:         paths.append(deadPath)
15:     end if
16: end for
```

As shown in Section 3.3, whenever Algorithm 3 can generate a test data for a test path, all its edges weights are added by 1. As a result, if there is any edge of the UCFG which has weight of 1, its related source code is a dead code. Its corresponding path is called dead path. The algorithm to collect dead paths is shown in Algorithm 4. In this algorithm, we use dot notation (".") to call a method of an edge or a path. The algorithm checks all test paths (*testPath*) of the UCFG (line 1) to find corresponding partial path (*deadPath*) which has all its edges of weight 1 (line 3 to 12). If *deadPath* is not empty, it is added to paths (line 13 to 15). When the algorithm stops, paths contains all possible dead paths of a given UCFG.

4. Generate Test Data from Boundary Values

In this section, we present IBVTG method which generates test data for unit functions from boundary values of simple conditions inside a given CFG. This is different from BVTG presented in our previous paper [18] in which test path constraints solution was employed to get boundary values. The key idea of this change is that these simple conditions from source code always include conditions defined in the requirements of the software. As a result, there is no need to solve the test path constraints to get list of boundary values for test data generation. In this paper, we only focus on primitive data types such as *short*, *int*, *long*, *f loat*, *double*, etc. In regards to *bool* data type, it is the fact that this type has only two values (*true*, *false*). For this reason, we do not generate boundary value for *bool* data type. Each numeric data type has a predefined range of valid values. However, for each system requirement, there is a valid range of value which is not necessarily the same as the range of the data type. In this paper, we only consider the domain valid value range of the parameters. We call the minimum and maximum values of the domain valid value range *Min* and *Max*, respectively.

To generate test data, before calling IBVTG, we generate the CFG with the coverage type of C3 to break all complex condition statements into simple conditions. Then, we traverse the CFG to find all simple conditions of the form $x_i \theta k$ which contains one of the input parameters. *SimpleCondList* list is passed to IBVTG for processing. From *SimpleCondList*, IBVTG can retrieve boundary values. Combining those boundary values, the corresponding data type's *Min* and *Max*, and a special value (*norm*), we can generate test data set for the unit under test. Algorithm 5 shows the detailed steps of IBVTG method. The algorithm takes three inputs which are the list of simple conditions of the unit (*SimpleCondList*), the list of the input parameters (*L*) of the unit under test, and a distance value to generate test data at the boundary values (*step*). The output of the algorithm is the set of all boundary values test data *S*.

Algorithm 5: IBVTG - Generate test data by boundary values

input : $SimpleCondList$: the list of simple conditions
$L = \{x_1, x_2, .., x_n\}$: A list of input parameters
$step$: value to generate test data at boundary values
Min, Max: Two lists of domain valid value ranges for all $x_i \in L$
output : S: The list of generated test data

```
1:  for each v = (x_i θ k) ∈ SimpleCondList do
2:      if k ∉ valueList(x_i) then
3:          valueList(x_i).add(k); // add boundary value of x to list x_i
4:      end if
5:      if (k + step) ∉ valueList(x_i) then
6:          valueList(x_i).add(k + step); // add upper boundary value
7:      end if
8:      if (k − step) ∉ valueList(x_i) then
9:          valueList(x_i).add(k − step); // add lower boundary value
10:     end if
11: end for
12: for each x_i ∈ L do
13:     min ← Min(x_i)
14:     max ← Max(x_i)
15:     valueList(x_i).add(min)
16:     valueList(x_i).add(min + step)
17:     valueList(x_i).add(max)
18:     valueList(x_i).add(max − step)
19:     valueList(x_i).sort() // sort the value list in ascending order
20:     ListNorm = ListNorm ∪ CreateNorm(x_i) // add the norm value to list
21: end for
22: for each x_i ∈ L do
23:     for each x_{i,j} ∈ valueList(x_i) do
24:         testData = {x_ij}
25:         for each t ∈ ListNorm where t = 1..n and t <> i do
26:             testData = testData ∪ ListNorm(x_t)
27:             S.add(testData)
28:         end for
29:     end for
30: end for
```

In Algorithm 5, we use *valueList* to store a list corresponding to all input parameters (L). This means that *valueList*(x_i) refers to the list of boundary values corresponding to x_i . We use *valueList*(x_i).*add*(*val*) to add a value (*val*) to the list of boundary values corresponding to x_i . In addition, we use *valueList*(x_i).*sort*() to sort the list of values of x_i in ascending order. The algorithm uses *ListNorm* to store the list of norm values corresponding to all input parameters L.

When the algorithm starts, we check all simple conditions which contains one of the input parameters of the form x_i θ k, where $\theta \in \{>, >=, <, <=, = =\}$ and $x_i \in L$. Then, the algorithm adds corresponding boundary (k), upper boundary (k + *step*), and lower boundary (k - *step*) values to *valueList*(x_i) if the list has not contained those values (line 1 to 11). Later, the algorithm adds the minimum (*min*), the upper boundary of *min* (*min* + *step*), maximum (*max*), and the lower boundary of *max* (*max* - *step*) values of the corresponding domain valid value range of every parameter to its boundary values list. After that, all the lists are sorted in ascending order for creating the norm value for x_i (line 12 to 21). Now, we have the boundary and norm values of all parameters. A *testData* is created by the combination of one boundary value of x_i and all other norm values of other parameters (line 22 to 30).

5. Hybrid Test Data Generation Method

We have presented WCFT and IBVTG methods to generate test data from weighted CFG and boundary values in Section 4 and Section 3. Although these are two different test data generation methods, they have one common point which is to use the CFG of the unit function under test to retrieve their needed information. From this observation, we present Hybrid method which integrates WCFT and IBVTG methods. The idea of Hybrid method is to extract the required information for both WCFT and IBVTG at the same time. Specifically, when traversing the given CFG to search for test paths

for WCFT, we collect the list of simple conditions for IBVTG. Then, in their turn, WCFT uses the returned test paths to generate one set of test data whilst IBVTG uses the returned list of simple conditions to generate another set of boundary values related test data. These two sets of test data are merged to have the required set of test data generated by Hybrid method. An overview of Hybrid method is shown in Figure 3.

5.1. Traverse the Given CFG

Given the CFG of the unit function under test, we can traverse it to get a list of test paths and a list of simple conditions. These lists are inputs of WCFT and IBVTG algorithms to generate required test data. The algorithm is a variation of Algorithm 2. Details of the algorithm are shown in Algorithm 6. The algorithm takes three inputs which are the first vertex (*v*) of the CFG, the number of a loop included in the test path (*depth*), and a global variable (*path*) containing a test path being generated from the algorithm. The output of the algorithm is a list of all feasible test paths (*P*) and a list of simple conditions (*C*). The difference between Algorithm 1 and Algorithm 6 is from line 5 to 7. If *v* contains any simple conditions, the algorithm adds those conditions to *C* (line 5 to 7). After traversing the given CFG, the algorithm returns the list of test paths *P* and list of simple conditions *C*.

5.2. The Hybrid Test Data Generation Method

This paper proposes Hybrid test data generation method which integrates WCFT and IBVTG. The integration is in the steps of traversing the given CFG to get a list of test paths and a list of simple conditions. Inputs of the algorithm are the CFG corresponding to the C3 coverage of the unit function under test (*CFG*), the distance value being used when generating the test data at boundary values (*step*), the list of input parameters ($L = \{x_1, x_2, .., x_n\}$), and the maximum number of iterations for a loop (*depth*). Outputs of the algorithm are a set of test data which includes test data from both WCFT and IBVTG and the list of dead paths (*DeadPaths*).

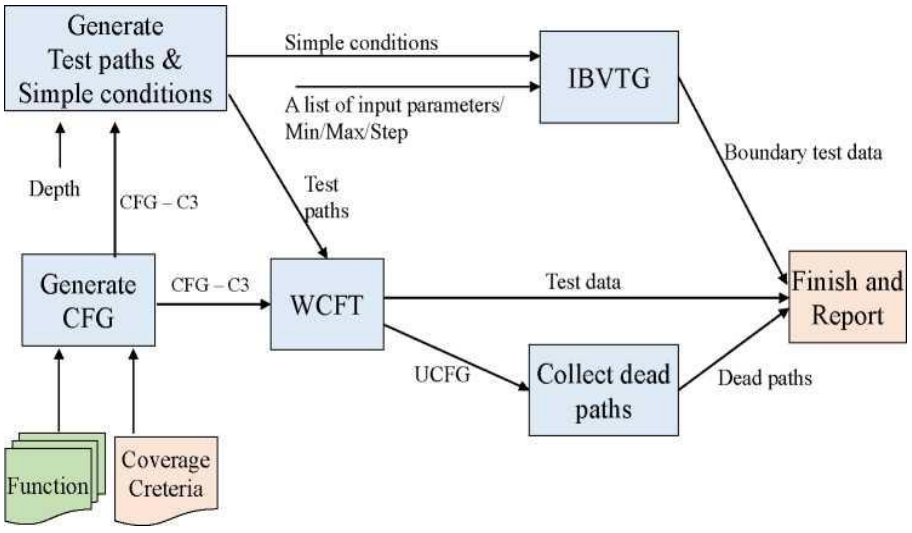

Figure 3. An overview of Hybrid method.

Details of Hybrid method are shown in Algorithm 7.

In Algorithm 7, *tempPath* is a temporary variable used for generating the list of test paths (*TestPaths*) (line 1). After initializing *tempPath* with an empty path, the algorithm starts by traversing the given CFG to get the list of test paths (*TestPaths*) and the list of simple

conditions (*SimpleCondList*) (line 2). Then, on one hand, the algorithm calls WCFT (i.e., Algorithm 3) to generate the set of test data (S_W) from weighted CFG and get the updated CFG *UCFG* (line 3). After that, the algorithm collects the list of dead paths *DeadPaths* by calling Algorithm 4 (line 4). On the other hand, the algorithm calls IBVTG (i.e., Algorithm 5 to generate another set of test data (S_B) from boundary values of the list of simple conditions *SimpleCondList* (line 5). The generated list of test data S is the union of S_W and S_B (i.e., $S = S_W \cup S_B$) (line 6).

6. Illustrative Example of Hybrid' Operation

Hybrid method is a combination of the two methods IBVTG and WCFT in which the generated test data is the union of the two test data sets generated them. The combination is obvious so we do not describe in detail. In this example, we focus on explaining how IBVTG and WCFT work to generate the required test data set and to find dead code.

We demonstrate the behaviors of IBVTG and WCFT methods using the example shown in Figure 4. This is a function, named *MathEnglishGrade*, which classifies students by their Math (*Math*) and English (*English*).

In this function, *Math* and *English* are integers with a valid range from 0 to 100. The purpose of this function is to classify students into four predefined levels 'A', 'B', 'C', and 'D' according to *Math* and *English* scores.

- *Level A*: *Math*>=50 and *English*>=60 and *Math*+*English*>=180

- *Level B*: *Math*>=50 and *English*>=60 and at least *Math*>=80 or *English*>=90

- *Level C:* *Math*>=50 and *English*>=60 and not level A or not level B

- *Level D*: *Math*<50 or *English*<60

In Figure 5, we present the domain values of *Math* and *English* scores in a 2D graph in which the horizontal axis represents *Math* and the vertical axis represents *English* scores. Students grades are shown in the corresponding area A, B, C, D. The function is shown in Figure 4 marked with numbers which are nodes in the corresponding CFG shown in Figure 6.

Algorithm 6: Generate test paths and list of simple conditions from a CFG

input : *v*: The first vertex of the CFG corresponding to C3 coverage
depth: the maximum number of iterations for a loop
path: a global variable to store a test path
output : *P*: a list of feasible test paths
C: a list of simple conditions

```
1:  if v == NULL or v is the end vertex then
2:      Add path to P
3:  else if the occurrence number of v in path
    <=depth then
4:      Add v to the end of path
5:      if v contains simple conditions then
6:          Add the simple conditions to C
7:      end if
8:      for each adjacent vertex u of v do
9:          call Algorithm 2 (u, depth, path)
10:     end for
11:     Remove the latest vertex added in path
        from it
12: end if
```

6.1. Generate Test Data using IBVTG

To generate test data using IBVTG method, we need to analyze the given CFG to get the list of simple conditions before calling IBVTG (i.e., Algorithm 5). Assume that *step* = 1. From *MathEnglishGrade* function, we can easily get the following list of simple conditions: {*Math* >= 50; *English* >= 60; *Math* >= 80; *English* >= 90}. Note that the condition *Math* + *English* >=180 is not a simple condition. In this function, according to the requirement, we can see that the value of *Math* and *English* in their simple conditions are not the same. This does not make the example lose its generality.

Algorithm 7: Hybrid test data generation

input : *CFG*: the CFG corresponding to C3 coverage
step: value to generate test data in boundary values
$L = \{x_1, x_2, .., x_n\}$: A list of input parameters
depth: the maximum number of iterations for a loop
output : S: the list of generated test data
DeadPaths: the list of dead paths

1: *tempPath* ← an empty path
2: *TestPaths*, *SimpleCondList* ← **call** Algorithm 6 (*CFG.root*, *depth*, *tempPath*)
3: $UCFG, S_W$ ← **call** Algorithm 3 (*CFG*, *TestPaths*)
4: *DeadPaths* ← **call** Algorithm 4 (*UCFG*)
5: S_B ← **call** Algorithm 5(*SimpleCondList*, *L*, *step*)
6: $S = S_W \cup S_B$

```
1.  char MathEnglishGrade (int Math, int English)
2.  {
3.      if (Math>=50 && English>=60)
                (1)          (2)
4.      {
5.          if (Math>=80 || English>=90)
                    (3)          (4)
6.          {
7.              return 'B';   (5)
8.          }
9.          else
10.         {
11.             if (Math + English>=180)
                            (6)
12.             {
13.                 return 'A';   (7)
14.             }
15.             else
16.             {
17.                 return 'C';   (8)
18.             }
19.         }
20.     }
21.     return 'D';   (9)
22. }
```

Figure 4. Source code of *MathEnglishGrade* function.

When IBVTG runs, for each simple condition in SimpleCondList, IBVTG adds three values to the corresponding list of either *Math* or *English*. For example, when checking *Math* >= 50, IBVTG adds 50, 50-*step* = 50-1 = 49, 50+*step* = 50+1 = 51 to *valueList*(*Math*). After finishing the loop from line 1 to line 11, we have the following two lists: *valueList*(*Math*) = {49; 50; 51; 79; 80; 81} and *valueList*(*English*) = {59; 60; 61;89; 90;91}. Later, IBVTG adds minimum and maximum values of the valid domain value ranges of *Math* and *English* to the list, sorts the list, and create norm value for each parameter. The algorithm also adds min's upper boundary and max's lower boundary to the list (line 12 to 21). Now we have the two lists as follows: {0; 1; 49; 50; 51; 79; 80; 81; 99; 100} and {0; 1; 59; 60; 61; 89; 90; 91; 99; 100}. Next, the method of creating norm value is as follows. From the list of simple conditions *SimpleCondList*,

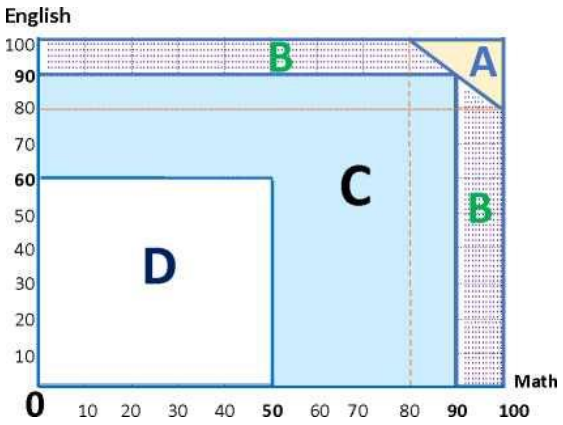

Figure 5. Student grades corresponding to *Math* and *English* scores.

we have two boundary values for *Math* which are 50 and 80. With Min and Max values of domain valid value range, we have four main values for *Math* which are 0, 50, 80, 100. From these values, we have three smaller segments [0, 50], [50, 80], [80, 100]. In this paper, we use a random selection to select norm value from the average values of those segments (i.e., norm(*Math*) = *random*{(0+ 50)/2, (50 + 80)/2, (80 + 100)/2}). The random value for *Math* in this case is 65. The same norm selection method is applied to *English*, we have the random value for *English* is 75. Later, IBVTG generates test

data for *MathEnglishGrade* function by combining *valueList* of *Math* and norm value of *English* and vice versa. We have the following set of test data: {(0, 75); (1, 75); (49, 75); (50, 75); (51, 75); (79, 75); (80, 75); (81, 75); (99, 75); (100, 75); (65, 0); (65, 1); (65, 59); (65, 60); (65, 61); (65, 89); (65, 90); (65, 91); (65, 99); (65, 100)}.

6.2. Generate Test Data using WCFT

Table 1. Test paths of MathEnglishGrade function

No.	Test paths
Test path 1	*begin* → 1 → 2 → 3 → 4 → 6 → 7 → *end*
Test path 2	*begin* → 1 → 2 → 3 → 4 → 6 → 8 → *end*
Test path 3	*begin* → 1 → 2 → 3 → 4 → 5 → *end*
Test path 4	*begin* → 1 → 2 → 3 → 5 → *end*
Test path 5	*begin* → 1 → 2 → 9 → *end*
Test path 6	*begin* → 1 → 9 → *end*

The corresponding CFG of *MathEnglishGrade* function is shown in Figure 6 in which each node is corresponding to one statement or a condition shown in Figure 4. Inputs of WCFT (i.e., Algorithm 3) are CFG of the unit under test (CFG) and the list of test paths (*TestPaths*) generated from the given CFG. For the given function, we have 6 test paths shown in Table 1.

The running process of WCFT for *MathEnglishGrade* function is shown in Table 2. At the beginning, WCFT initializes the weight of all edges with 1 (line 1). After traversing CFG, 6 test paths are obtained and 6 iterations are needed to check all test paths. At first, all test paths are marked as *not visited*. For each iteration, a *not visited* test path having the greatest weight is selected to generate test data. For that reason, in the first iteration, WCFT takes test path 1 (weight = 5) (i.e., *t* = test path 1) (line 3). The result of solving the corresponding constraints of *t* is that the solution does not exist (line 4 to 7) (marked as no solution). In the next iterations, the following test paths are selected in order: *Test path* 2, *Test path* 3, *Test path* 4, *Test path* 5, *Test path* 6. Details of the running process are shown in Table 2. In Table 2, six iterations are shown from *Iteration* 1 to *Iteration* 6. Columns "*Test paths*", "*Weight*", "*Test data (Math, English)", "Return value",* and *"Visited*" show the test paths, their weight, generated test data in form of (*Math*, *English*), the return result ('A', 'B', 'C', 'D'), and if the test path is visited, respectively. Returned values are 'A', 'B', 'C', and 'D'. In "*Test data* (*Math*, *English*)" column, a value of "*no solution*" means that the corresponding test path constraints has no solution. As a result, there is no generated test data to cover that test path. In "*Visited*" column, "*True*" value means that the corresponding test path has been checked. Otherwise, the test path has not been checked.

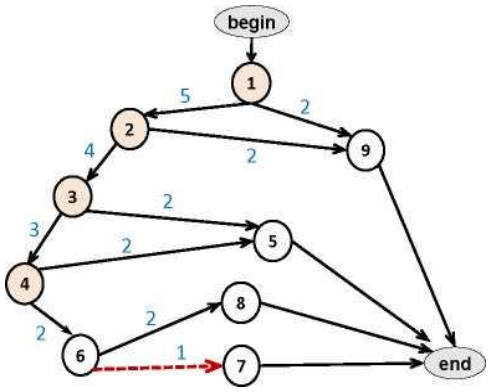

Figure 6. UCFG for *MathEnglishGrade* function.

From the data shown in Table 2, we have the following observations:

• There are 5 test data generated by WCFT. They are (50, 60), (50, 90), (80, 60), (50, 59), (49, .). The test data (49, .) means that *Math* = 49, *English* can be any value. In this case, the function returns 'D' value.

• With the minimum of 5 test data, we achieve a branch coverage of 9/10 = 90%.

• There is one branch which the generated test data cannot cover. That is the branch from node No.6 to node No.7. This is corresponding to the test path "begin->1->2->3->4->6->7->end" whose constraints expression does not have solution. This is the fact that there are no values for *Math* and *English* which satisfy both conditions *Math* <= 80 && *English* <= 90 and *Math* + *English* >= 180. The statement in node No.7 (*return* 'A') is called a *dead code*.

Table 2. The running process of WCFT for *MathEnglishGrade* function

No.	Test paths	Weight	Test data (*Math*, *English*)	Return value	Visited
Initial					
1	begin ->1 ->2 ->3 ->4 ->6 ->7 ->end	5			False
2	begin ->1 ->2 ->3 ->4 ->6 ->8 ->end	5			False
3	begin ->1 ->2 ->3 ->4 ->5 ->end	4			False
4	begin ->1 ->2 ->3 ->5 ->end	3			False
5	begin ->1 ->2 ->9 ->end	2			False
6	begin ->1 ->9 ->end	1			False
Iteration 1					
1	**begin ->1 ->2 ->3 ->4 ->6 ->7 ->end**	**5**	**No solution**		**True**
2	begin ->1 ->2 ->3 ->4 ->6 ->8 ->end	5			False
3	begin ->1 ->2 ->3 ->4 ->5 ->end	4			False
4	begin ->1 ->2 ->3 ->5->end	3			False
5	begin ->1 ->2 ->9 ->end	2			False
6	begin ->1 ->9 ->end	1			False
Iteration 2					
1	begin ->1 ->2 ->3 ->4 ->6 ->7 ->end	9	No solution		True
2	**begin ->1 ->2 ->3 ->4 ->6 ->8 ->end**	**10**	**(50, 60)**	C	**True**
3	begin ->1 ->2 ->3 ->4 ->5 ->end	7			False
4	begin ->1 ->2 ->3 ->5->end	5			False
5	begin ->1 ->2 ->9 ->end	3			False
6	begin ->1 ->9 ->end	1			False
Iteration 3					
1	begin ->1 ->2 ->3 ->4 ->6 ->7 ->end	12	No solution		True
2	begin ->1 ->2 ->3 ->4 ->6 ->8 ->end	13	(50, 60)	C	True
3	**begin ->1 ->2 ->3 ->4 ->5 ->end**	**11**	**(50, 90)**	B	**True**
4	begin ->1 ->2 ->3 ->5->end	7			False
5	begin ->1 ->2 ->9 ->end	4			False
6	begin ->1 ->9 ->end	1			False
Iteration 4					
1	begin ->1 ->2 ->3 ->4 ->6 ->7 ->end	14	No solution		True
2	begin ->1 ->2 ->3 ->4 ->6 ->8 ->end	15	(50, 60)	C	True
3	begin ->1 ->2 ->3 ->4 ->5 ->end	13	(50, 90)	B	True
4	**begin ->1 ->2 ->3 ->5->end**	**10**	**(80, 60)**	B	**True**
5	begin ->1 ->2 ->9 ->end	5			False
6	begin ->1 ->9 ->end	1			False
Iteration 5					
1	begin ->1 ->2 ->3 ->4 ->6 ->7 ->end	15	No solution		True
2	begin ->1 ->2 ->3 ->4 ->6 ->8 ->end	16	(50, 60)	C	True
3	begin ->1 ->2 ->3 ->4 ->5 ->end	14	(50, 90)	B	True
4	begin ->1 ->2 ->3 ->5->end	11	(80, 60)	B	True
5	**begin ->1 ->2 ->9 ->end**	**7**	**(50, 59)**	D	**True**
6	begin ->1 ->9 ->end	1			False
Iteration 6					
1	begin ->1 ->2 ->3 ->4 ->6 ->7 ->end	15	No solution		True
2	begin ->1 ->2 ->3 ->4 ->6 ->8 ->end	16	(50, 60)	C	True
3	begin ->1 ->2 ->3 ->4 ->5 ->end	14	(50, 90)	B	True
4	begin ->1 ->2 ->3 ->5->end	11	(80, 60)	B	True
5	begin ->1 ->2 ->9 ->end	7	(50, 59)	D	True
6	**begin ->1 ->9 ->end**	**2**	**(49, .)**	**D**	**True**

7. Experiments

To evaluate the effectiveness of the proposed methods, we have implemented a tool, named HybridCFT4Cpp, which is based on WCFT4Cpp [18]. HybridCFT4Cpp contains implementation of STCFG proposed by Nguyen et al., [17], WCFT and BVTG proposed in our previous paper [18], IBVTG and Hybrid proposed in this paper. We performed experiments to make the following two evaluations: i) To compare BVTG and IBVTG in terms of total number of test data, error detection, and test data generation time; and ii) To compare STCFG [17] and Hybrid in terms of test data generation time, C3 coverage, and error detection ability.

To prevent the affection of the test machine status on the experimental results, we have run each experiment twenty times. The average results are shown in Table 3 and Table 4. Experiments are performed on a machine whose configuration is as follows: Windows 10, Intel(R) Core(TM) i5-7100U CPU @ 2.40GHz and 8GB RAM. We use Mingw32 compiler in Dev- Cpp version 5.9.2 IDE. In our experiments, we employ an SMT-Solver named Z3 for solving test path constraints. Unit functions used in our experiments are obtained from geeksforgeeks.org, pathcrawler-online.com. Some of which have been widely used in the community.

7.1. The Comparison between BVTG and IBVTG

In this paper, we have proposed a main change to BVTG method in which SMT-Solver is no longer used to solve test path constraints. We generate test data directly from simple conditions inside the unit under test. This experiment is to compare the error detection ability at the boundary values and test data generation time of BVTG and IBVTG. For this purpose, we intentionally add errors to the functions at the boundary values and perform experiments using those functions (i.e., some simple conditions are changed). For example, x>=5 is replaced with x>5, x == 5, or x!=5. Step value used in these experiments is 1. Experimental results are shown in Table 3.

In Table 3, we have reused 5/6 functions from [18] paper. Those are *Grade*, *PDF*, *isTriangle*, *Tritype*, *leapYear*. Other functions are added to clarify the differences between BVTG and IBVTG.

When performing experiments, the coverage of both methods is C3. We compare the two methods according to three criteria: total number of test data, detected errors ratio, and test data generation time. In Table 3, columns "*Function*", "*Simp Cond*", "*Num Para*", and "*Para Type*" show ratio of number of simple conditions and total number of conditions, number of input parameters, and parameter types, respectively. The total number of test data is shown in column "*Test data*". The detected errors ratio of each function is shown in column "*Det Err*". Test data generation time is shown in column "*Time* (*ms*)".

From the main difference between IBVTG and BVTG discussed above, the experimental results shown in Table 3 are divided into a number of groups based on the correlation between the number of simple conditions and the number of constraints between the function's parameters.

The number of test data

• In functions where there are only simple conditions, the number of test data generated by BVTG is normally smaller than the number of test data generated by IBVTG. The reason is that in IBVTG, in addition to generating test data using the boundary values, IBVTG generates additional test data for boundary values of the valid value range (i.e., *Min* and *Max*) of each parameter. In our experiments, *calculateZodiac*, *Grade*, and *getFace* are functions which solely have single conditions.

• In functions where there are both simple conditions and complex conditions (i.e., conditions which contain more than one input parameters), if the number of simple conditions is greater than the number of complex conditions, the number of test data generated by IBVTG is greater than that of BVTG. In our

experiments, *i4_power*, *GCD*, and *MathEnglishGrade* are functions of this group. On the contrary, if the number of simple conditions is less than the number of complex conditions, the number of test data generated by IBVTG is less than that of BVTG. In our experiments, *Tritype*, *factorial*, and *foo* are functions of this group.

• In functions where boundary values of each parameter have duplicate values, IBVTG keeps one value from the duplicated ones. This explains why the number of test data generated by IBVTG is significantly smaller than that of BVTG. For example, *calculateZodiac* function has the following simple conditions *month* >= 2, *month* >= 4, *month* >= 5. From these conditions, IBVTG analyzes and has these boundary related values {1, 2, 3, 3, 4, 5, 4, 5, 6}. After removing the duplicate values, we have the following values {1, 2, 3, 4, 5, 6}. These values are combined with values of other parameters to create test data.

• In function where there is no simple condition, boundary values are the minimum (*Min*), maximum (*Max*) of the domain valid value range. The number of test data generated by IBVTG is typically smaller than that of BVTG. In our experiments, *isTriangle*, *PDF*, and *leapYear* are functions of this group.

Table 3. The comparison of BVTG and IBVTG

Fuction	Simple Cond	Num para	Para Type	BVTG			I BVTG		
				Test Data	Det Err	Time (ms)	Test Data	Det Err	Time (ms)
Grade	10/10	1	int	18	6/6	2	20	6/6	2
getFare	8/8	2	int	11	4/4	1116	17	4/4	1.8
calculateZodiac	60/60	3	int	31	2/4	529	17	3/4	2
i4_power	6/7	2	int	12	2/2	786	11	2/2	2.8
GCD	4/6	2	int	10	2/2	3238	10	2/2	1
MathEnglishGrade	4/5	3	int	12	4/4	1448	16	4/4	2.4
Tritype	4/10	3	double	22	1/1	19782	9	1/1	2.8
factorial	1/2	1	int	3	1/1	2484	5	1/1	23
foo	1/3	3	int	36	2/2	4547	11	0/2	8
isTriangle	0/3	3	double	9	0/2	9641	6	0/2	39
leapYear	0/3	1	int	4	1/1	2759	2	1/1	2
PDF	0/2	3	int	6	3/4	4057	6	0/4	2

Error detection ability

• In functions where there is only simple condition or there are more simple than complex conditions, the number of errors detected by IBVTG is greater than or equal to that of BVTG. In our experiments, *calculateZodiac*, *Grade*, *getFare*, *i4_power*, *GCD*, and *MathEnglish-Grade* functions are of this group.

• In functions where there are only complex conditions, the number of detected errors of IBVTG is often very low, even zero. The reason is that in this case, IBVTG only relies on the minimum and maximum values of the domain valid value range. Meanwhile, with BVTG, condition constraints are passed to the solver for getting the required solution. The generated test data helps BVTG to have a higher error detection ability in comparison with IBVTG. In our experiments, *isTriangle*, *leapYear*, and *PDF* are functions of this group.

The generation time

The time to generate test data using IBVTG is much smaller than that of BVTG. The reason is that generating test data using BVTG must use an SMT-Solver to solve test path constraints. This process takes much more time than combining boundary values of input parameters from simple conditions to generate test data in IBVTG method.

7.2. The Comparison Between STCFG and Hybrid

In this experiment, we compare test data generation time of STCFG and Hybrid methods. The results are shown in Table 4. In Table 4, Columns "*Function*", "*Dep*", "*LOC*", "*Simp Cond*", "*Num para*", and *"Para Type*" show the function name, the maximum number of loops in a test path, number of line of code of each function, ratio of number of simple conditions and total number of conditions, number of input parameters, parameter types, respectively. We compare the following two key indicators: the time each method takes to generate test data and number of detected errors. These indicators are shown in "*Time (ms)*" and "*Det Err*" columns, respectively. The coverage for both methods are shown in "*Cov* (%)" column.

In Table 4, we have reused 11/12 function from [18]. Those are *leapYear, isTriangle, PDF, CheckValidDate, Tritype, Grade, foo, calculateZodiac, simpleWhileTest, GCD.* In addition, we have reused 6 functions *Tritype, Grade, foo, GCD, Average, SelectionSort* from [17] paper. Other functions are added to enrich the experimental results. For this reason, we can have a better evaluation about the methods under experiment.

Table 4. The Comparison Between STCFG and Hybrid

Function	Dep	LOC	Sim Cond	Num Para	Para Type	STCFG		Hybrid		Cov (%)
						Time (ms)	Det Err	Time (ms)	Det Err	
calculateZodiac	0	60	60/60	3	int	13453	1/4	12879	3/4	100
CheckValidTime	0	7	6/6	3	int	344	1/6	252	6/6	100
CountSecond	0	10	12/12	3	int	825	6/6	683	6/6	100
distanceTest	0	15	5/5	1	float	2270	1/2	2728	2/2	100
getFare	0	20	8/8	2	int	1613	1/4	1771	4/4	100
Grade	0	13	10/10	1	int	705	1/6	678	6/6	100
multiConditionTest	0	30	13/13	1	short	5172	3/4	4241	4/4	100
smallIntervalTest	0	15	5/5	1	double	3593	1/2	1347	2/2	100
CheckValidDate	0	9	23/25	3	int	2443	0/6	750	5/6	88.5
i4_power	1	50	6/7	2	int	1140	1/2	927	2/2	100
MathEnglishGrade	0	13	4/5	3	int	3245	1/4	389	4/4	90
GCD	1	14	4/6	2	int	3167	2/2	3063	2/2	100
factorial	1	10	1/2	1	int	1799	1/2	1418	1/2	100
twoForLoop	3	10	2/4	2	int	21830	0/2	3463	2/2	100
CDateToNumber	1	25	2/5	3	int	4636	1/1	2730	1/1	100
Average	2	14	2/5	5	double int	10248	0/0	7061	0/0	75
Tritype	0	40	4/10	3	double	28823	1/1	22126	1/1	100
foo	0	15	1/3	3	int	898	1/2	947	2/2	87.5
NextDate	1	25	2/6	3	int	26186	2/2	21093	1/2	75
MoreComplexCond	0	20	3/18	5	long	14741	1/3	13899	3/3	100
simpleWhileTest	4	6	0/1	2	int	2224	1/1	1447	1/1	100
pdF	0	6	0/2	3	int	161	1/4	798	3/4	100
isTriangle	0	6	0/3	3	double	1543	1/1	747	1/1	100
leapYear	1	6	0/3	1	int	202	0/1	200	1/1	100
SelectionSort	2	20	0/3	2	int	18654	0/0	3827	0/0	100

From experimental results shown in Table 4, we have the following observations:

• *Coverage*: The coverage of the generated test data of the two methods is the same. This comes from the fact that the data generation process of both methods is performed based on the same CFG and an SMT-Solver (i.e., Z3).

• *Error detection ability*: The number of errors detected by Hybrid is greater than or equal to that of STCFG. The reason is that the test data set generated by Hybrid method includes both test data generated from the corresponding CFG and from boundary values. Meanwhile, STCFG generates test data only from the given CFG.

• *The generation time*: The test data generation time of STCFG is greater than that of Hybrid in most cases. The reason is that STCFG checks all sub-execution paths in the process of finding a full-execution path. When checking, these sub-execution path constraints are solved by an SMT-Solver. This makes STCFG slower than WCFT which requires the whole test path constraints to be solved only once for a full-execution path. In addition, because IBVTG has much faster speed than BVTG, including IBVTG in Hybrid does not make Hybrid slower than STCFG. The functions *Average*, and *SelectionSort* do not have errors. We can see that the test data generation time of Hybrid is less than that of STCFG.

8. Related Works

There are some research related to the proposed methods in this paper. They are researches about ganerating test data automatically from the source code [10, 16-18] and at the boundary values [19, 21-23].

Nguyen et al., improved the execution path exploration method from CFG [17]. In this method, the source code is converted to the coresponding CFG. Then, CFG is explored to find the test path using backtracking algorithm. In each step of exploration process, at each decision node, the feasibility of the test path from the inittial node to the decision node is examined. This method eliminates infeasible execution paths as soon as possible. However, it takes the method long time to process the constraints when CFG has many nodes and infeasible execution paths.

Nguyen et al., proposed *SDART* [10] method to improve the coverage by combining the breadth first search strategy of *DART* [6] with the static test data generation method. Specifically, after some times where the generated test data do not increase the code coverage, the method uses static testing to generate test data.

Marashdih et al., proposed the path weight method (PWM) to avoid detecting duplicated feasible paths [16]. The method uses an SMT-slover to check the feasibility of a given test path.

In our previous paper [18], we proposed a method to generate test data based on weighted CFG (named WCFT). This is the method presented in Section 3 in this paper. This paper proposes an improved method for BVTG, named IBVTG, by using simple conditions to generate test data. IBVTG can greatly reduce the boundary test data generation time in comparison with BVTG. In addition, we combine the two methods of WCFT and IBVTG methods to have a hybrid method which has the advantages of both WCFT and IBVTG methods.

Feng et al., proposed a number of papers for boundary values related generation methods. These methods are for the cases where function parameters are related to each others.

First, Feng et al., proposed a sequence method to evaluate the relationship between input parameters [19]. The method has a limitation of low test data generation performance with functions of the form $Y = f(X)$ and $Z = g(Z,Y)$. The reason is that the sequence method can only be applied for two functions. For other functions, the method cannot generate test data.

Second, Feng et al., proposed Divide-and-Rule method to break the dependency between input parameters for creating independent variables [20]. The method can create some

required test data which cannot be generated by traditional methods.

Finally, Wenying Feng et al. proposed a function tree method to generate test data at boundary values in case the input parameters are constrained to each other [21]. To find test data for a given function with three inputs X, Y, and Z, the method assumes that Y is a function of X and Z. The method is inspired from geometry view and is a generalization of the traditional boundary value analysis in black-box testing. We share the interest in generating test data for boundary values. However, our proposed method bases directly on simple conditions, but not on the dependencies between input parameters of a function.

For achieving the highest coverage, Zhang et al., proposed a method of defining the boundaries of condition predicates used in white-box testing [22]. The method employed constrained combinational testing to cover these boundaries with reduced number of test cases. The method can guarantee to cover all possible boundaries for each selected execution path. We share the interest in generating test data for boundary values as Zhang et al., However, our IBVTG method bases directly on simple conditions, but not on the boundaries of execution paths.

Dobslaw et al., proposed a Boundary Value Exploration (BVE) method to detect and identify boundary inputs [23]. Additionally, Dobslaw proposed two concrete BVE methods based on information-theoretic distance functions: i) A boundary detection algorithm; and ii) A method to explore the behavior of the unit under test and identify its boundary behaviors. We share the interest about generating test data for boundary values. However, we focus on using simple conditions of input parameters.

9. Conclusion

This paper proposes IBVTG method to improve the boundary values test data generation method (BVTG) and Hybrid method which is the combination of IBVTG and WCFT methods. With IBVTG method, we analyze the source code to find simple conditions and generate test data sets from those conditions. The method is capable of detecting errors at boundary values without using an SMT-Solver. Experimental results show that it takes IBVTG less time to generate test data than BGVT method. Depends on test cases, the test data set generated by IBVTG has higher error detection capacity whilst having a smaller number of test data than that of BVTG.

In Hybrid method, we analyze the source code of the unit under test, generate corresponding CFG, assign weights to the graph, and select the execution path with the highest weight to generate test data. In the process of choosing an execution path, we also collect simple conditions which are used for IBVTG method to generate test data for boundary values. As a result, Hybrid method generates test data which both ensure source code coverage while having high error detection ability. In addition, Hybrid method keeps the same capability as WCFT to detect infeasible execution paths or dead code. These methods are implemented in the same tool named HybridCFT4Cpp. Experimental results with some popular unit functions in the research community show that these methods are superior to STCFG method in terms of test data generation time and boundary values error detection ability.

Although Hybrid method is implemented in a tool and some experiments are performed with some common unit functions in the research community, there are many works we need to do. The current Hybrid method can generate test data for only primitive types, we need to extend the method for other complex types such as pointers, struct, class, etc. Moreover, we need to implement more advanced user interface for HybridCFT4Cpp so that normal software engineers can use. This aims to make the method more widely used in projects in practice.

Acknowledgements

This work has been supported by VNU University of Engineering and Technology under project number CN21.18.

Do Minh Kha was funded by Vingroup JSC and supported by the Master, PhD Scholarship Programme of Vingroup Innovation Foundation (VINIF), Institute of Big Data, code VINIF.2021.ThS.24.

References

[1] J. C. King, Symbolic Execution and Program Testing, Commun. ACM, Vol. 19, No. 7, 1976, pp. 385-394, https://doi.org/10.1145/360248.360252.

[2] C. Cadar, K. Sen, Symbolic Execution for Software Testing: Three Decades Later, Commun, ACM 56 Vol. 2, 2013, pp. 82-90, https://doi.org/10.1145/2408776.2408795.

[3] J. Burnim, K. Sen, Heuristics for Scalable Dynamic Test Generation, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, IEEE Computer Society, USA, 2008, pp. 443-446, https://doi.org/10.1109/ASE. 2008.69.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, EXE: Automatically Generating Inputs of Death, ACM Trans. Inf. Syst. Secur, Association for Computing Machinery, New York NY, United States, Vol. 12, No. 2, 2008, pp 1-38, https://doi.org/10.1145/1455518.1455522.

[5] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, USA, 2008, pp. 209-224.

[6] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed Automated Random Testing, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 213-223, https://doi.org/10.1145/1065010.1065036.

[7] N. Williams, B. Marre, P. Mouy, M. Roger, Pathcrawler: Automatic Generation of Path Tests by Combining Static And Dynamic Analysis, in: M. Dal Cin, M. Kaâniche, A. Pataricza (Eds.), Dependable Computing - EDCC 5, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 281-292.

[8] K. Sen, D. Marinov, G. Agha, Cute: A Concolic Unit Testing Engine for C,, Association for Computing Machinery, New York, NY, USA, Vol. 30, 2005, pp. 263-272, https://doi.org/10.1145/1095430.1081750.

[9] Z. Wang, X. Yu, T. Sun, G. Pu, Z. Ding, J. Hu, Test Data Generation for Derived Types in C Program, in: Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE '09, IEEE Computer Society, USA, 2009, pp. 155-162, https://doi.org/10.1109/TASE.2009. 10.

[10] D. A. Nguyen, T. N. Huong, H. D. Vo, P. N. Hung, Improvements of Directed Automated Random Testing in Test Data Generation for C++ Projects, International Journal of Software Engineering and Knowledge Engineering, Vol. 29, 2019, pp. 1279-1312.

[11] Z. Xu, T. Kremenek, J. Zhang, A Memory Model for Static Analysis of C Programs, in: Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I, ISoLA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 535-548.

[12] B. Elkarablieh, P. Godefroid, M. Y. Levin, Precise Pointer Reasoning for Dynamic Test Generation, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 129-140, https://doi.org/10.1145/1572272.1572288.

[13] D. Trabish, A. Mattavelli, N. Rinetzky, C. Cadar, Chopped Symbolic Execution, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 350-360, https://doi.org/10.1145/3180155.3180251.

[14] D. M. Perry, A. Mattavelli, X. Zhang, C. Cadar, Accelerating Array Constraints in Symbolic Execution, ISSTA 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 68-78, https://doi.org/10.1145/3092703.3092728.

[15] H. Palikareva, C. Cadar, Multi-solver Support in Symbolic Execution, in: Proceedings of the 25th International Conference on Computer Aided Verification, CAV 2013, SpringerVerlag, Berlin, Heidelberg, Vol. 8044, 2013, pp. 53-68.

[16] A. W. Marashdih, Z. Zaaba, K. Suwais, An Approach for Detecting Feasible Paths Based on

Minimal Ssa Representation and Symbolic Execution, Applied Sciences, Vol. 11, 2021, pp. 5384, https://doi.org/10.3390/ app11125384.

[17] D. A. Nguyen, P. N. Hung, V. H. Nguyen, A Method for Automated Unit Testing of C Programs, in: 2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS), 2016, pp. 17-22, https://doi.org/10.1109/NICS.2016.7725644.

[18] T. N. Huong, D. M. Kha, H. V. Tran, P. N. Hung, Generate Test Data from C/C++ Source Code using Weighted Cfg and Boundary Values, in: 2020 12th Int. Conf. on Knowledge and Systems Engineering (KSE), 2020, pp. 97-102, https://doi.org/10.1109/KSE50997.2020. 9287629.

[19] W. Feng, Z. Zhang, Sequence Algorithms for Boundary Value Analysis With Constrained Input Parameters, in: R. T. Hurley, W. Feng (Eds.), Proceedings of the ISCA 14th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering, July 20-22, 2005, Novotel Toronto Centre, Toronto, Canada, ISCA, 2005, pp. 255-160.

[20] K. Vij, W. Feng, Boundary Value Analysis Using Divide- and-rule Approach, in: Fifth Int. Conf. on Information Technology: New Generations (itng 2008), 2008, pp. 70-75, https://doi.org/10.1109/ITNG.2008.218.

[21] W. Feng, A Generalization of Boundary Value Analysis for Input Parameters with Functional Dependency, in: Computer and Information Science, ACIS Int. Conf. on, IEEE Computer Society, Los Alamitos, CA, USA, 2010, pp. 776-781, https://doi.org/10.1109/ICIS.2010.39.

[22] Z. Zhang, T. Wu, J. Zhang, Boundary Value Analysis in Automatic White-Box Test Generation, in: 2015 IEEE 26th Int, Symposium on Software Reliability Engineering (ISSRE), 2015, pp. 239-249, https://doi.org/10.1109/ISSRE.2015.7381817.

[23] F. Dobslaw, F. G. D. O. Neto, R. Feldt, Boundary Value Exploration for Software Analysis, 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). https://doi.org/10.1109/icstw50294.2020.00062.