How Effective is Branch-based Combinatorial Testing? An Exploratory Study

Huiyan Wang, Chang Xu*, Jun Sui, Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
cocowhy1013@gmail.com, changxu@nju.edu.cn, smilent_sj@163.com, lj@nju.edu.cn

***Abstract*—Combinatorial testing detects faults by trying different value combinations for program inputs. Traditional combinatorial testing treats programs as black box and focuses on manipulating program inputs (named input-based combinatorial testing or ICT). In this paper, we explore the possibility of conducting combinatorial testing via white-box branch information. Similarly, different combinations of branches taken in an execution are tried to test whether they help detect faults and to what extent. We name this technique branch-based combinatorial testing (BCT). We propose ways to address challenges in realizing BCT, and evaluate BCT with Java programs. The results reported that BCT can effectively detect faults even with low-level combinations, say 3-4 ways, which suggest it to be a strong test adequacy criterion. We also found that our greedy strategy for minimizing test suites reduces over 50% tests for reaching certain way levels, and merging nested branches detects faults more cost-effectively than considering them separately.**

***Index Terms*—Combinatorial testing, testing adequacy criteria**

I. Introduction

Software has become increasingly important in our daily life, and many different kinds of testing techniques have been proposed to assure its quality. One popular testing technique is combinatorial testing, which has been studied for two decades [39]. *Combinatorial testing* uses value combinations for program inputs to detect the failures triggered by certain input parameter interactions. When a program under test has multiple input parameters and each parameter has multiple possible values, it can be cost-effective for detecting most faults by trying only limited selected value combinations for input parameters. In combinatorial testing, *k-way testing* denotes that every possible value combination of any *k* program input parameters has been tested at least once [41]. It was reported that 5- or 6-way testing has been very effective as it can detect most faults [30].

However, combinatorial testing only focuses on input parameters of a program but pays little attention to the program's internal structure (i.e., black box). Therefore, it can fail to test certain paths of the program, whose conditions can hardly be satisfied. As such, we explore in this paper the feasibility of conducting combinatorial testing via white-box branch information (e.g., which branches of the program under test are taken in the execution of a test input). To do so, we propose a novel *branch-based combinatorial testing technique* (or *BCT*), and analyze its effectiveness. For ease of presentation, we name the traditional *input-based combinatorial testing ICT*.

In software testing, if a certain test input causes a program under test to fail against its test oracle, we say that this program contains a bug. A program can contain multiple input parameters. ICT would try different combinations of values for these input parameters to exercise the program to check whether it contains any bug. A notion of *k*-way testing, from ICT, refers to the practice of trying every possible combination of values for any *k* input parameters at least once in testing a program. BCT works similarly to ICT except that it replaces the use of input parameters by that of branches taken in test executions. As such, we need to map branch-taking conditions (i.e., which branches are taken in testing) in BCT to input-value conditions (i.e., which values are taken in testing) in ICT. However, how this mapping can be done is unclear. Besides, it is also unclear how branch-taking conditions can be enumerated since branches taken in test executions, which are beyond control in BCT, are different from values of input parameters, which can be easily controlled in ICT.

ICT assumes that all possible values of input parameters for a program under test are known in advance, and thus ICT can enumerate their combinations and easily control input-value conditions. For example, a program has two input parameters, which can both take a value from {1, 2, 3}. Then, ICT can enumerate totally nine input-value conditions (3×3). The counterpart, branch-taking conditions, in BCT means combinations of branches taken in a program's execution. For example, a program has two branch statements, one of which is *if-then-else* and the other is *if-then*. For the former, there are four cases in program executions, namely, only *then* sub-branch executed, only *else* sub-branch executed, both executed, and neither executed. Similarly, the latter has two cases. Then, BCT needs to enumerate totally eight branch-taking conditions (4×2). To do so, we propose to monitor and measure branch-taking conditions during testing for BCT instead of controlling them directly before testing. Besides, we also propose a greedy strategy to minimize the number of tests required to achieve the *k*-way testing goal in BCT. Similarly, *k-way testing* in BCT refers to the effort of trying every possible combination in branch-taking conditions associated with any *k* branch statements at least once.

We evaluated BCT on a set of Java programs. Our experiments show that achieving 3- or 4-way testing in BCT

*Corresponding author

can already bring satisfactory effectiveness, e.g., over 80% fault detection rate. We observed that the greedy strategy can reduce over 50% tests under 3- or 4-way setting, thus saving huge test execution overhead. We also observed that merging nested branches (i.e., treating a branch statement and its nested ones as a whole) in a program can detect more faults under the same *k*-way setting than considering them separately (i.e., treating them as different branch statements). For example, under the same 2-way setting, the former can detect 3% more faults than the latter on average. Moreover, when BCT achieves comparable fault detection rates to ICT, it requires much fewer tests (up to 90% reduction). This trend is consistent and becomes more obvious with the growth of *k* in *k*-way testing. Besides, BCT can also achieve higher fault detection rates than existing techniques guided by statement coverage (15−60%) or branch coverage (5−15%) under the 3- or 4-way setting.

In summary, we make the following contributions in this paper:

- We propose the idea of conducting combinatorial testing based on white-box branch information and realize it as a novel technique BCT.
- We take a greedy strategy in BCT to minimize the number of tests required for achieving certain *k*-way testing.
- We evaluate our BCT experimentally with Java programs.

The rest of this paper is organized as follows. Section II overviews ICT and discusses the differences between ICT and BCT using an illustrative example. Section III presents necessary terminologies and elaborates on our BCT framework and its realization. Section IV evaluates BCT's effectiveness in software testing. Section V reviews related work in recent years, and finally Section VI concludes this paper.

II. Overview

In this section, we briefly introduce the traditional input-based combinatorial testing and compare it with our proposed branch-based combinatorial testing.

A. Input-based Combinatorial Testing

Input-based combinatorial testing was first introduced into software testing in 1985 by Mandl [39] to test Ada compilers. ICT tries different combinations of values for input parameters to exercise a program under test to detect faults.

In ICT, *k*-way testing refers to the practice of trying every possible combination of values for any *k* input parameters at least once in testing a program. Among them, 2-way testing, also called *pairwise testing*, has been widely used due to its high cost-effectiveness [3].

There are various popular strategies to help generate test inputs for ICT, such as greedy [8], [32], [42], [45], [53] and heuristic strategies [15], [24]. There are also other studies focusing on how to prioritize tests to achieve a *k*-way testing goal in ICT [3].

B. Differences between ICT and BCT

In this sub-section, we discuss the differences between BCT and ICT using an illustrative example.

Take the simple function `foo` in Fig. 1 as a motivating example, which triggers a `java.lang.ArithmeticException` exception when `flag` at Line 12 is equal to zero. There are three *if-then* statements in total in `foo`, and each of them contains two possible cases in program executions, namely, *then* sub-branch ever executed, and *then* sub-branch never executed. We use `br1`, `br2` and `br3` as in Fig. 1 to name these branch statements for ease of presentation. In order to enumerate the two cases for each branch, we use 1 to represent *then* sub-branch ever executed, and 0 to represent *then* sub-branch never executed. All possible values of input parameters for test inputs and corresponding values br_1, br_2 and br_3 of branch-taking conditions for branches `br1`, `br2` and `br3` in executions are listed in Table I.

To explain the differences between ICT and BCT, we conducted 2-way testing on `foo` by applying both ICT and BCT in turn. In ICT, 2-way testing means to try every combination of values for any two input parameters in `foo`, or in other words, trying every combination in input-value conditions associated with any two input parameters. For example, suppose that for any two input parameters among all of the three, e.g., `type` and `x`, there are totally six input-value conditions (3×2) for testing, and each represents one combination of values for these two parameters.

On the other hand, 2-way testing in BCT means to try every combination in branch-taking conditions associated with any two branch statements. For example, for any two branch statements in `foo`, e.g., `br1` and `br2`, there are four branch-taking conditions (2×2) associated with them, and each represents a combination for testing. If a test input includes certain combinations of values for input parameters or its execution can test certain combinations in branch-taking conditions, we say that these combinations are *covered* by this test input.

```
1  enum Type { L, M, R }
2  int foo(Type type, boolean x, boolean y){
3      int flag = 1;
4      int result = 0;
5      if (type == Type.M) { //br1
6          result = -- flag;
7      }
8      if (x != y && type != Type.R) { //br2
9          result = ( ++ flag ) * 2;
10     }
11     if (y == z) { //br3
12         result = 1 / flag;
13     }
14     return result;
15 }
```

Fig. 1: Motivating example: a function `foo`

TABLE I: 12 tests for `foo`

Test	type	x	y	br_1	br_2	br_3
test1	Type.L	false	false	0	0	0
test2	Type.L	false	true	0	1	1
test3	Type.L	true	false	0	1	0
test4	Type.L	true	true	0	0	1
test5	Type.M	false	false	1	0	0
test6	Type.M	false	true	1	1	1
test7	Type.M	true	false	1	1	0
test8	Type.M	true	true	1	0	1
test9	Type.R	false	false	0	0	0
test10	Type.R	false	true	0	0	1
test11	Type.R	true	false	0	0	0
test12	Type.R	true	true	0	0	1

To reduce the number of tests for achieving a *k*-way testing goal in testing `foo`, we can use a greedy strategy to select a subset of tests from all the tests given in Table I. Each time we select a test that covers the most *uncovered combinations* (i.e., combinations that have not been covered by selected tests so far). The selection continues until the selected subset of tests can already cover all combinations required for achieving a *k*-way testing goal. It works similarly for both ICT and BCT.

For ICT, one acceptable test suite generated by the greedy strategy to achieve 2-way testing can be T_{ICT} = {test1, test4, test6, test7, test9, test12}. T_{ICT} covers all combinations of values for any two input parameters in `foo`, thus achieving the 2-way testing goal for ICT. Since ICT pays no attention to the internal structure of `foo`, T_{ICT} can only make efforts to try every combination of values for input parameters, and cannot trigger the `java.lang.ArithmeticException` exception since `flag` at Line12 cannot be zero as tested by T_{ICT}.

On the other hand, BCT considers the internal structure of `foo` and focuses on its *branch-taking information* (i.e., which cases of branch statements are taken in certain executions), as illustrated by values of br_1, br_2 and br_3 in Table I. Then, using our greedy strategy to select tests to achieve the 2-way testing goal for BCT, we can generate another test suite T_{BCT} = {test1, test2, test7, test8}. T_{BCT} can trigger the `java.lang.ArithmeticException` exception at Line 12, because T_{BCT} can exercise Line 6 and Line 12 in turn and this leads `flag` to be zero at Line 12 in the execution of test8.

III. Branch-based Combinatorial Testing

In this section, we present some terminologies first. Then, we propose our BCT framework and discuss its realization in practice.

A. Terminologies

Suppose that a program under test contains n branch statements. We define several concepts below:

***Definition 1* (*Branch Range*)**: A branch statement contains one or more sub-branches or clauses, which can have many possible cases in its execution. We denote these different cases by different integers. The branch range of a branch statement is a set of such integers. We use B_i (*i* = 1, 2,..., *n*) to denote the branch range of the *i*-th branch statement in the execution of a program and denote these different cases by successive integers starting from zero.

For example, four cases of *if-then-else* branch statements in executions, namely, neither executed, only *then* sub-branch executed, only *else* sub-branch executed, and both executed, are denoted by 0, 1, 2 and 3 in order. Our earlier example function `foo` happens to contain no loop and have three branches with *then* sub-branch only, `br1`, `br2` and `br3`. This makes B_i (*i* = 1, 2, 3) can only take one value from {0, 1}. Then, for any *i*-th branch statement in `foo`, it has two different possible cases in the program's executions, and thus its branch range is B_i = {0, 1} (*i* = 1, 2, 3).

***Definition 2* (*Branch-taking condition*)**: A branch-taking condition associated with *k* certain branch statements represents a possible combination of values for branch ranges of these *k* branch statements.

For example, the branch range for `br1` in the motivating example `foo` is {1, 0}, and so is `br2`. Therefore, there are four branch-taking conditions associated with these two branches (2×2), i.e., {br_1 = 1 && br_2 = 1}, {br_1 = 1 && br_2 = 0}, {br_1 = 0 && br_2 = 1} and {br_1 = 0 && br_2 = 0}. Each of them represents a certain combination of values for associated branch ranges.

***Definition 3* (*k-way testing* in BCT)**: If every combination in branch-taking conditions associated with any *k* branch statements in a program has been tested at least once, we say that *k*-way testing in BCT has been achieved, and name the corresponding test suite a *k*-way test suite in BCT.

For example, testing with the test suite T_{BCT} can cover every combination in branch-taking conditions associated with any two branches in `foo`, so T_{BCT} is a 2-way test suite and testing with T_{BCT} achieves the 2-way testing goal.

Besides, we also define input-value condition and *k*-way testing in ICT for ease of presentation.

***Definition 4* (*Input-value condition*)**: An input-value condition associated with any *k* input parameters represents a possible combination of values for any *k* input parameters in a program.

For example, there are six input-value conditions (3×2) associated with input parameters `Type` and `x` in `foo`, i.e., {`type = Type.L && x = true`}, {`type = Type.L && x = false`}, {`type = Type.M && x = true`}, {`type = Type.M && x = false`}, {`type = Type.R && x = true`} and {`type = Type.R && x = false`}.

***Definition 5* (*k-way testing* in ICT)**: If every combination in input-value conditions associated with any *k* input parameters in a program has been tested at least once, we say that *k*-way testing in ICT has been achieved, and name the corresponding test suite a *k*-way test suite in ICT.

For example, testing with the test suite T_{ICT} can cover every combination in input-value conditions associated with any two input parameters in `foo`, so T_{BCT} is a 2-way test suite and testing with T_{BCT} achieves the 2-way testing goal.

B. BCT Framework

Our BCT conducts combinatorial testing based on white-box branch information. Different from ICT, which uses value combinations for program inputs to detect faults triggered by certain input parameter interactions, BCT tries to detect faults that are tough to be detected because of their complex triggering conditions that are difficult to satisfy. So, BCT uses combinations in branch-taking conditions instead for help.

As we mentioned earlier in Section I, there are two challenges in conducting BCT. The first is how to map BCT to ICT at a conceptual level. To address the first challenge, we define branch-taking conditions in BCT and input-value conditions in ICT to bridge them by mapping. The second challenge is how to enumerate branch-taking conditions since the actual branches taken in executions are beyond control. An execution path is determined by a program under test itself and its corresponding test input. It can hardly be controlled before actual execution. To address the second challenge, we propose to monitor and measure branch-taking conditions during testing. We will explain the details of addressing these challenges in the following.

We present our BCT framework in Fig. 2. This framework consists of three steps: extracting branch-taking information from test executions, pruning redundant branch-taking information, and selecting tests greedily. For ease of controlling branch-taking conditions, we select tests (i.e., *a subset of tests*) from all tests we generated in advance (i.e., *a universal set of tests*) instead of generating tests during testing directly. With the goal of *k*-way testing, BCT applies these steps and eventually check whether faults can be detected by selected tests. Details of each step is as follows:

1) Extracting branch-taking information: This step extracts branch-taking information from test executions.

BCT assumes to obtain tests for programs in advance. Under this assumption, we extract branch-taking information from all executions of these obtained tests. For branch statements such as *if*, *switch* and *try-catch*, we extract information about which cases have been taken in executions of these tests. For loop statements such as *while*, *do-while* and *for*, we extract information about whether statements inside a loop have been executed.

During extracting branch-taking information from test executions, we obtain all executed cases for each branch statement and consider them as all optional cases for each branch. Then we can obtain the branch range for each branch by mapping each case to a unique integer, as illustrated in our earlier `foo` example in Section II. After that, it is straightforward to enumerate branch-taking conditions according to Definition 2.

For example, in order to achieve the 2-way testing goal for `foo` in Fig. 1, we generate branch-taking conditions associated with any two branches, i.e., three (C_3^2) different choices for two branches. However, directly combining the values for branch ranges of different branches (i.e., making up theoretical branch-taking conditions according to Definition 2) may bring *infeasible combinations* (i.e., combinations that cannot be covered by any test input). Consider `br1` and `br2` in the `foo` example. Theoretical branch-taking conditions associated with them contain four combinations (2×2), since branch ranges of `br1` and `br2` are both {0, 1}. Those combinations can be all covered by T_{BCT}. However, if we change the condition `type == Type.M` at Line 5 to `type == Type.R`, corresponding combinations would include infeasible ones, e.g., {br_1 = 1 && br_2 = 1}. This is because for the modified `foo`, the condition `type == Type.R` at Line5 is opposite to the internal condition `type != Type.R` at Line 8. Therefore Line 6 and Line 9 cannot be exercised at the same time by any test. This makes {br_1 = 1 && br_2 = 1} an infeasible combination. However, it is impossible to decide whether certain combinations in theoretical branch-taking conditions contain any one that is infeasible in advance. So, instead of using theoretical branch-taking conditions, we choose to focus on practical branch-taking conditions (i.e., only considering existing combinations in the executions of generated tests). This step maps BCT to ICT at a conceptual level, thus addressing the first challenge in BCT.

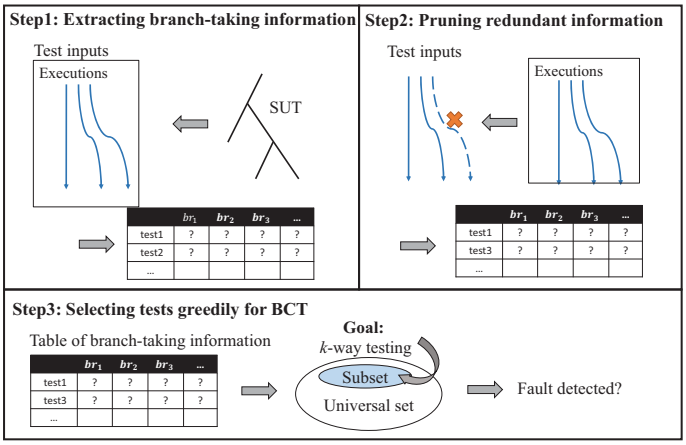

Fig. 2: The BCT framework

2) Pruning redundant information: This step prunes redundant branch-taking information extracted in the first step.

Branch-taking information is the only criterion to distinguish test inputs in BCT. When two test inputs share the same branch-taking information in their executions, we treat them as the same with respect to achieving a certain *k*-way testing goal. So we prune such redundant branch-taking information in this step.

For example, when analyzing the function `foo` in Fig. 1, we prune redundant test inputs such as test9, test10, test11 and test12, since they are the same as at least another test in branch-taking information. For example, we consider test1 and test9 as the same, because both of their branch-taking information are {br_1 = 0 && br_2 = 0 && br_2 = 0}, thus pruning test9. This is for obtaining a universal set of tests, which has no repeated branch-taking information.

3) Greedily selecting tests: This step selects tests greedily to achieve a required *k*-way testing for BCT.

In this step, we use monitoring and measuring to address the second challenge. Since branch-taking conditions are beyond control, it is difficult to determine them before actual executions. So, we do not manipulate them directly. Instead, we use branch-taking information extracted from Step1, and select tests from the universal set of tests with the goal of achieving *k*-way testing in BCT. Then, we monitor the whole selecting process and measure the corresponding coverage for the selected tests. In this way, we skip controlling branch-taking conditions directly, but instead measure coverage information to achieve different *k*-way testing goals.

We also use a greedy strategy to minimize the number of tests required to achieve a certain *k*-way testing goal in BCT by Algorithm 1. Each time we select a test that contains the most uncovered combinations in corresponding branch-taking conditions from a universal set of tests (Lines 6-10). This procedure is repeated until all combinations are covered. During the selection, we ignore those combinations that have already been covered and ensure that each test we select brings at least one new combination. In this way, we control and measure branch-taking conditions with the goal of achieving a required *k*-way testing.

We present our greedy strategy used in Algorithm 1 and elaborate on function *calculatedUncoverCom* in Algorithm 2. In order to select a test that covers the most uncovered combinations for achieving the *k*-way testing goal, we list all combinations in branch-taking conditions associated with any *k* branches among all the *n* branches (C_n^k different choices) in a program and calculate the number of uncovered combinations for each remaining test. The variable score at Line 3 indicates a test's ability to cover remaining uncovered combinations (the larger, the better).

We adopt some optimizations so that we do not have to calculate C_n^k times in each selection. For example, when all tests behave the same for some branch statements in their executions, we would ignore such branch statements since they do not contribute to our greedy strategy. Suppose that there are x such branches. In this way, we only need to analyze C_{n-x}^k different choices of *k* branch statements in enumerating all branch-taking conditions. The complexity is reduced exponentially with respect to the decrease of branch statements considered. Moreover, there are some other heuristic strategies such as [9], which may bring some more optimizations. As an exploratory study, we only realize the basic, yet widely used, the greedy strategy. We may try to implement other heuristic ones in the future.

Algorithm 1: Greedy strategy

1 *Let T be the set of test inputs prepared for P*;
2 *Let U be the set of all the combinations in branch-taking conditions of T*;
3 *Let $branchTable$ be the table of branch-taking information of all tests*;
4 *Let k and N be required k in k-way testing goal and branch sum of P;*
5 **while** $U \neq \emptyset$ **do**
6 $comForAllTests \leftarrow calculateUncoverCom(branchTable, T, k, N, U)$;
7 $t \leftarrow returnMaxTest(comForAllTests)$;
8 $C \leftarrow returnCombinations(t, branchTable)$;
9 Remove t from T;
10 Remove C from U;
11 **end while**;

Algorithm 2: Calculating uncovered combinations

1 **Input:** $branchTable$, T, k, N, U
2 **Output:** $comForAllTests$
3 **int** $score$;
4 **for** $t \leftarrow selectOneRemainingTest(T)$ **do**
5 $score \leftarrow 0$;
6 **for** $locs \leftarrow branchConds(k, N)$ **do**
7 $locValue \leftarrow getValue(locs, branchTable, t)$;
8 **if** $hasnotChosenBefore(locValue, U)$
9 **then**
10 $score \leftarrow score + 1$;
11 **else**
12 **continue**;
13 **end if**
14 **end for**
15 $add(comForAllTests, score)$;
16 **end for**
17 **return** $comForAllTests$;

C. Realization

We plan to exploratively study the effectiveness and efficiency of our proposed BCT technique, and we realize the following parts for its practical runs:

1) Generating the universal set of test inputs: Given a program under test, we need to prepare tests due to the BCT's assumption. For convenience, we use *Random Testing* (*RT*) to generate tests. Other test generation strategies also work here.

2) Generating faulty programs: Mutation Testing is a fault-based testing technique, which has been studied for more than three decades [24], [27]. We adopt a publicly available tool named MuJava [38], which supports the whole mutation testing process, including generating faulty versions (i.e., *mutants*), executing tests against mutants, and calculating mutant scores. In our realization, we use its built-in mutation operators to generate the set of mutants for the program under test and use mutation scores to measure the fault detection ability for each selected test.

We apply First Order Mutants (FOMs) [24] instead of Higher Order Mutants (HOMs), since HOMs can usually be constructed from a sequence of FOMs.

3) Instrumentation: Instrumentation is one popular technique to monitor executions of programs, and can be used for us to collect actual branch-taking information at runtime. We instrument faulty versions generated by MuJava and execute

generated test inputs against the instrumented versions. Then, during test executions, we can extract branch-taking information from test executions. We define BCT at the source level, so we conduct static source code instrumentation. Byte code instrumentation will also work with the help of the mapping process between source code and byte code versions of programs.

4) Evaluating BCT: We compose everything together and realize our BCT framework to achieve a required k-way testing goal. If the goal is achieved, we check whether the seeded fault of each mutant can be detected by the selected tests for achieving this goal. We use mutation score to measure a test suite's ability to detect faults when achieving k-way testing, since mutation score has been widely adopted as a proxy of detection ability [27].

IV. Evaluation

In this section, we evaluate BCT's effectiveness experimentally, and present our observations.

A. Research Questions

In order to comprehensively evaluate BCT, we raise four research questions in terms of effectiveness and efficiency.

RQ1: *Is BCT effective and how does its ability to detect faults change with different k-way settings?*

By answering this research question, we try to find some properties of BCT and evaluate its effectiveness. In ICT and BCT, k-way testing refers to the practice of trying every possible combination in input-value conditions associated with any k input parameters (ICT) or branch-taking conditions associated with any k branches (BCT) at least once in testing a program. ICT can detect most faults by 5- or 6-way testing [30]. We investigate whether BCT has a similar property (e.g., how many faults can BCT detect by different k-way settings).

RQ2: *Using a greedy strategy, how much overhead can it save?*

In the framework as we introduced in Section III, we apply a greedy strategy in our test selection, which is used to minimize the number of tests. The greedy strategy helps reduce the number of tests, thus reducing the execution overhead in testing. Tests often need to be executed for many times, especially in regression testing, so it is practically useful to reduce the number of required tests.

RQ3: *Is BCT's effectiveness related to nested branches in a program, and how much impact do nested branches have?*

Nested branches play an important role in programs, and may lead to complex restrictions in executing the concerned branch statements (e.g., a branch must be taken before another executes). So we try to explore the possible behavior and compare the impact of merging nested branches as a whole to that of not doing so.

RQ4: *Compared with ICT and other existing techniques guided by statement or branch coverage, how much does our BCT improve?*

We also compare BCT with existing testing techniques. On one hand, we compare the effectiveness and the number of selected tests between BCT and ICT, respectively. On the other hand, we select existing techniques guided by statement or branch coverage to compare with our approach, since they are widely used in practical software testing [57], [17], [43].

B. Experimental Subjects

We selected nine Java programs as our experimental subjects, and all of them can be obtained from open-source websites like GitHub [16], SIR [50], LeetCode [31]. The statistics of these subject programs are listed in Table II. The second column lists a concerned program's source or or its brief description. The column "BranchNum" lists the number of branches at the source and byte code (in brackets) level for each program. We divide all subject programs into three groups, namely, Small, Medium and Large, according to their branch numbers, as shown in column "Group". We prepared for the experiments according to our earlier explained realization part in Section III.

First, we randomly generated test inputs for each subject program. We generated a total of 1,000 tests for group "Small", and 1,200 for group "Medium". For Jtcas, we used the tests that come along with this program (1,487), since they were also generated by a similar random mechanism. For the programs in group "Large", we generated 1,500 tests for ShortestPath and 1,200 tests for ClosestPair.

Then, for each subject program, we used MuJava to generate mutants, and executed tests on these mutants. After filtering out invalid mutants (crashed or failed in compilation), we obtained a total of 3,511 mutants (the fourth column in Table II). We discarded those mutants that cannot be *killed* by any test (i.e., a test can kill a mutant when the output of this mutant after executing the test is different from the output of its original program) [24], and conducted experiments on the remaining 2,489 mutants (the fifth column in Table II).

C. Experimental Setup

We evaluated our technique BCT on the Java programs listed in Table II. We conducted experiments to answer research questions with respect to the divided different program groups.

For RQ1, we measure BCT's effectiveness by mutation score (as our dependent variable) since it is a proxy of detection ability as mentioned earlier [27]. We control the value of k in k-way testing as our independent variable.

For RQ2, we set the number of tests as the dependent variable, and different selecting strategies (random or greedy) as the independent variable.

For RQ3, to find whether merging nested branches affects BCT's effectiveness, we still measure BCT's mutation score (as our dependent variable). Besides, we control the treatment on nested branches (i.e., considering them as a whole or separately) as the independent variable.

Finally, for RQ4, to compare BCT with other testing techniques, we control the selected technique for comparison

TABLE II: Statistics of subject programs

Subject name	Source/description	LOC	# Mutants	# Killed	% Killed	TestNum	Group	BranchNum	MergeBranch
PrimeNumber	Judge the prime number	17	47	34	72%	1,000	Small	3 (5)	1
LeapYear	Judge the leap year	21	61	49	80%	1,000	Small	3 (3)	2
Median	Return the median number	22	101	82	81%	1,000	Small	5 (5)	2
DayOfMonth	Display day number	40	227	183	81%	1,000	Small	6 (17)	3
Decode	LeetCode OJ Problem	78	563	350	62%	1,200	Medium	13 (39)	6
BoyerMoore	LeetCode OJ Problem	93	345	259	75%	1,200	Medium	13 (31)	9
Jtcas	SIR	169	545	362	66%	1,487	Medium	18 (48)	13
ShortestPath	LeetCode OJ Problem	271	653	478	73%	1,500	Large	32 (53)	16
ClosestPair	LeetCode OJ Problem	370	969	692	71%	1,200	Large	40 (54)	27
Total	-	1,081	3,511	2,489	71%	10,587	-	140 (255)	79

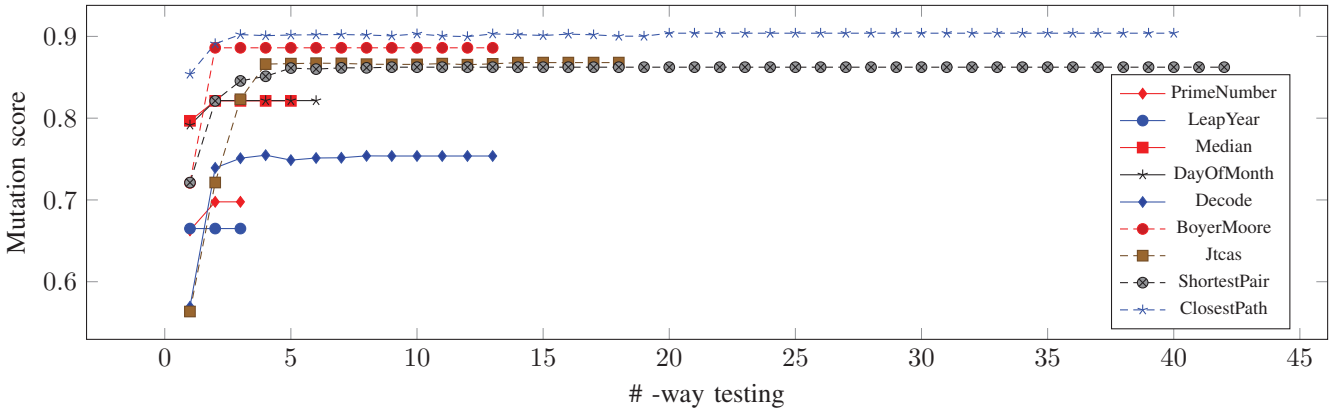

Fig. 3: Different fault detection rates when achieving different k-way settings for BCT

(i.e., BCT, ICT or techniques guided by statement or branch coverage) as the independent variable. We still compare their testing effectiveness by mutation score (as the dependent variable). Besides, to compare the cost-effectiveness of BCT and ICT, we use both mutation score and number of tests as dependent variables.

D. Experimental Results and Analyses

In the following, we list the four research questions and answer them in turn. We conducted our experiments on a Linux Server with 32 cores of Intel Xeon CPU @2.66GHz.

RQ1: *Is BCT effective and how does its ability to detect faults change with different k-way settings?*

To answer this question, we measure the mutation score for BCT under different k-way settings. Fig. 3 shows the results. We tested all programs. We observe that BCT's effectiveness in testing different programs has a similar pattern with the growth of k in k-way testing: increasing sharply first, then becoming stable when k reaches a certain value (around three or four). However, LeapYear behaved differently. Its mutation score for LeapYear score did not even change when we increased k. We studied it and found that LeapYear's program logic is so simple that its mutation score reached its limit even in the case of 1-way testing. In general, BCT exhibits satisfactory effectiveness (over 80% fault detection rate on average) when we set the value of k to three or four. This result is impressive.

We investigated BCT's effectiveness and found that achieving 3- or 4-way testing in BCT can already bring satisfactory effectiveness (over 80% fault detection rate).

RQ2: *Using a greedy strategy, how much overhead can it save?*

We chose programs in groups "Medium" and "Large" to conduct experiments to answer this question. After preparing tests for each program and pruning redundant branch-taking information, we used both a greedy strategy and a random strategy in turn to select tests from the universal set. Fig. 4 shows the results. The x-axis represents k-way testing with different values of k, and y-axis represents the ratio of selected tests against all tests, averaged on all tested programs. In Fig. 4, we observed that the greedy strategy can reduce over half tests for 3- or 4-way testing, which is the level of inducing satisfactory testing effectiveness as answered in RQ1. Therefore, our greedy strategy can greatly improve the efficiency for BCT execution and save its testing overhead.

We evaluated the benefit of our greedy strategy, and found that it can help reduce over half tests, thus saving huge execution overhead in testing.

RQ3: *Is BCT's effectiveness related to nested branches in a program, and how much impact do nested branches have?*

Nested branches play an important role in programs, and may lead to complex restrictions in program executions. In our experiments, we had different treatments on nested branches

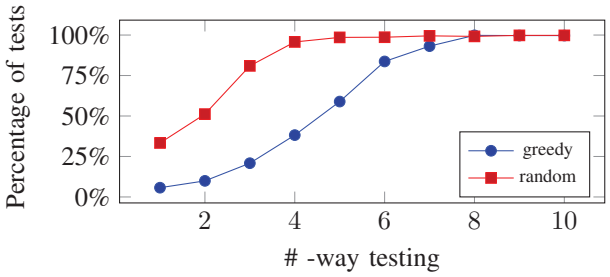

Fig. 4: Comparison between the greedy and random strategies in selecting tests when achieving different k-way settings for BCT

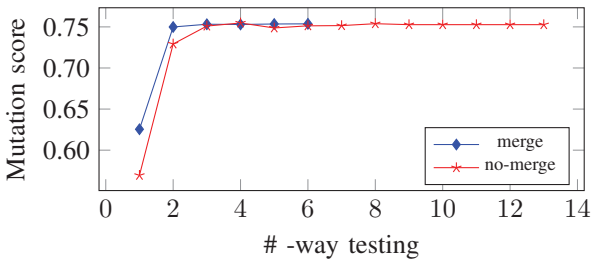

Fig. 5: Comparison between *merge* and *no-merge* when achieving different k-way settings for Decode

(merge or no-merge). *Merge* refers to the treatment of considering a branch statement and its nested branch ones as a whole, while *no-merge* refers to the treatment of considering each branch statement separately, no matter whether it contains any nested branch statement or not. This would change the number of branches under consideration in a program, as well as changing their corresponding branch ranges.

We selected all three programs from group "Medium" and one program from group "Large" to investigate this question. Programs in group "Small" are ignored as they typically contain too few branch statements. We list the number of original branches (BranchNum) and number of branches after merging nested ones (MergeBranch) for each program in Table II.

Fig. 5 to Fig. 8 show the comparison results for different programs, respectively. They consistently show that the mutation score of BCT with merge is relatively higher than that of BCT with no-merge within valid k value ranges. Therefore, applying the merge treatment can help detect more faults than no-merge under the same k-way setting, e.g., around 3% more on average under the 2-way setting.

> *We observed that merging nested branches helps detect more faults in the same k-way testing, (e.g., 3% more under the 2-way setting), than considering them separately in BCT.*

RQ4: *Compared with ICT and other existing techniques guided by statement or branch coverage, how much does our BCT improve?*

We consider two aspects to answer this question. First, we conducted experiments to compare BCT with ICT. We chose Jtcas as our subject program because it has 12 input parameters, which are comparable to its contained branch statements. Note that it is impossible to try every possible value for an input parameter when it has infinite or too many possible values. So, we chose to partition the infinite or a too large value range for an input parameter into equivalent groups and use a single sample in each group as its representative. For Jtcas, whose input parameters are all integers, we partitioned possible values for each parameter by a modulus operator. For ICT1, we conducted the modulus operator with 5 against all possible values and partitioned these values into different

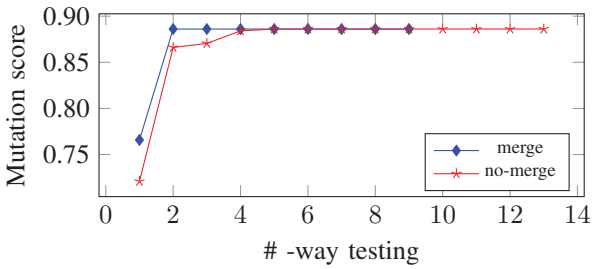

Fig. 6: Comparison between *merge* and *no-merge* when achieving different k-way settings for BoyerMoore

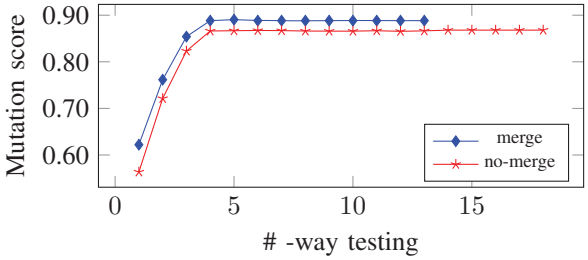

Fig. 7: Comparison between *merge* and *no-merge* when achieving different k-way settings for Jtcas

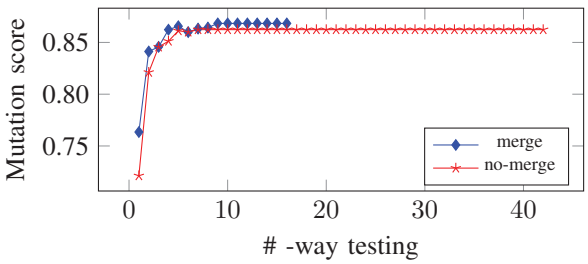

Fig. 8: Comparison between *merge* and *no-merge* when achieving different k-way settings for ShortestPath

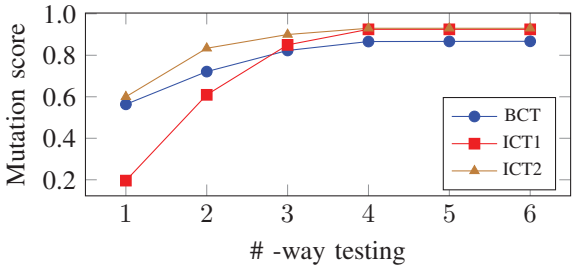

Fig. 9: Comparison on fault detection rate when achieving different k-way testing for BCT and ICT

groups according to their remainders from this operation. ICT2 worked similarly but with 10 for the modulus operation. To conduct a fair comparison with BCT, we used the same set of universal tests and only partition those tests available in this set. We conducted experiments until 6-way testing as results became stable then. Fig. 9 shows the result. We observe that ICT1 and ICT2 behaved slightly better than BCT when k is larger enough ($\geq$3). When k is smaller than three, BCT behaved between ICT1 and ICT2. However, we note that this seemingly slight advantage of ICT over BCT came at the cost of much more tests required, as shown in Fig. 10. To achieve satisfactory testing effectiveness (e.g., with a mutation score greater than 0.8), ICT1 and ICT2 both require much more tests than BCT. This trend is consistent and becomes more obvious with the growth of k in k-way testing. For example, BCT requires only 56 tests while ICT1 requires over 600 tests and ICT2 requires over 800 under the same 6-way setting. This indicates that BCT can save much more execution overhead compared to ICT, when achieving comparable effectiveness.

Second, we also compared our BCT with existing techniques that are guided by statement or branch coverage in their fault detection rates. We selected all three programs in group "Medium" (Decode, BoyerMoore and Jtcas) and one program in group "Large" (ClosePair) as our subject programs. Fig. 11 shows the results. We used five settings for statement coverage (60%, 70%, 80%, 90% and 95%) and three settings for k-way testing for BCT (k = 2, 3, 4). For Decode, we could only realize a statement coverage up to 92% due to this program's own limit. We observe that BCT clearly achieves the highest fault detection rate for all the four programs. Besides, its 3- or 4-way testing behaved up to 60% better than techniques guided by statement coverage and up to 15% better than techniques guided by branch coverage. This indicates that our BCT represents a very strong test adequacy criterion for testing.

We observed that when BCT reaches a comparable detection rate to ICT (over 80%), it requires much less tests (up to around 90% reduction). We also observed a great improvement in BCT's test effectiveness than existing techniques guided by statement (15−60%) or branch coverage(5−15%).

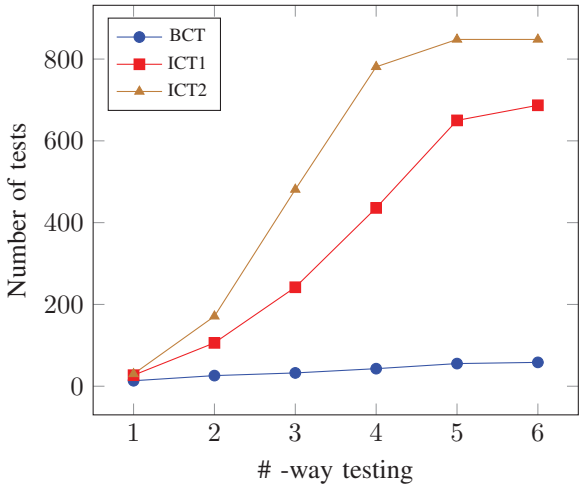

Fig. 10: Comparison on number of required tests when achieving different k-way testing for BCT and ICT

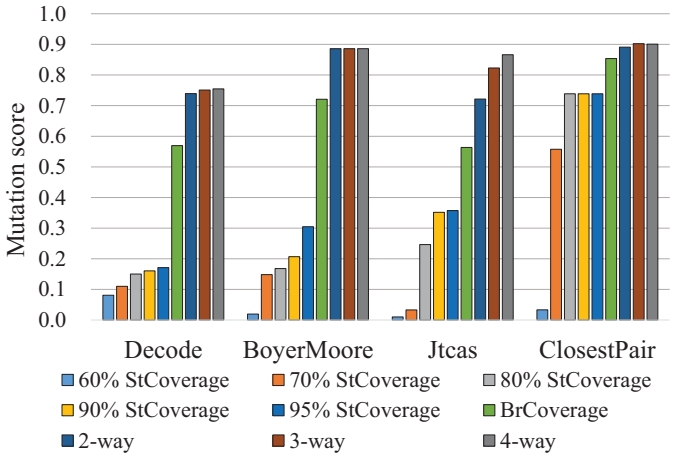

Fig. 11: Comparison on fault detection rate when achieving different k-way testing for BCT and techniques guided by statement or branch coverage

E. Threats to Validity

In our experiments, we empirically explored and evaluated our BCT's effectiveness in software testing. One major threat could be the selected subject programs that may seem not sufficiently large. We note that none of our experiments utilized any special feature relating to such "may-not-be-large" programs. Our experiments tried to alleviate this threat by controlling different variables and isolating irrelevant factors in different groups of experiments. Besides, although these programs themselves are not very large, our experiments are complex enough, covering extremely lots of combinations from programs, mutants, branches and ways. Our experiments were conducted on a powerful server as mentioned earlier. Nevertheless, even if we already ran experiments using eight threads, it still took us eight continuous weeks to complete due to its prohibitively large scale. Still, we acknowledge that

our experiments deserve further extensions on larger subject programs to better validate the results reported in this paper.

F. Discussion

On a technical perspective, we realize BCT as a novel technique and evaluate it experimentally with real-world Java programs. It is shown that BCT can be both feasible and cost-effective. As an exploratory study, our primary goal is to explore the feasibility of conducting combinatorial testing based on white-box branch information, rather than propose BCT as a mature testing technique, which may require a more complete and detailed evaluation.

On a practical perspective, we believe that BCT can be easily applied under suitable testing scenarios, e.g., regression testing minimization, selection and prioritization [19], [59]. On one hand, our BCT framework of monitoring and measuring can be directly used in minimizing, selecting or prioritizing reused tests [59] in regression testing, which are retained and reused between different versions. *Reused tests* are used to exercise the parts of a program that remain unchanged across different versions. So, it is natural to obtain their coverage information in executions from a former version, which is also reusable in its later version [62]. Since regression testing is performed to make sure that newly introduced functions of the program do not interfere with the existing ones, it actually makes it worthy and necessary to exercise unchanged parts repeatedly with reused tests and it also makes our work useful on this respect. On the other hand, we also expect that BCT can be applied in various ways with the help of other techniques. For example, since symbolic execution proposes an analysis technique to generate test inputs for certain paths, we believe that combining BCT with symbolic execution can help automate generating test inputs for specific paths that are required by BCT.

As an exploratory study, we present a main framework of BCT and conduct preliminary experiments to evaluate its performance. More practical applications, like combining BCT with symbolic execution, remain for further study because they require further extensions to BCT. We keep it as our future work.

V. Related Work

Our work in this paper relates to various existing studies on ICT, control flow testing criteria, and coverage-guided testing. In this section, we discuss the most relevant work to our work in these fields.

A. Input-based Combinatorial Testing

ICT has been studied for over three decades [39] and has been widely used for addressing practical problems (e.g., testing embedded systems). By trying every possible combination of values for input parameters at least once in testing a program, ICT shows its cost-effectiveness and has been well-accepted in software testing. Recently, there is also research [20] on comparing ICT with other black-box techniques, and ICT performs quite satisfactorily among them.

Some pieces of work focus on generating test inputs for ICT [2], [8], [15], [22], [53], and various tools have been developed such as AETG [8], PICT [10] and CATS [48]. All these existing tools have their own strengths and weaknesses, and they can be selected according to different testing requirements in practice. On a technical perspective, to date, there are generally four main groups of techniques or algorithms proposed: greedy algorithms, heuristic algorithms, mathematic techniques, and random techniques [41]. Greedy algorithms have been most widely used to generate test inputs using ICT in practice due to their simpleness and accuracy, e.g., In Parameter Order (IPO) [33], while heuristic algorithms are applied in order to accelerate the test generation at a small cost of accuracy, e.g., hill climbing, great flood, tabu search, simulated annealing [9] and genetic techniques [15]. In addition, mathematic techniques are often used in the mathematic community, and random techniques are often used as a benchmark technique to analyze effectiveness of other techniques. Some other researchers combine different techniques mentioned. For example, mathematic techniques are combined with simulated annealing in [9] for test generation.

However, when applying ICT to test generation, researchers may suffer from combinatorial explosion when a program contains numerous input parameters and each parameter has numerous optional values. Many pieces of work on ICT focus on alleviating this problem. The latest one [21] proposes a flexible search-based technique using similarity to bypass combinatorial explosion. We expect that such work can enlighten us for further handling the similar combinatorial explosion problem in BCT.

Except for test input generation, which is the most active research area for ICT, there are also some pieces of work on applying ICT to test input prioritization [4], [44], failure diagnosis [49], [58], metrics and evaluation [55]. Furthermore, ICT is also useful in various types of practical applications [12], [28], [29], such as mobile applications [28], satellite communications [23], and regression testing [44].

B. Control-flow Testing Criteria

There are already some well-known testing criteria, such as statement coverage, branch coverage, conditional coverage and path coverage. They can be used as a predication to determine whether a program has already been tested "enough" [14]. Our BCT seems to be seated between branch coverage and path coverage. In fact, 1-way testing for BCT is equivalent to techniques guided by branch coverage, and *full-way* testing (i.e., k is equal to the number of all branches in a program) is equivalent to techniques guided by path coverage.

Besides, there are also some other testing criteria, such as Modified Condition/Decision Coverage (MC/DC) [13], [54] and Reinforced Condition/Decision Coverage (RC/DC) [52]. Indeed, MC/DC seems to be one of the most complicated and controversial control-flow testing criteria and RC/DC is designed to eliminate some of its shortcomings. We consider MC/DC as the closest testing criterion to our BCT. However, MC/DC considers combinations for internal conditions in

each branch statement, while BCT considers combinations in branch-taking conditions and treats each branch statement as a whole, skipping its internal conditions. Thus, they work at different levels and may potentially complement to each other. This deserves further study. Finally, our BCT is customizable by selecting different k values in testing, and this makes it flexible for suiting different needs in software testing (e.g., from the most light branch coverage to the most heavy path coverage).

C. Coverage-guided Testing

Generally, BCT uses coverage of combinations in branch-taking conditions associated to guide the whole testing process. Many testing techniques [11], [25], [35], [51], [56], [60], [62] share the similar idea behind and make efforts to guide testing with the help of extra runtime information, and one of the most popular ways is to use coverage information, named *coverage-guided techniques*.

Coverage-guided techniques have been widely applied to regression testing [25], [46], [61], [62] since coverage knowledge produced by prior executions of tests from former versions has already be obtained. Various types of coverage information have been investigated. For example, branch coverage is used to guide test selection in Adaptive Random Testing (ART) in [7] and it brings great performance improvement [6], [62]. Apart from regression testing, coverage information can also be applied to test generation with the help of test generation techniques like symbolic execution [5], [18], [47], which has been combined with many different testing techniques [26], [34] for specific purposes.

Scenarios mentioned above are also suitable for our BCT. Our proposed technique can not only be directly useful for test prioritization in regression testing, but it can also be useful for automated test generation with the help of techniques like symbolic execution. We also plan to apply BCT to popular Android testing in the future. However, when analyzing Android apps, some additional problems like Android-specific event rules [36], [37] and simulations of user-interaction and sensory data [1], [40] should be also taken into consideration.

VI. Conclusion

In this paper, we propose a novel technique named *branch-based combinatorial testing* (*BCT*), which is based on white-box information. BCT adopts the key insight of combinatorial testing and uses branch-taking conditions to replace input-value conditions in ICT. We evaluated BCT on Java programs and the results show that BCT can achieve satisfactory effectiveness under a 3- or 4-way setting, clearly lighter than ICT. We also evaluated the greedy strategy for selecting tests, and it turns out that this strategy can reduce over half of tests under 3- or 4-way testing. Besides, we compared BCT to ICT and found that BCT requires much less tests (up to 90% reduction) while it still achieves a comparable fault detection rate to ICT. We also compared BCT with existing techniques guided by statement or branch coverage, and the result shows that BCT consistently outperforms these techniques in test effectiveness.

However, our work still has limitations. For example, it may take much time to conduct BCT on those subjects that contain numerous branch statements. We will further investigate and optimize for BCT's performance in such cases. Besides, currently the universal tests are generated randomly, and branch-taking conditions are monitored, measured and controlled at runtime. In future, we plan to combine BCT with existing test generation techniques such as concolic testing or dynamic symbolic execution, so that BCT can be more effective and efficient at generating test inputs.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 83–93.

[2] A. Barrett and D. Dvorak, "A combinatorial test suite generator for gray-box testing," *Space Mission Challenges for Information Technology*, pp. 387–393, 2009.

[3] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[4] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 1–7.

[5] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, pp. 82–90, Feb. 2013.

[6] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Code coverage of adaptive random testing," *Reliability, IEEE Transactions on*, vol. 62, no. 1, pp. 226–237, 2013.

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

[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," *Software Engineering, IEEE Transactions on*, vol. 23, no. 7, pp. 437–444, 1997.

[9] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, 2003, pp. 394–405.

[10] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, 2006, pp. 419–430.

[11] T. Dang and T. Nahhal, "Coverage-guided test generation for continuous and hybrid systems," *Formal Methods in System Design*, vol. 34, no. 2, pp. 183–213, 2009.

[12] I. S. Dunietz, W. Ehrlich, B. Szablak, C. L. Mallows, and A. Iamino, "Applying design of experiments to software testing: experience report," in *Proceedings of the 19th international conference on Software engineering*. ACM, 1997, pp. 205–215.

[13] A. Dupuy and N. Leveson, "An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software," *Digital Avionics Systems Conference, 2000. Proceedings. DASC*, 2000.

[14] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[15] S. Ghazi and M. Ahmed, "Pair-wise test coverage using genetic algorithms," *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 2, pp. 1420–1424, 2003.

[16] GitHub, *http://www.github.com*.

[17] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 302–313.

[18] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *PLDI'05*, 2005.

[19] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[20] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016.

[21] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *Software Engineering, IEEE Transactions on*, vol. 40, no. 7, pp. 650–670, 2014.

[22] R. Huang, X. Xie, T. Y. Chen, and Y. Lu, "Adaptive random test case generation for combinatorial testing," *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pp. 52–61, 2012.

[23] J. Huller, "Reducing time to market with combinatorial design method testing," in *Proceedings of 10th Annual International Council on Systems Engineering (INCOSE'00) 2000*, July 2000.

[24] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

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

[26] H. Jin, Y. Jiang, N. Liu, C. Xu, X. Ma, and J. Lu, "Concolic metamorphic debugging," in *IEEE Computer Software and Applications Conference*, 2015, pp. 232–241.

[27] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing," *ACM SIGSOFT Symposium on The Foundation of Software Engineering*, pp. 654–665, 2014.

[28] R. Krishnan, S. M. Krishna, and P. S. Nandhan, "Combinatorial testing: learnings from our experience," *ACM SIGSOFT Software Engineering Notes*, vol. 32, no. 3, pp. 1–8, 2007.

[29] D. R. Kuhn, R. Kacker, and Y. Lei, "Combinatorial and random testing effectiveness for a grid computer simulator," *NIST Tech. Rpt*, vol. 24, 2008.

[30] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *It Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[31] LeetCode, *http://www.leetcode.com*.

[32] 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*, vol. 18, no. 3, pp. 125–148, 2008.

[33] Y. Lei and K.-C. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*. IEEE, 1998, pp. 254–261.

[34] J. J. Li, D. Weiss, and H. Yee, "Code-coverage guided prioritized test generation," *Information and Software Technology*, vol. 48, no. 12, pp. 1187–1198, 2006.

[35] X. Li, Y. Jiang, Y. Liu, and C. Xu, "User guided automation for testing mobile apps," in *Asia-Pacific Software Engineering Conference*, 2014, pp. 27–34.

[36] Y. Liu, C. Xu, S.-C. Cheung, and J. Lu, "Greendroid: automated diagnosis of energy inefficiency for smartphone applications," *Software Engineering, IEEE Transactions on*, vol. 40, no. 9, pp. 911–940, 2014.

[37] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016)*, in press.

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

[39] R. Mandl, "Orthogonal latin squares: An application of experiment design to compiler testing." *Communications of the Acm*, vol. 28, no. 10, pp. 1054–1058, 1985.

[40] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 461–471.

[41] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 33–63, 2011.

[42] C. Nie, B. Xu, Z. Wang, and S. Liang, "Generating optimal test set for neighbor factors combinatorial testing," in *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, 2006, pp. 259–265.

[43] A. Perez, A. Rui, and A. Riboira, "A dynamic code coverage approach to maximize fault localization efficiency," *Journal of Systems and Software*, vol. 90, no. 2, pp. 18–28, 2014.

[44] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 255–264.

[45] R. Reussner, J. Mayer, and J. A. Stafford, *Quality of Software Architectures and Software Quality*. Springer, Berlin, 2009.

[46] D. S. Rosenblum and E. J. Weyuker, "Using coverage information to predict the cost-effectiveness of regression testing strategies," *Software Engineering, IEEE Transactions on*, vol. 23, no. 3, pp. 146–156, 1997.

[47] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ESEC/FSE'05*. ACM.

[48] G. B. Sherwood, S. S. Martirosyan, and C. J. Colbourn, "Covering arrays of higher strength from permutation vectors," *Journal of Combinatorial Designs*, vol. 14, no. 3, pp. 202–213, 2006.

[49] L. Shi, C. Nie, and B. Xu, "A software debugging method based on pairwise testing," in *Computational Science–ICCS 2005*. Springer, 2005, pp. 1088–1091.

[50] SIR, *http://sir.unl.edu*.

[51] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson, "Bita: Coverage-guided, automatic testing of actor programs," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 114–124.

[52] S. A. Vilkomir and J. P. Bowen, "From mc/dc to rc/dc: formalization and analysis of control-flow testing criteria," *Formal Aspects of Computing*, vol. 18, no. 1, pp. 42–62, 2006.

[53] Z. Wang, C. Nie, and B. Xu, "Generating combinatorial test suite for interaction relationship." *Soqua Fourth International Workshop on Software Quality Assurance in Conjunction with E*, pp. 55–61, 2007.

[54] M. W. Whalen, S. Person, N. Rungta, M. Staats, and D. Grijincu, "A flexible and non-intrusive approach for computing complex structural coverage metrics," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, 2015.

[55] A. W. Williams and R. L. Robert, "A measure for component interaction test coverage," in *Computer Systems and Applications, ACS/IEEE International Conference on. 2001.* IEEE, 2001, pp. 304–311.

[56] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 359–368.

[57] M. H. Yao, Xiangjuan and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 919–930.

[58] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006.

[59] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[60] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," in *Acm Sigplan Notices*, vol. 47, no. 10. ACM, 2012, pp. 485–502.

[61] C. Zhang, Z. Chen, Z. Zhao, S. Yan, J. Zhang, and B. Xu, "An improved regression test selection technique by clustering execution profiles," in *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 2010, pp. 171–179.

[62] Z. Q. Zhou, "Using coverage information to guide test case selection in adaptive random testing," in *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*. IEEE, 2010, pp. 208–213.