Boundary Value Analysis in Automatic White-box Test Generation

Zhiqiang Zhang*†, Tianyong Wu*† and Jian Zhang*
{zhangzq, wuty, zj}@ios.ac.cn

*State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences

†University of Chinese Academy of Sciences

***Abstract*—White-box testing is an effective technique for generating test cases to provide high coverage for programs. We usually select several execution paths using some strategy, and generate a corresponding test case that follows each path. Each execution path corresponds to an input subspace bounded by constraints. Extreme values in these subspaces are very likely to trigger failures since the constraints bounding input subspaces may be faulty. In this paper, we propose a new way of defining the boundaries of comparison predicates in white-box testing, and apply constrained combinatorial testing to cover these boundaries with reduced number of test cases. Our approach can guarantee to cover all possible boundaries for each selected execution path, thus achieving high fault coverage for boundary faults, while the original structural coverage is still preserved.**

***Index Terms*—white-box test generation, boundary value analysis, mutation, constrained combinatorial testing.**

I. Introduction

Software testing is one of the most adopted approaches to ensure high software quality and reliability. Specially, white-box testing is an effective technique to generate test cases based on program structures. In recent years, several automatic techniques were proposed to solve this problem, among which Symbolic Execution (SE) [35] and Dynamic Symbolic Execution (DSE) [20] are two state-of-the-art approaches. Generally, they traverse different program paths according to a given structural coverage criterion, and then collect the symbolic constraints along each execution path. The constraints of a path correspond to an input subspace, which will be solved by a constraint solver to generate a new test case. Existing test generation tools (like SimC [33], CUTE [30], KLEE [6], etc.) usually use statement or branch coverage as their elementary criterion. However, recent experimental results show that statement or branch coverage cannot be treated as the sole criteria for effective testing, since they have low fault detection capacity in some situations [13].

The input space of a program is partitioned into several subspaces by path conditions. Each subspace is bounded by several constraints in its path condition. Test cases in the same subspace have the same execution trace, while test cases in different subspaces have different execution traces. A common type of faults is at the subspace boundaries. The developer may make mistakes on the constraints, therefore making inputs

```
1  bool is_inside_rect(int x, int y) {
2      if (x <= 0 || x >= 10) {
3          return false;
4      }
5      if (y <= 0 || y > 10 /* should be >= */) {
6          return false;
7      }
8      return true;
9  }
```

Fig. 1: An example program

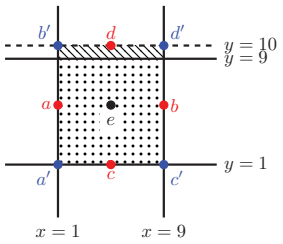

Fig. 2: The input space for `is_inside_rect`

which should be in a subspace be partitioned into a wrong subspace. A motivating example is shown in Figure 1. The function checks whether a 2D point (x,y) is in a rectangular region. The program has a fault in constraint `y>10`, which should be `y>=10`.

The input space of the function is shown in Figure 2. The solid lines are the correct boundaries for the rectangle area, and the dashed line is the wrong boundary for `y>10` as implemented in the program. Suppose we are selecting input inside the rectangle. Typically in SE or DSE, a single input is selected for each generated path (subspace), and any arbitrary value inside the rectangle may be chosen. If the chosen value is point e, then the fault will not be detected. For this reason, we need to generate test cases covering the boundary values to detect the boundary faults.

Boundary-value analysis (BVA) is a testing technique for detecting these boundary faults. The technique was originally used in black-box testing, and usually assumes that the input variables are independent. Thus there are some difficulties to apply BVA to white-box testing, since the path conditions

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

usually make the variables dependent on each other. Recently, researchers have proposed several approaches to apply BVA in white-box test generation [29], [22].

Let's go back to Figure 2. When considering BVA, we want to check each boundary of the rectangle. One way to do that is to select an input on each of the boundaries. For example, we choose points a, b, c and d, then the output for d will be faulty. However, the cost of selecting one data point for each boundary is relatively large. Actually, the coverage of all the four boundaries can be achieved by using only two data points, either $\{a', d'\}$ or $\{b', c'\}$, since one test case can cover multiple boundaries at the same time.

In this paper, we propose a BVA-aware white-box test generation technique. First, we give a new definition for boundary conditions of comparison predicates. Then, we use a technique called combinatorial testing to cover these boundary conditions with reduced number of test cases, while still preserving the original structural coverage.

The remainder of this paper is organized as follows: Section II describes the basic concepts and techniques used in our approach, including the introduction to white-box test generation and constrained combinatorial testing. Section III talks about BVA in white-box testing. Section IV describes our BVA-aware white-box test generation technique. Section V discusses topics related to our approach. Section VI presents our experimental results, and Section VII discusses the related work. Finally, we conclude our work in Section VIII.

II. Preliminaries

A. White-box test generation

White-box test generation takes the source code as input and automatically generates a series of inputs to test the software. Because of the infinite execution space of the software, a coverage criterion is usually required to evaluate the quality of test suites. Generally, the test generation procedure can be summarized as Fig. 3, and it contains three key steps.

- **Conversion:** Convert the source code into an intermediate structure, like control flow graph (CFG). A CFG may be treated as a directed graph that consists of nodes and directed edges between two nodes and each node represents a basic block, which is a straight-line piece of code without any jumps and jump targets.
- **Path generation:** This step is an iterative step. It selects one program path from CFG at one iteration, and ends till the given coverage criterion is satisfied.
- **Path feasibility checking & test data generation:** After extracting an execution path from the CFG, we need to check its feasibility, and generate test data to trigger the path, if it is a feasible path. A commonly used approach to solve this problem is based on symbolic execution and constraint solving [35].

B. Constrained combinatorial testing

Combinatorial testing (CT) aims at testing parameterized systems. The system under test (SUT) is modeled as a black-box, taking several parameters as input, and each parameter has several possible value options. Given strength t, CT is able to generate a *covering array*, which covers all possible combinations between any t parameters.

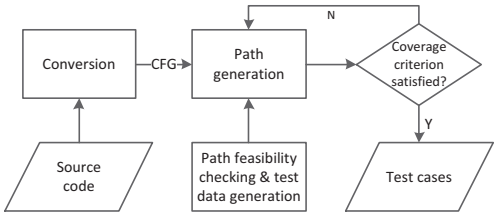

Fig. 3: A common procedure for white-box test generation

Traditional CT assumes that all parameters are independent. However in real scenarios, there are usually constraints on the parameters. A test case violating the parameter constraints will be invalid and cannot be executed. When generating test cases with the presence of parameter constraints, the CT test generation algorithm should generate only valid test cases and cover as many t-way parameter combinations as possible. This kind of constraint-aware CT is called *constrained combinatorial testing*.

Here we give an example. Suppose we are generating a covering array for an SUT model, which has 4 input parameters p_1, p_2, p_3, p_4, each taking values in $\{1, 2, 3\}$. Suppose there is one constraint: "$p_1 = 1 \rightarrow p_2 = 2$". Table I shows a covering array of strength 2. The array covers all possible pairs of parameter values, except for "$p_1 = 1 \wedge p_2 = 1$" and "$p_1 = 1 \wedge p_2 = 3$", which violate the constraint.

TABLE I: An example covering array

p_1	p_2	p_3	p_4
3	2	1	3
2	2	2	2
2	3	3	3
3	3	2	1
2	1	1	1
3	1	3	2
1	2	3	1
3	3	1	2
3	1	2	3
1	2	2	2
1	2	1	3

Another useful extension for CT is *seeding*. Seeding allows the tester to specify additional coverage requirements as complement to the given universal strength t. This is done by giving several critical parameter combinations (a.k.a. *seeds*), which must be tested. The test generation algorithm will generate a covering array as small as possible to cover all t-way combinations as well as the seed combinations.

III. BVA in White-box Testing

BVA is originally used as a black-box testing technique, which is usually combined with equivalence class partitioning (ECP). ECP mainly focuses on the partition of program inputs according to its behaviors, while BVA focuses on the extreme values of different partitions. Typically, when using BVA in black-box testing, for each boundary (e.g. $x = 10$), we need

one value far below the boundary ($x = -100$), one value just below the boundary ($x = 9$), one value just on the boundary $x = 10$, one value just above the boundary ($x = 11$), and one value far above the boundary ($x = 100$).

In white-box testing, BVA and ECP have their respective counterparts. The path selection strategy performs similar functions with ECP. The program input space can be considered to be partitioned into several subspaces, where all the inputs of each subspace have the same execution path.

However, the BVA input selection method used in black-box testing cannot be directly used in white-box testing. This is because BVA in black-box testing usually assumes the input variables are independent, thus it only needs to focus on individual variables. As for BVA in white-box testing, each input subspace is bounded by multiple constraints, which may make the input variables dependent. Besides, we need to generate boundary inputs for all the bounding constraints.

In our work, we only generate boundary inputs inside each subspace. This may make only one side of an edge be tested. But the other side will be tested when generating boundary inputs for other subspaces. Besides, in black-box BVA, a value far from the boundaries is usually needed. In our approach, selecting inputs far from the boundaries is optional. If we need to select these non-boundary values, the number of test cases will increase.

Here we call each atomic Boolean expression in the path condition a *predicate*. Predicates could be Boolean variables, comparison predicates ($>, \geq, <, \leq, =, \neq$), etc., and should not contain any Boolean operator (such as $\wedge$, $\vee$, $\neg$, etc.). In white-box BVA, we want to cover the boundary values for comparison predicates.

One problem with BVA in white-box testing is how to define the boundaries. It might not be as easy as it seems.

A. How close should boundary values be to the edge?

The first challenge is that the boundary values may have different distances from the edge if the variables are integers. We measure this distance of an input to the edge of a comparison predicate by calculating the difference between the left-hand side and right-hand side of the predicate. For example, for constraint $x < 2$, the boundary value is $x = 1$, making the difference between the two sides of the inequation be 1; However for constraint $2 * x < 2$, the boundary value is $x = 0$, yet making the difference between the two sides be 2.

A more complex example is shown in Figure 4. There are two highlighted subspaces (filled with dots and lines respectively). We are selecting boundary values for constraint $2x + 3y < 18$. For the subspace filled with dots, the boundary value is point $a = (2, 4)$, making the difference between the two sides be 2. For the subspace filled with lines, the boundary value is point $b = (4, 3)$, making the difference between the two sides be 1.

Generally, to constrain the input near the boundary, we can select an appropriate Δ and restrict the absolute difference between the two sides to be smaller than or equal to Δ. However, such a Δ is sometimes hard to find, and the resulting solution might not be the closest to the boundary. Kosmatov et al. [24] discussed how to define boundary values using neighborhood. However this problem still needs further research.

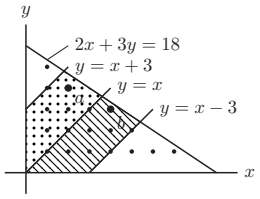

Fig. 4: An illustration of boundary values

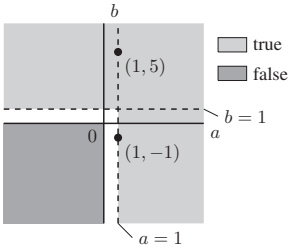

Fig. 5: Boundary and non-boundary values of $a > 0 \vee b > 0$

In our work, we fix Δ to 1 for integer comparisons, or a very small real number for floating-point comparisons. This is reasonable since most of boundary faults are off-by-one bugs (OBOB), which is a kind of faults when some computation process uses some wrong value which is 1 more or 1 less than the correct value.

B. Physical boundaries or determining boundaries

Although some input may be close to the edge of a predicate, it still may not be a boundary value of the whole path condition. See Figure 5. Suppose we have path condition $(a > 0) \vee (b > 0)$. Although input $(a, b) = (1, 5)$ is close to the edge of predicate $a > 0$, it is not a boundary value. A correct boundary value for predicate $a > 0$ is $(a, b) = (1, -1)$. There are two possible approaches: physical boundaries and determining boundaries.

1) Physical boundaries: One way to find this kind of boundary values is to look for the physical boundaries. If some input making the path condition evaluates to true, and if it moves a small distance towards the predicate edge, the path condition will evaluate to false. Kosmatov et al. [24] introduced several ways to define and find physical boundaries. Physical boundaries may have problem when dealing with path conditions with disjunctions. For example, suppose we have path condition $(a > 0) \vee (a \leq 0)$, then there will be no physical boundary.

2) Determining boundaries: In our work, we use an input that is near the edge of a given predicate and makes the predicate determine the evaluation result of the path condition as a boundary value. The term "determine" originates from the GACC coverage criterion [2], which is a version of MC/DC (modified condition/decision coverage). It means that

TABLE II: An input determining $(a > 0) \vee (b > 0)$

	$a > 0$	$b > 0$	$a > 0 \vee b > 0$
$(a, b) = (1, -1)$	true	false	true
replace $a > 0$ with false	**false**	false	false

TABLE III: Mutants for comparison predicates

Predicate	Mutant 1	Mutant 2
$A < B$	$A < B - \Delta$	$A < B + \Delta$
$A \leq B$	$A \leq B - \Delta$	$A \leq B + \Delta$
$A > B$	$A > B - \Delta$	$A > B + \Delta$
$A \geq B$	$A \geq B - \Delta$	$A \geq B + \Delta$
$A = B$	$A = B - \Delta$	$A = B + \Delta$
$A \neq B$	$A \neq B - \Delta$	$A \neq B + \Delta$

the evaluation of the predicate directly affects the decision of the path condition. It can be formally defined as follows:

Definition 1. Suppose we have a path condition $\mathcal{P}$, having predicates $c_1, c_2, \ldots, c_n$. An input τ makes predicates $c_1, c_2, \ldots, c_n$ evaluate to $b^\tau_{c_1}, b^\tau_{c_2}, \ldots, b^\tau_{c_n}$, and lets $\mathcal{P}$ evaluate to $b^\tau_{\mathcal{P}}$. Predicate c_i is said to determine $\mathcal{P}$ under τ if switching only $b^\tau_{c_i}$ to $\neg b^\tau_{c_i}$ will affect the evaluation of $\mathcal{P}$, i.e.

$$\begin{array}{rcl} b^\tau_{c_1}, b^\tau_{c_2}, \ldots, b^\tau_{c_i}, \ldots, b^\tau_{c_n} & \Rightarrow & b^\tau_{\mathcal{P}} \\ b^\tau_{c_1}, b^\tau_{c_2}, \ldots, \neg b^\tau_{c_i}, \ldots, b^\tau_{c_n} & \Rightarrow & \neg b^\tau_{\mathcal{P}} \end{array} .$$

An equivalent definition for "c_i determines $\mathcal{P}$ under τ" is that τ must satisfy $\mathcal{P} \oplus \mathcal{P}_{c_i, \neg c_i}$, where $\oplus$ is logical xor, and $\mathcal{P}_{a,b}$ is the constraint of replacing all occurrences of a in $\mathcal{P}$ with b.

For example, $(a, b) = (1, -1)$ is a boundary value of $(a > 0) \vee (b > 0)$ w.r.t. predicate $a > 0$. As shown in Table II, $a > 0$ evaluates to true and $b > 0$ evaluates to false, and the whole path condition evaluates to true. If we replace $a > 0$ with false, the whole path condition will evaluate to false.

To generate a determining boundary value input, we use a constraint to constrain the input on the boundary, and use a constraint solver to solve it. Suppose the path condition is $\mathcal{P}$, and c is one of its comparison predicates. c has two off-by-one mutants (as shown in Table III). We want to generate inputs to test whether $\mathcal{P}$ has a boundary fault on c. We propose the following definition:

Definition 2. For each off-by-one mutant c' of each predicate c, we define the following condition as a *boundary condition*:

$$\mathcal{P} \wedge \neg \mathcal{P}_{c,c'}.$$

This boundary condition constrains the input to make $\mathcal{P}$ evaluate to true, and also constrains c and c' to evaluate to different values, which makes the input just near the edge of c, and lets c determine $\mathcal{P}$. From another perspective, this condition means that the input must follow the current path, and will follow a different path if predicate c is implemented as c'. If c is faulty and the correct predicate should be c', a faulty path will be detected by the corresponding test case.

IV. Covering Boundary Values in White-Box Testing

In this section, we describe our approach to applying BVA while preserving the original coverage criterion. Normal white-box test generation techniques take the program source or model as input, try to explore execution paths to achieve a certain coverage criterion, and generate a corresponding test case for each path. A straightforward way to make the test suite also cover the boundary values is to generate a test case for each boundary condition for each path, a.k.a. using the base choice (BC) strategy [1]. However, in this way the number of test cases may be too large.

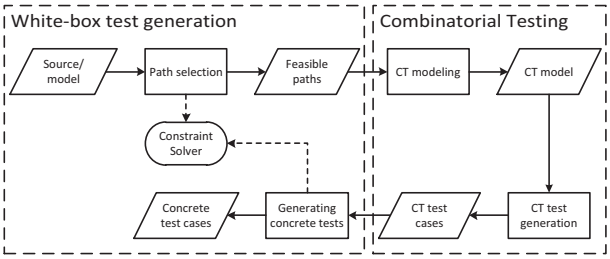

Fig. 6: Overview of our approach

A notable fact is that some boundary conditions can be satisfied at the same time, which means a test case can cover multiple boundaries at the same time. To reduce the number of test cases for BVA, we apply a combinatorial test generation technique called AETG-SAT [9]. The overview of our approach is illustrated in Figure 6. Our approach adds additional processes after each path is selected in normal white-box test generation. For each path, we build a CT model for the path condition, and use a CT test generation tool to generate several sets of compatible boundary conditions that can be satisfied at the same time. Finally, these constraint sets are converted into concrete test cases using the constraint solver. In this way, BVA can be achieved without affecting the original path selection strategy for the structural coverage criterion.

A. Constructing the CT Model

Suppose we obtain a path condition $\mathcal{P}$ during white-box test generation. Each comparison predicate of $\mathcal{P}$ has two off-by-one mutants, as shown in Table III.

Here we require each mutant of each predicate to be covered if possible. To do this, for each mutant c' of comparison predicate c, we build a CT parameter $p_{c'}$ having two values $\{0, 1\}$:

- If $p_{c'}$ takes the value of 0, the boundary need not to be covered;
- If $p_{c'}$ takes the value of 1, the boundary value need to be covered, and a constraint $\neg \mathcal{P}_{c,c'}$ is added to the constraint set.

The coverage requirement is, for each mutant c' of comparison predicate c, a 1-way seed combination is added requiring "$p_{c'} : 1$" to be covered if possible. The covering strength does not need to be specified – all the coverage requirements have already been specified by the seeds.

There are constraints in this model. Firstly, all test cases should follow the current path, thus $\mathcal{P}$ is in the constraint set.

Additionally, for each mutant c' of predicate c, if $p_{c'}$ takes the value of 1, then $\neg\mathcal{P}_{c,c'}$ must be true.

Unlike traditional CT constraint handling, where the constraints are static, our work uses an incremental way of constraint handling. We maintain a constraint set, which is initialized as $\{\mathcal{P}\}$. When some assignment "$p_{c'} : 1$" is made, a corresponding constraint $\neg\mathcal{P}_{c,c'}$ will be added to the constraint set.

An important fact is that, in many cases, the boundary conditions of many different constraints can be covered at the same time. Therefore by applying CT on this model, we can cover all these boundary conditions with a small number of test cases.

B. Combinatorial test generation

After the model is constructed, the next step is to generate a covering array covering all the seed combinations. Here, we use a widely used CT test generation technique called the AETG algorithm [8]. Cohen et al. proposed a modified version of AETG called AETG-SAT [9], which can handle constraints. The general process of the AETG-SAT algorithm is shown in Algorithm 1.

Input: Input parameters, their value domains, constraints and coverage requirement
Output: Test suite T meeting the coverage requirement

```
1  Initialize Π as the set of all parameter combinations as
   specified in the coverage requirement;
2  while Π ≠ ∅ do
3      τ* = empty test case;
4      for i ∈ {1, ..., M} do
5          Generate a new test case τ, which satisfies the
           constraints and covers as many uncovered
           combinations in Π as possible;
6          if τ covers more uncovered combinations than τ* then
7              τ* = τ;
8          end
9      end
10     T = T ∪ {τ*};
11     Remove the combinations covered by τ* from Π;
12 end
13 return T;
```

Algorithm 1: A simplified AETG-SAT algorithm

During the test generation process, the algorithm first initializes Π as the set of all parameter combinations required to be covered as specified by the coverage requirement. We call these combinations *target combinations*, which are specified by the seed combinations in our work. Then the algorithm tries to generate test cases to cover as many target combinations as possible. For generating each test case, the algorithm proceeds in a greedy manner. It generates M candidate test cases (denoted by τ^*) by assigning values to parameters while maximizing the number of newly covered target combinations. Then it selects the best candidate as the next test case (denoted by τ), adds it to the test suite T, and removes all the newly covered target combinations from Π. In our work, the number of test cases M is fixed to 5[1]. When some assignment is done, we may need to check whether a partial/complete assignment satisfies the constraints. Thus, we pass the partial/complete assignment to the constraint solver used in white-box test generation, and the solver will return whether the assignment satisfies the constraints. If the result is infeasible, then the assignment is rejected, and the algorithm tries another value for assignment. Please note our incremental constraint processing as described in Section IV-A. Besides, some target combinations (boundary conditions) may never be covered since they conflict with the constraints. CT test generation will guarantee to cover all possible target combinations.

After the test cases (choices of boundary conditions) are generated by CT, we use the constraint solver to generate a valid input for each CT test case. If there are no CT test cases generated, it means that all the boundaries cannot be covered. In this case, we directly call the constraint solver once using $\mathcal{P}$ to generate an input in the subspace.

```
1  int function(bool w, int x, int y, int z) {
2    if (!w) { return 0; }
3    if (x > 1) {
4      if (x < 10) {
5        if (y >= z) {
6          return 1;
7        }
8      }
9    }
10   return 2;
11 }
```

Fig. 7: An example code snippet

TABLE IV: Generated CT model

Predicate c	Parameters $p_{c'}$	Values	Mutated path condition $\mathcal{P}_{c,c'}$
w	NA	NA	NA
$x > 1$	$p_{x>2}$	$\{0,1\}$	$w \wedge \underline{x > 2} \wedge x < 10 \wedge y \geq z$
	$p_{x>0}$	$\{0,1\}$	$w \wedge \underline{x > 0} \wedge x < 10 \wedge y \geq z$
$x < 10$	$p_{x<11}$	$\{0,1\}$	$w \wedge x > 1 \wedge \underline{x < 11} \wedge y \geq z$
	$p_{x<9}$	$\{0,1\}$	$w \wedge x > 1 \wedge \underline{x < 9} \wedge y \geq z$
$y \geq z$	$p_{y\geq z+1}$	$\{0,1\}$	$w \wedge x > 1 \wedge x < 10 \wedge \underline{y \geq z+1}$
	$p_{y\geq z-1}$	$\{0,1\}$	$w \wedge x > 1 \wedge x < 10 \wedge \underline{y \geq z-1}$

C. An Example

Here we give an example to illustrate how our method works. See the code snippet shown in Figure 7.

Suppose one generated path in white-box test generation reaches line 6. The path condition $\mathcal{P}$ is

$$w \wedge x > 1 \wedge x < 10 \wedge y \geq z,$$

and has the following predicates:

$$\{w, x > 1, x < 10, y \geq z\}$$

The generated CT model is as shown in Table IV. Note that in the boundary conditions, the underlined predicates are replaced by their mutants.

[1]The original algorithm uses $M = 50$, which makes it more possible to find better test cases, in order to reduce the final test suite size. Since our CT model is very simple, we use 5 to reduce the computational cost.

There are 6 seed (target) combinations in total:

$$\left\{ \begin{array}{lll} p_{x>2}:1, & p_{x<11}:1, & p_{y\geq z+1}:1, \\ p_{x>0}:1, & p_{x<9}:1, & p_{y\geq z-1}:1 \end{array} \right\}.$$

The initial constraint set is $\{\mathcal{P}\}$:

$$\{w \wedge x > 1 \wedge x < 10 \wedge y \geq z\}$$

Here we set the number of candidates for each test case M as 1 for simplicity. Suppose when generating the first test case, the current partial assignment is:

$$p_{x>2}:1, p_{x>0}:0.$$

The current constraint set is:

$$\left\{ \begin{array}{c} w \wedge x > 1 \wedge x < 10 \wedge y \geq z, \\ \neg(w \wedge \underline{x > 2} \wedge x < 10 \wedge y \geq z) \end{array} \right\}. \quad (1)$$

Suppose the next assignment attempt is $p_{x<9} : 1$. The algorithm adds the corresponding constraint into the constraint set, and checks the satisfiability:

$$\left\{ \begin{array}{c} w \wedge x > 1 \wedge x < 10 \wedge y \geq z, \\ \neg(w \wedge \underline{x > 2} \wedge x < 10 \wedge y \geq z), \\ \neg(w \wedge x > 1 \wedge \underline{x < 9} \wedge y \geq z) \end{array} \right\}. \quad (2)$$

However, this set of constraints is infeasible, since the first two constraint force $w \wedge x = 2 \wedge x < 10 \wedge y \geq z$, which conflicts with the new constraint. Therefore, this attempt fails, and $p_{x<9}$ must take value 0. The constraint set is rolled back to (1).

The remaining assignment process is as follows:

- Try assignment $p_{x<11} : 1$, the constraint set is infeasible, so this assignment is undone, and $p_{x<11}$ takes value 0;
- Try assignment $p_{y\geq z+1} : 1$, the constraint set is feasible, so this assignment is accepted.
- Try assignment $p_{y\geq z-1} : 1$, the constraint set is infeasible, so this assignment is undone, and $p_{y\geq z-1}$ takes value 0.

Finally, the generated CT test case is:

$$\left\{ \begin{array}{lll} p_{x>2}:1, & p_{x<11}:0, & p_{y\geq z+1}:1, \\ p_{x>0}:0, & p_{x<9}:0, & p_{y\geq z-1}:0 \end{array} \right\}.$$

The combinations covered by this test case are removed, so the set of remaining target combinations is:

$$\left\{ \begin{array}{lll} \cancel{p_{x>2}:1}, & p_{x<11}:1, & \cancel{p_{y\geq z+1}:1}, \\ p_{x>0}:1, & p_{x<9}:1, & p_{y\geq z-1}:1 \end{array} \right\}.$$

The generation of the second test case is similar, so we do not show the details. The second test case is:

$$\left\{ \begin{array}{lll} p_{x>2}:0, & p_{x<11}:0, & p_{y\geq z+1}:0, \\ p_{x>0}:0, & p_{x<9}:1, & p_{y\geq z-1}:0 \end{array} \right\}.$$

The combination covered by this test case is removed, so the set of remaining target combinations is:

$$\left\{ \begin{array}{lll} \cancel{p_{x>2}:1}, & p_{x<11}:1, & \cancel{p_{y\geq z+1}:1}, \\ p_{x>0}:1, & \cancel{p_{x<9}:1}, & p_{y\geq z-1}:1 \end{array} \right\}.$$

There are no more test cases since all remaining target combinations cannot be covered. In the end, we convert all the CT test cases into concrete test cases by calling the constraint solver again.

The first test case:

$$\left\{ \begin{array}{lll} p_{x>2}:1, & p_{x<11}:0, & p_{y\geq z+1}:1, \\ p_{x>0}:0, & p_{x<9}:0, & p_{y\geq z-1}:0 \end{array} \right\}$$
$$\Downarrow$$
$$\left\{ \begin{array}{c} w \wedge x > 1 \wedge x < 10 \wedge y \geq z, \\ \neg(w \wedge \underline{x > 2} \wedge x < 10 \wedge y \geq z), \\ \neg(w \wedge x > 2 \wedge x < 10 \wedge \underline{y \geq z+1}), \end{array} \right\}$$
$$\Downarrow$$
$$w \wedge x = 2 \wedge y = 1 \wedge z = 1;$$

The second test case:

$$\left\{ \begin{array}{lll} p_{x>2}:0, & p_{x<11}:0, & p_{y\geq z+1}:0, \\ p_{x>0}:0, & p_{x<9}:1, & p_{y\geq z-1}:0 \end{array} \right\}$$
$$\Downarrow$$
$$\left\{ \begin{array}{c} w \wedge x > 1 \wedge x < 10 \wedge y \geq z, \\ \neg(w \wedge x > 1 \wedge \underline{x < 9} \wedge y \geq z, \end{array} \right\}$$
$$\Downarrow$$
$$w \wedge x = 9 \wedge y = 2 \wedge z = 1.$$

V. Discussion

A. Fault detection ability

After applying our BVA approach to traditional white-box test generation techniques, we can effectively detect boundary faults, but how will the detection ability of other kind of faults be affected? The answer is the fault detection ability for a specific fault type will increase or stay unchanged. The reason is that if we apply our approach to white-box test generation technique to meet a certain structural coverage criterion, the path selection strategy is unchanged, and we derive at least one test case for each selected path. Therefore, our test suite also meets the original structural coverage. Our test suite can detect the fault types which the original test suite can detect. This will be studied in Section VI.

B. Further reducing the BVA cost

The goal of applying BVA to white-box test generation is to cover the boundary values with as low additional cost as possible. We have used CT to reduce the number of test cases. However, there are still possible reductions. Usually, some paths may share the same predicates. We consider that these shared predicates should have the same intentions. Therefore, each boundary only needs to be tested once. If some boundary of a predicate is covered once, we mark it as covered, and this boundary needs not to be covered in other paths. Besides, if some predicate appears in a path condition multiple times, all its occurrences should be considered together, and their mutants should be applied simultaneously.

C. About SMT solvers

SMT (Satisfiability Modulo Theories) solvers are used in white-box test generation to check path condition satisfiability and generate feasible solutions. Due to their different internal strategies, different SMT solvers may produce different feasible solutions. Some SMT solvers are more likely to

produce solutions close to the boundaries, while some others are more likely to produce solutions far away from boundaries. By applying our technique, the boundary coverage can be effectively increased regardless of which SMT solver is used.

D. Complexity

The calculation in AETG-SAT is very lightweight, while most of the time is spent on checking the feasibility of parameter assignments. Here we measure the complexity by estimating the number of solver calls for each path. Suppose the number of comparison predicates in $\mathcal{P}$ is n. The number of mutants will be $2n$. Thus the maximum number of test cases is $2n$, and the maximum number of solver calls for each test case is $2Mn$ (here M is the number of candidates in Algorithm 1). Therefore, the worst case complexity for each path is $4Mn^2$ solver calls. In the best case, no boundary conditions can be covered. The cost of determining the infeasibility of all boundary conditions is $2n$, and then the solver is called again to generate one test for the path, making the total number of solver calls be $2n + 1$. In a word, the number of solver calls for each path is $2n + 1 \sim 4Mn^2$.

E. On multiple boundary faults

In our work, we assume that the boundary failures are independent. However, it is possible that multiple boundary failures correlate with each other. Suppose we have path condition $(a > 0) \vee (a > 0)$, and the two predicates are not considered as the same, then mutating any of the two predicates to $a > 1$ will not change the semantics. Therefore no boundary value can be generated for mutant $a > 1$. Only by mutating both of them at the same time will change the semantics, and the boundary value can be generated. However, when the path condition has too many predicates, it is extremely difficult to determine to mutate which predicates simultaneously, because we need to consider every combination of simultaneous mutations.

F. Relation with logical boundaries

There are plenty of works focusing on logical coverage, such as condition coverage, decision coverage and modified condition/decision coverage (MC/DC), among which MC/DC coverage is a coverage criterion for logical boundaries, and has been widely researched [2], [7], [17], [19], [21], [22], [29], [32]. The boundaries for comparison predicates discussed in this paper are very similar with logical boundaries. Our definition of boundary conditions uses the concept of "determination", which is borrowed from GACC, which is a version of MC/DC. MC/DC requires for each clause in a decision, there exists a pair of inputs having different values on the clause and the same values for all other clauses, and making the decision evaluate to different results. In future work, we will seek for possible extensions for MC/DC.

VI. Evaluation

To evaluate our approach, we selected several programs used in existing literature. The descriptions of these subject programs are shown in Table V.

TABLE V: Experimental programs

Name	Description
abs	Absolute value
calDate [18]	Convert the special date into a Julian date
complexCheck [18]	A custom program with complex logic
findMiddle [18]	Find the middle number among three numbers
isLeapYear	Determine whether a year is a leap year
triType [31]	The type of a triangle
spaceManage [32]	An aircraft program provided by our industry partner
nextDate [3]	Calculate the following date of the given day
showHand [32]	A popular card game Show Hand
tcas [11]	Aircraft collision avoidance system

To study the effectiveness of our BVA approach, we generated 4 sets of test cases, in order to compare their boundary fault detection rate. The 4 test case sets are:

- C1: branch coverage only;
- C2: apply the BVA method by Jamrozik et al. [22] to branch coverage. Here one individual test case is generated for each boundary condition, i.e. using base choice (BC) strategy [1];
- C3: apply our BVA method to branch coverage, but do not use CT to reduce the number of test cases. Here base choice (BC) strategy is used;
- C4: apply our BVA method to branch coverage, and use CT to reduce the number of test cases.

In our evaluation, the SMT solver has multiple choices: Z3 [27], Yices [12], Boolector [5], CVC4 [4], STP [14]. We encapsulated these solvers into a uniform interface, thus we can use them transparently. The choice of SMT solver does not affect the number of test cases and the number of solver calls, but it can affect the covered boundaries for C1 and C2, as well as the execution time.

Some measurements of the 4 test sets are shown in Table VI. We can see that the number of test cases for the 4 test sets follows $C1 < C4 < C2 \approx C3$. C2 and C3 has about 3 to 10 times of solver calls compared with C1, and C4 has about 10 to 50 times of solver calls compared with C1, i.e. $C1 < C2 \approx C3 < C4$. The test generation time follows $C1 < C2 \approx C3 < C4$. Although C4 requires relatively more solver calls and longer execution time, but it is still acceptable. We found that most of the additional time is spent on constraint solving. The improvements described in AETG-SAT [9] can be used to reduce the number of solver calls significantly.

We use mutation testing to study the fault detection rate of test sets. Firstly, several faults are injected (seeded) into each program. Each seeded fault yields a faulty version. For each program and test set, we run the whole test set on each faulty version, and count the number of killed mutations. Here two sets of mutations are injected: the first set contains off-by-one bugs (OBOB, where we add/subtract 1 to the right-hand side of comparison predicates) and off-by-many bugs (OBMB, where we add/subtract a random value from 2 to 1000 to the right-hand side of comparison predicates) which are generated by using a manually crafted `awk`[2] script; the second set is generated by the Milu mutation generation tool

[2] `http://www.gnu.org/software/gawk/`

TABLE VI: Measurements for different test generation methods

Program	#P[a]|#B[b]	# test cases				# solver calls				Time (in seconds, using Z3)			
		C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
abs	1|1	2	2	2	2	2	5	5	25	0.34	0.361	0.356	0.416
calDate	10|6	7	20	20	14	19	55	65	1029	0.614	0.723	0.759	4.159
complexCheck	13|4	5	26	26	16	8	45	52	1512	0.496	0.619	0.63	5.779
findMiddle	14|5	6	28	28	12	10	77	77	1341	0.528	0.749	0.752	5.288
isLeapYear	4|2	3	8	7	5	4	15	17	230	0.408	0.452	0.454	1.26
triType	17|8	8	31	30	14	54	138	142	1316	0.968	1.203	1.208	5.084
spaceManage	16|8	7	31	30	16	56	121	126	2212	0.948	1.166	1.19	8.963
nextDate	24|10	9	42	42	23	48	162	173	3719	0.893	1.267	1.28	13.524
showHand	36|12	13	74	72	32	24	203	232	3722	0.791	1.443	1.567	14.041
tcas	37|21	10	61	55	13	136	266	325	2918	1.826	2.282	2.466	12.5

[a] #P: number of predicates;
[b] #B: number of conditional branch statements.

[23], and contains various types of faults. In our experiments, 8 types of faults are injected by Milu, as shown in Table VII (other types of faults in Milu are not applicable to the subject programs). We use two ways to determine whether a fault is killed:

- Output kill: at least one test case has a different output with the correct version;
- Path kill: at least one test case has a different execution path with the correct version.

The results are shown in Table VIII–XI. Table VIII and Table IX show the number of output kills and path kills for OBOB and OBMB respectively. Please compare the results between columns. For example, In Table VIII, the entry at triType:C1:Boolector is "15|17", and the entry at triType:C4:Boolector is "28|31". This means that the test suite in $C4$ have 13 (28-15) more output kills and 14 (31-17) more path kills than $C1$. For Yices and Boolector, the results for `isLeapYear` and `nextDate` are not available, since they have modulo operators which the two solvers do not support.

From Table VIII we can see that C1 has the worst output/path kill rate. C2 is relatively better. C3 and C4 have the best output/path kill rate. In a word, the fault detection rate of OBOB follows $C1 < C2 < C3 \approx C4$. The results in Table IX also shows that the fault detection rate for OBMB follows $C1 < C2 < C3 \approx C4$. Since OBMB is easier to detect than OBOB, the fault detection rate is relatively higher than Table VIII.

When comparing different solvers, we can see that different solvers have different output/path kills in C1. The test cases generated by Boolector and STP are less likely to be boundary values. For C2, C3 and C4 which are BVA-aware, the test cases generated by different solvers have almost the same likelihood to be boundary values.

Table X and Table XI show the kill rate for the faults injected by Milu, which is categorized by different types of faults. Each cell shows the average fault detection rate for the five solvers (except for the cases where the program has modulo operators which Yices and Boolector do not support).

From Table X and Table XI, we can see that the output/path kill rate for different types of faults varies. The following factors may affect the output/path kill rate.

A: **Some faults of this type are irrelevant to boundary faults.** In this case, the number and the diversity of test cases dominate. We know that $C2$, $C3$ and $C4$ also satisfy branch coverage just as $C1$ does. Besides, they have more test cases, and their test case diversity is better since each test cases covers specific boundaries. Therefore, the fault detection ability should follow $C1 < C2 \approx C3 \approx C4$;

B: **Some faults of this type are relevant to boundary faults.** In this case, the fault detection ability should follow $C1 < C2 < C3 \approx C4$, just as we discussed;

C: **The fault type is severe, and is very easy to detect.** In this case, the fault detection ability for all the four test suites should be very large (close to 100%), so the difference is insignificant;

D: **The fault type is relevant to logical faults.** Since our definition for boundary condition uses the concept of "determination" from GACC, it implies some logical boundaries, thus the fault detection ability for this type of faults follow $C1 < C2 < C3 \approx C4$;

TABLE VII: Fault types injected by Milu and factors affecting the output/path kill rate

Name	Description	Factors
ABS	abs function insersion	A
CRCR	constant replacement	B
OAAN	arithmetic operator replacement	A, B
OCNG	if/while expression negation	C
OLLN	and/or replacement	A, D
OLNG	and/or operator negation and and/or operand negation	C, D
ORRN	relational operator replacement	A, B
UOI	unary operator (increment/decrement) insertion	A, B

The output/path kill rate for different fault types can be affected by one or more factors described above, which is also shown in Table VII. In general, the output/path kill rate for C2, C3 and C4 is higher than C1. Note that there are only 8 mutant types in our experiments. The effects on fault detection ability for other fault types need further study.

VII. Related work

White-box testing is a commonly used software testing technique to ensure the high quality of software, while the source code is available. Specifically, how to generate the test data with high coverage and high fault-detection capability is one of the most important issues in white-box testing. Manual generation of test suites can hardly achieve high coverage and it is often time-consuming. Therefore, automated

TABLE VIII: Fault detection ability for OBOBs injected by us*

Program	# of mutants (faulty versions)	C1					C2				
		Z3	Yices	Boolector	CVC4	STP	Z3	Yices	Boolector	CVC4	STP
abs	2	1|2	1|2	0|0	1|2	0|1	1|2	1|2	1|2	1|2	1|2
calDate	20	9|10	8|10	9|9	9|10	10|10	13|14	12|14	13|15	13|14	14|15
complexCheck	26	12|12	10|10	8|8	10|10	9|9	24|24	21|21	14|14	16|16	17|17
findMiddle	28	14|16	14|16	11|11	12|14	8|10	20|24	20|24	24|28	16|20	18|22
isLeapYear	8	6|6	-|-	-|-	4|4	4|4	7|7	-|-	-|-	7|7	7|7
triType	34	25|25	23|25	15|17	22|22	18|18	28|28	28|29	22|24	23|24	21|21
spaceManage	32	14|20	13|19	10|10	14|19	13|17	25|31	23|28	22|24	24|28	25|31
nextDate	48	18|28	-|-	-|-	13|25	14|24	26|48	-|-	-|-	20|45	21|46
showHand	72	55|55	53|53	43|43	55|55	56|56	65|65	57|57	60|60	59|59	60|60
tcas	74	41|55	38|54	25|31	40|54	30|40	41|58	40|56	47|58	40|55	38|54
Program	# of mutants (faulty versions)	C3					C4				
		Z3	Yices	Boolector	CVC4	STP	Z3	Yices	Boolector	CVC4	STP
abs	2	1|2	1|2	1|2	1|2	1|2	1|2	1|2	1|2	1|2	1|2
calDate	20	19|20	18|20	18|20	19|20	19|20	19|20	18|20	18|20	19|20	19|20
complexCheck	26	26|26	26|26	26|26	26|26	26|26	26|26	26|26	26|26	26|26	26|26
findMiddle	28	24|28	24|28	24|28	24|28	24|28	24|28	24|28	24|28	24|28	24|28
isLeapYear	8	8|8	-|-	-|-	8|8	8|8	8|8	-|-	-|-	8|8	8|8
triType	34	31|31	30|31	28|31	29|31	31|31	31|31	28|31	28|31	28|31	31|31
spaceManage	32	25|32	26|32	25|32	25|32	25|32	25|32	26|32	25|32	25|32	25|32
nextDate	48	26|48	-|-	-|-	20|48	21|48	26|48	-|-	-|-	20|48	21|48
showHand	72	72|72	72|72	72|72	72|72	72|72	72|72	72|72	72|72	72|72	72|72
tcas	74	41|58	41|58	44|58	41|58	41|58	41|58	41|58	41|58	41|58	41|58

* Each cell in this table is in the form of "a|b", where a is the number of output kills and b is the number of path kills.

TABLE IX: Fault detection ability for OBMBs injected by us*

Program	# of mutants (faulty versions)	C1					C2				
		Z3	Yices	Boolector	CVC4	STP	Z3	Yices	Boolector	CVC4	STP
abs	2	1|2	1|2	0|0	1|2	0|1	1|2	1|2	1|2	1|2	1|2
calDate	20	14|15	13|15	13|13	14|15	15|15	16|17	15|17	16|18	16|17	17|18
complexCheck	26	14|15	13|14	9|10	14|15	14|15	24|25	21|22	15|15	17|17	18|18
findMiddle	28	16|22	15|18	11|13	13|19	9|12	23|28	23|27	24|28	20|26	19|26
isLeapYear	8	6|6	-|-	-|-	4|4	3|3	6|6	-|-	-|-	6|6	6|6
triType	34	26|26	24|26	16|18	23|23	18|18	25|25	26|32	20|26	23|23	20|20
spaceManage	32	15|20	14|19	10|10	15|19	14|19	25|31	23|28	22|24	24|28	25|31
nextDate	48	18|28	-|-	-|-	16|24	16|24	22|48	-|-	-|-	20|44	20|46
showHand	72	55|55	53|53	49|49	54|54	54|54	65|65	57|57	63|63	59|59	60|60
tcas	74	42|64	40|63	24|38	40|63	37|58	42|67	42|65	44|58	42|65	41|64
Program	# of mutants (faulty versions)	C3					C4				
		Z3	Yices	Boolector	CVC4	STP	Z3	Yices	Boolector	CVC4	STP
abs	2	1|2	1|2	1|2	1|2	1|2	1|2	1|2	1|2	1|2	1|2
calDate	20	19|20	18|20	18|20	19|20	19|20	19|20	18|20	18|20	19|20	19|20
complexCheck	26	26|26	26|26	26|26	26|26	26|26	26|26	26|26	26|26	26|26	26|26
findMiddle	28	24|28	24|28	24|28	24|28	24|28	24|28	24|28	24|28	24|28	24|28
isLeapYear	8	8|8	-|-	-|-	8|8	8|8	8|8	-|-	-|-	8|8	8|8
triType	34	32|32	26|32	26|32	32|32	32|32	32|32	26|32	26|32	26|32	32|32
spaceManage	32	25|32	26|32	25|32	25|32	25|32	25|32	26|32	25|32	25|32	25|32
nextDate	48	22|48	-|-	-|-	20|48	20|48	22|48	-|-	-|-	20|48	20|48
showHand	72	72|72	72|72	72|72	72|72	72|72	72|72	72|72	72|72	72|72	72|72
tcas	74	42|67	42|67	42|66	42|67	42|67	42|67	42|67	42|66	42|67	42|66

* Each cell in this table is in the form of "a|b", where a is the number of output kills and b is the number of path kills.

white-box test generation has been proposed to reduce human efforts. There exist a number of approaches, such as symbolic execution [35], dynamic symbolic execution [20] and heuristic algorithms [3]. However, most of them do not consider BVA, which is the main topic of this paper.

There are several related works on applying BVA to white-box test generation. Pandital et al. [29] proposed an automated instrumentation approach, which adds several auxiliary statements to the program to guide the path selection strategy for BVA or MC/DC coverage. However the instrumented code may influence the path selection strategy. Jamrozik et al. [22] proposed a general method which applies different strategies to each generated path to generate test suites for different purposes such as BVA, mutation analysis, logical coverage, etc. They focused on the general method, but did not thoroughly discuss the strategy of selecting test cases for BVA. The generated inputs may not be true boundary values, and the number of test cases may be relatively high.

Combinatorial testing has been studied many years. Many test generation methods have been proposed, such as AETG [8] and IPO [26]. Constraint handling is an important aspect in CT test generation. Algorithms such as AETG-SAT [9], PICT [10], CASA [15], [16], IPOG-C [34] and Cascade [38], [37] focus on test generation with the presence of constraints. More details of different aspects of CT can be found in the survey by Nie and Leung [28] and the book by Kuhn et al. [25]. Zhang et al. wrote a comprehensive survey of combinatorial test generation techniques [36]. Despite all these, the application of CT is still limited to black-box testing. Our practice successfully combined CT into white-box test generation to reduce the number of test cases while still guaranteeing the boundary coverage.

TABLE X: Output kill rate (in percentage) for faults injected by Milu [23] with different test suites

Mutant type	ABS				CRCR				OAAN				OCNG			
	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
abs	33	33	33	33	62	70	70	70	100	100	100	100	100	100	100	100
calDate	41	45	48	48	82	86	93	93	94	94	94	94	100	100	100	100
complexCheck	33	43	51	50	58	77	100	100	46	81	100	100	100	100	100	100
findMiddle	25	53	55	46	-	-	-	-	-	-	-	-	100	100	100	100
isLeapYear	25	25	25	25	70	82	95	95	44	58	78	78	100	100	100	100
triType	35	44	45	39	88	90	93	89	76	83	87	84	100	100	100	100
spaceManage	27	39	38	38	71	97	97	97	85	100	100	100	100	100	100	100
nextDate	27	30	30	30	41	47	47	47	56	56	56	56	100	100	100	100
showHand	44	49	50	50	74	79	100	100	63	68	81	81	100	100	100	100
tcas	26	30	30	27	53	59	57	57	52	62	60	50	72	82	81	81
Mutant type	OLLN				OLNG				ORRN				UOI			
	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
abs	-	-	-	-	-	-	-	-	64	60	60	60	67	67	67	67
calDate	20	95	100	100	100	100	100	100	72	78	88	88	62	70	71	71
complexCheck	73	93	100	100	100	100	100	100	65	73	83	83	45	73	79	79
findMiddle	84	89	89	89	92	100	100	100	59	85	94	94	82	89	90	90
isLeapYear	100	100	100	100	89	100	100	100	63	75	85	85	69	88	88	88
triType	37	66	83	83	100	100	100	100	76	86	86	83	46	59	71	70
spaceManage	25	70	75	75	58	96	100	100	42	66	70	70	40	56	59	59
nextDate	38	38	38	38	50	51	51	51	37	40	40	40	62	76	76	76
showHand	38	58	100	100	100	100	100	100	78	84	89	89	80	86	86	86
tcas	44	44	44	44	73	77	75	75	47	51	50	50	31	43	43	42

TABLE XI: Path kill rate (in percentage) for faults injected by Milu [23] with different test suites

Mutant type	ABS				CRCR				OAAN				OCNG			
	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
abs	40	33	33	33	86	100	100	100	100	100	100	100	100	100	100	100
calDate	43	48	50	50	83	89	96	96	94	94	94	94	100	100	100	100
complexCheck	33	43	51	50	58	77	100	100	46	81	100	100	100	100	100	100
findMiddle	33	62	69	58	-	-	-	-	-	-	-	-	100	100	100	100
isLeapYear	25	25	25	25	70	82	95	95	44	58	78	78	100	100	100	100
triType	38	46	46	42	92	95	99	99	76	83	87	84	100	100	100	100
spaceManage	31	44	45	44	74	98	100	100	85	100	100	100	100	100	100	100
nextDate	31	46	46	46	63	87	91	91	72	83	100	100	100	100	100	100
showHand	44	49	50	50	74	79	100	100	63	68	81	81	100	100	100	100
tcas	45	48	50	46	71	74	76	76	95	100	100	100	100	100	100	100
Mutant type	OLLN				OLNG				ORRN				UOI			
	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
abs	-	-	-	-	-	-	-	-	80	80	80	80	67	67	67	67
calDate	20	95	100	100	100	100	100	100	74	80	90	90	62	70	71	71
complexCheck	73	93	100	100	100	100	100	100	65	73	83	83	45	73	79	79
findMiddle	100	100	100	100	97	100	100	100	71	89	100	100	83	89	90	90
isLeapYear	100	100	100	100	89	100	100	100	63	75	85	85	69	88	88	88
triType	54	77	100	100	100	100	100	100	83	88	88	85	47	60	72	72
spaceManage	42	92	100	100	100	100	100	100	70	86	88	88	49	62	68	68
nextDate	64	100	100	100	99	99	100	100	80	90	92	92	79	81	84	84
showHand	38	58	100	100	100	100	100	100	78	84	89	89	80	86	86	86
tcas	100	100	100	100	100	100	100	100	74	76	78	78	47	58	60	60

VIII. Conclusion

Software is usually problematic at extreme inputs, a.k.a. boundary values. Boundary value analysis (BVA) focuses on testing software against boundary values to detect potential defects. This technique is usually used in black-box testing. However there is also a need for BVA in white-box testing. In our work, we proposed a new definition for boundary conditions based on mutation, and gave a new test selection strategy for covering the boundary conditions, while the original structural coverage is preserved. We applied combinatorial testing technique to make each test case cover as many boundaries as possible, in order to reduce the total number of test cases. Experimental results show that our approach can achieve high detection rate for boundary faults, and the number of test cases can be effectively reduced by applying CT.

The future work includes: (1) reducing the constraint solving cost using the improvements of AETG-SAT algorithm; (2) discover the possibility of using CT to reduce the number of test cases in other mutation-based test generation techniques; (3) finding relations of comparison predicate boundaries with logical boundaries, which is related to MC/DC, and extending our work to MC/DC test generation if possible.

Acknowlegement

This work is supported in part by National Basic Research (973) Program of China (No. 2014CB340701), National Natural Science Foundation of China (Grant No. 91418206, 91118007). We thank Cunjing Ge and Feifei Ma for their helpful comments.

References

[1] P. Ammann, and J. Offutt. "Using formal methods to derive test frames in category-partition testing". In *Proc. of the Ninth Annual Conference on Computer Assurance (COMPASS '94)*, pp. 69–79. IEEE.

[2] P. Ammann, J. Offutt, and H. Huang. "Coverage criteria for logical expressions". In *Proc. of the 14th International Symposium on Software Reliability Engineering (ISSRE '03)*, pp. 99–107. IEEE.

[3] Z. Awedikian, K. Ayari, and G. Antoniol. “MC/DC automatic test input data generation”. In *Proc. of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO ’09)*, pp. 1657–1664. ACM.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. “CVC4”. In *Proc. of the 23rd International Conference on Computer Aided Verification(CAV ’11)*, pp. 171–177. Springer Berlin Heidelberg.

[5] R. Brummayer, and A. Biere. “Boolector: An efficient SMT solver for bit-vectors and arrays”. In *Proc. of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’09)*, pp. 174–177. Springer Berlin Heidelberg.

[6] C. Cadar, D. Dunbar and D. Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI ’08)*, pp. 209–224.

[7] J. R. Chang, and C. Y. Huang. “A study of enhanced MC/DC coverage criterion for software testing”. In *Proc. of the 31st Annual International Computer Software and Applications Conference (COMPSAC ’07)*, pp. 457–464. IEEE.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. “The AETG system: An approach to testing based on combinatorial design”. *IEEE Transactions on Software Engineering*, 23(7): 437–444, 1997.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. “Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach”. *IEEE Transactions on Software Engineering*, 34(5): 633-650, 2008.

[10] J. Czerwonka. “Pairwise testing in the real world”. In *Proc. of the 24th Pacific Northwest Software Quality Conference (PNSQC 06)*, pp. 419–430.

[11] H. Do, S. Elbaum, and G. Rothermel. “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact”. *Empirical Software Engineering*, 10(4): 405–435, 2005.

[12] B. Dutertre and L. de Moura. “A fast linear-arithmetic solver for DPLL(T)”, In *Proc. of the 18th International Conference on Computer Aided Verification (CAV ’06)*, pp. 81–94. Springer Berlin Heidelberg.

[13] C. Fang, Z. Chen, and B. Xu. “Comparing logic coverage criteria on test case prioritization”. *Science China Information Sciences*, 55(12): 2826–2840, 2012.

[14] V. Ganesh, and D. L. Dill. “A decision procedure for bit-vectors and arrays”. In *Proc. of the 19th International Conference on Computer Aided Verification (CAV ’07)*, pp. 519–531. Springer Berlin Heidelberg.

[15] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. “An improved meta-heuristic search for constrained interaction testing”. In *Proc. of the 1st International Symposium on Search Based Software Engineering (SSBSE ’09)*, pp. 13–22. IEEE.

[16] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. “Evaluating improvements to a meta-heuristic search for constrained interaction testing”. *Empirical Software Engineering*, 16(1): 61–102, 2011.

[17] K. Ghani, and J. A. Clark. “Automatic test data generation for multiple condition and MCDC coverage”. In *Proc. of the Fourth International Conference on Software Engineering Advances (ICSEA ’09)*, pp. 152–157. IEEE.

[18] K. Ghani, *Searching for Test Data*, PhD Thesis, The University of York, 2009.

[19] S. Godboley, G. S. Prashanth, D. P. Mohapatro, and B. Majhi. “Increase in Modified Condition/Decision Coverage using program code transformer”. In *Proc. of the IEEE 3rd International Advance Computing Conference (IACC ’13)*, pp. 1400–1407. IEEE.

[20] P. Godefroid, N. Klarlund and K. Sen. “DART: Directed Automated Random Testing”. In *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’05)*, pp. 213–223. ACM.

[21] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, L. K. Rierson, “A practical tutorial on modified condition/decision coverage”. *Technical Report*, NASA/TM-2001-210876. NASA, 2011.

[22] K. Jamrozik, G. Fraser, N. Tillman, and J. de Halleux. “Generating test suites with augmented dynamic symbolic execution”. In *Proc. of the 7th International Conference Tests and Proofs (TAP ’13)*, pp. 152–167. Springer Berlin Heidelberg.

[23] Y. Jia, and M. Harman. “MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language”. In *Proc. of the 3rd Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART ’08)*, pp. 94–98. IEEE.

[24] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. “Boundary coverage criteria for test generation from formal models”. In *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE ’04)*, pp. 139–150. IEEE.

[25] D. R. Kuhn, R. N. Kacker., Y. Lei. *Introduction to Combinatorial Testing*. Chapman and Hall/CRC, London, UK, 2013.

[26] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. “IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing”. *Software Testing, Verification and Reliability*, 18(3): 125–148, 2008.

[27] L. de Moura, and N. Bjørner. “Z3: An efficient SMT solver”. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’08)*, pp. 337–340. Springer Berlin Heidelberg.

[28] C. Nie, and H. Leung. “A survey of combinatorial testing”. *ACM Computing Surveys*, 43(2): 11, 2011.

[29] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. “Guided test generation for coverage criteria”. In *Proc. of 2010 IEEE International Conference on Software Maintenance (ICSM ’10)*, pp. 1–10. IEEE.

[30] K. Sen, D. Marinov and G. Agha. “CUTE: A Concolic Unit Testing Engine for C”. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE ’05)*, pp. 263–272. ACM.

[31] N. Williams, B. Marre, P. Mouy, and M. Roger. “Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis”. In *Proc. of the 5th European Dependable Computing Conference on Dependable Computing (EDCC ’05)*, pp. 281–292. Springer Berlin Heidelberg.

[32] T. Wu, J. Yan, and J. Zhang. “Automatic Test Data Generation for Unit Testing to Achieve MC/DC Criterion”. In *Proc. of the Eighth International Conference on Software Security and Reliability (SERE ’14)*, pp. 118–126. IEEE.

[33] Z. Xu, and J. Zhang. “A Test Data Generation Tool for Unit Testing of C Programs”. In *Proc. of the Sixth International Conference on Quality Software (QSIC ’06)*, pp. 107–116. IEEE.

[34] L. Yu, Y. Lei, M. N. Borazjany, R. N. Kacker, and D. R. Kuhn. “An efficient algorithm for constraint handling in combinatorial test generation”. In *Proc. of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST ’13)*, pp. 242–251. IEEE.

[35] J. Zhang, X. Chen, and X. Wang. “Path-oriented test data generation using symbolic execution and constraint solving techniques”. In *Proc. of the Second International Conference on Software Engineering and Formal Methods (SEFM ’04)*, pp. 242–250. IEEE.

[36] J. Zhang, Z. Zhang, and F. Ma. *Automatic Generation of Combinatorial Test Data*. Springer Berlin Heidelberg, 2014.

[37] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang. “Generating combinatorial test suite using combinatorial optimization”. *Journal of Systems and Software*, 98: 191–207, 2014.

[38] Y. Zhao, Z. Zhang, J. Yan, and J. Zhang. “Cascade: a test generation tool for combinatorial testing”. In *Proc. of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW ’13)*, pp. 267–270. IEEE.