FAULT LOCALIZATION BASED ON COMBINATORIAL TESTING

by

LALEH SHIKH GHOLAMHOSSEINGHANDEHARI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2016

Abstract

FAULT LOCALIZATION BASED ON COMBINATORIAL TESTING

Laleh Shikh Gholamhosseinghandehari, PhD

The University of Texas at Arlington, 2016

Supervising Professor: Yu Lei

Combinatorial testing is a software testing strategy that has received a significant amount of attention from academia and industry. After executing a combinatorial test set, the execution status, i.e., pass or fail, of each test is obtained. If there is one or more failed tests, the next task is fault localization, i.e. localizing the fault in the source code. This dissertation addresses the problem of how to perform fault localization by leveraging the result of combinatorial testing.

The major contribution of this dissertation is a fault localization approach called BEN that consists of two major phases: 1) failure-inducing combination identification, 2) faulty statement localization. A combination is failure-inducing if its existence in a test causes the test to fail. The failure-inducing combination identified in the first phase is used to generate a group of tests such that the spectra of these tests can be analyzed quickly to identify the faulty statement in the source code. To the best of our knowledge, BEN is the first approach that performs code-based fault localization by leveraging the result of combinatorial testing. We conducted experiments in which BEN was applied to a set of programs from the Software Infrastructure Repository (SIR). The programs include the programs in the Siemens suite and two real-life programs, i.e., grep and gzip. The experimental results show that our approach can effectively and efficiently localize the faulty statements in these programs.

This dissertation also includes two empirical studies on the effectiveness of combinatorial testing. In the first study, we evaluate the effectiveness of combinatorial testing on the Siemens programs. In the second study, we compare the stability of combinatorial testing to that of random testing. These two studies are conducted as part of our effort to evaluate the effectiveness of BEN, since combinatorial testing must be performed on a subject program before BEN is applied to the program. Both studies contribute to the literature by providing additional data that demonstrate the effectiveness of combinatorial testing.

This dissertation is presented in an article-based format and includes six research papers. The first paper reports our work on the first phase of BEN. The second paper reports our work on the second phases of BEN. The third paper is a journal extension that combines the first two papers and also adds several significant extensions of BEN. The fourth paper is a tool paper that describes the design and usage of a prototype tool that implements BEN. The fifth paper reports the empirical study on input parameter modeling. The sixth paper reports the empirical study, on comparing combinatorial testing and random testing. All these papers have been published in peer-reviewed venues except the third one, which is currently under review.

Acknowledgements

I would like to thank my supervising professor Dr. Jeff Lei for his wisdom, enthusiasm, and encouragement and also for pushing me further than I thought I could go. This thesis would not have been possible without his help and generous support. I wish to thank my committee, Dr. David Kung, Dr. Christoph Csallner, Dr. Donggang Liu and Dr. Junzhou Huang for generously sharing their time and ideas.

I would also like to extend my appreciation to my parents for their constant love and immeasurable sacrifice. I am grateful to my brothers for their support, interest and advice. Finally, I wish to give my heartfelt thanks to my husband whose unconditional love, patience and continual support enabled me to complete this thesis.

March 2016

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

Combinatorial testing is a software testing strategy that has received a significant amount of attention from academia and industry. The key observation is that most software failures are caused by interactions of only a few parameters. A widely cited NIST study reports that failures in several real-life systems involved no more than six parameters. A t-way combinatorial test set is built to cover all the t-way interactions, i.e., interactions involving t parameters, where t is typically a small integer. Empirical results have shown that combinatorial testing is very effective for failure detection while significantly reducing the number of tests.

Most research in combinatorial testing has focused on developing efficient combinatorial test generation algorithms and conducting empirical studies to evaluate the failure-detection effectiveness of combinatorial testing. After a failure is detected, the next task is to find the fault that caused the failure. An important research problem is how to leverage the result of combinatorial testing for fault localization.

1.1 Research overview

In this dissertation, we present an approach to locate one or more faulty statements in the source code using the result of combinatorial test set. Our approach, called BEN, consists of two major phases. In the first phase, BEN identifies failure inducing combination. A combination is failure inducing, or simply inducing, if it causes any test in which it appears to fail. To identify inducing combination, our approach takes as input a t-way test set and their status and reports as output the inducing combination of size t, i.e., the strength of initial test set, or larger. BEN iteratively identifies a set of suspicious combinations in the current test set, initially the combinatorial test set. Suspicious combinations are candidate of inducing combinations. Then, BEN systematically generates a small number of tests which can be executed to refine the suspicious combinations set. The process continues until the failure inducing combination is identified.

In the second phase, BEN localizes the faulty statement in the source code. The second phase takes as input the identified inducing combination, and produces as output a ranking of statements in terms of their likelihood of being faulty. In this phase BEN generates a small group of tests from the inducing combination. The tests are generated in a way such that the spectra of these tests can be analyzed quickly to generate ranking of statements.

To the best of our knowledge BEN is the first approach that deals with code-based fault localization using combinatorial testing. Existing work on fault localization based on combinatorial testing focuses on identifying failure inducing combinations. Also, there are several studies on general code based fault localization problem. BEN differs from the other spectrum-based fault localization approaches, which they do not deal with the test generation problem, and they assume the existence of a large number of tests, which are generated randomly and/or using other techniques. Moreover, the other approaches do not use the benefit of the combinatorial test set, therefore, BEN is more effective and efficient comparing to the general spectrum-based approaches.

BEN was applied to the Siemens suite which contains seven relatively small programs and two large programs, i.e., grep and gzip from the Software Infrastructure Repository (SIR). The results show that BEN is effective in localizing faulty statements and also efficient in that only a small number of tests need to be executed and instrumented. Moreover, we compared the results of BEN and two other spectrum based approaches, Tarantula and Ochiai. Our experimental results show that BEN achieved results that are competitive to or better than Tarantula and Ochiai but with a significantly less number of tests.

We also conducted two empirical studies on the effectiveness of combinatorial testing. These two empirical studies are performed to evaluate the effectiveness of fault localization process using BEN, since BEN is a combinatorial testing based approach, i.e., combinatorial testing must be performed before BEN is applied. In the first study, we applied combinatorial testing on the Siemens programs. The results show that combinatorial testing is very effective and detects most faulty versions of these programs. In the second study, we compared two testing strategies, i.e., combinatorial testing and random testing, in terms of their stability. The effectiveness of each testing strategy is measured in terms of the code coverage and fault detection. The results of our study suggest that in most cases, combinatorial testing performed as good as or better than random testing.

1.2 Summary of publications

This dissertation is presented in an article-based format and includes six research papers. In Chapter 2, we present the paper titled, "Identifying failure inducing combination in a combinatorial test set", which was published in IEEE fifth International Conference on Software Testing, Verification and Validation (ICST), in 2012. The paper reports our work on the first phase of BEN.

Chapter 3 presents the paper titled, "Fault localization based on failure inducing combinations". The paper was published in IEEE 24th International Symposium on Software Reliability Engineering (ISSRE) in 2013. The paper presents our work on the second phase of BEN.

The two approaches presented in Chapter 2 and Chapter 3 are combined and revised to form an extended version of BEN. This version is presented in Chapter 4 by the paper titled "A combinatorial testing-based approach to fault localization", which is submitted in January 2016 and is currently under review. The extended version of BEN supports systems whose inducing combinations are larger than the strength of the initial combinatorial test set. Moreover, by revising the stopping condition, BEN can identify an inducing combination by generating a less number of tests in comparison with the approach presented in Chapter 2. Moreover, additional experiments are conducted to further evaluate the effectiveness of BEN.

Chapter 5 presents a tool paper titled “BEN: A combinatorial testing-based fault localization tool”, which was published in IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2015. In this paper, we presented the major user scenarios and also the architectural design of BEN. BEN provides both Graphical User Interface and Command Line Interface.

The first empirical study is presented in Chapter 6, using the paper titled, “Applying combinatorial testing to the Siemens suite”. The paper presented in IEEE 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2013. In this chapter, we reported an experiment that applies combinatorial testing to the Siemens suite. The chapter describes the details of our three-step modeling process. Note that the Siemens suite has been widely used as a benchmark to evaluate the effectiveness of many testing and fault localization techniques. We also used the Siemens suite to evaluate our proposed approaches in Chapter 3 and Chapter 4.

Chapter 7 presents a paper titled “An empirical comparison of combinatorial and random testing”. The paper was published in IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2014. In this paper we compared the stability of combinatorial testing to that of random testing and also measure it in terms of both code coverage and fault detection. Our experimental results show that in most cases, combinatorial testing performed as good as or better than random testing. There are few cases where random testing performed better, but with a very small margin.

Finally, in Chapter 8 we provide the concluding remarks and discuss several directions for our future work.

Chapter 2. Identifying failure inducing combinations in a combinatorial test set

The chapter contains a paper published in IEEE fifth International Conference on Software Testing, Verification and Validation (ICSE), in 2012.

Identifying failure inducing combinations in a combinatorial test set*

Laleh Shikh Gholamhossein Ghandehari[1], Yu Lei[1], Tao Xie[2], Richard Kuhn[3], Raghu Kacker[3]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA

[2]Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695, USA

[3]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA

Abstract - A **t**-way combinatorial test set is designed to detect failures that are triggered by combinations involving no more than **t** parameters. Assume that we have executed a **t**-way test set and some tests have failed. A natural question to ask is: what combinations have caused these failures? Identifying such combinations can facilitate the debugging effort, e.g., by reducing the scope of the code that needs to be inspected.

In this paper, we present an approach to identifying failure-inducing combinations, i.e., combinations that have caused some tests to fail. Given a **t**-way test set, our approach first identifies and ranks a set of suspicious combinations, which are candidates that are likely to be failure-inducing combinations. Next, it generates a set of new tests, which can be executed to refine the ranking of suspicious combinations in the next iteration. This process can be repeated until a stopping condition is satisfied. We conducted an experiment in which our approach was applied to several benchmark programs. The

* Copyright © 2012 IEEE. Reprinted, with permission, from Laleh Sh. Ghandehari, Yu Lei, Tao Xie, Richard Kuhn, Raghu Kacker Identifying failure inducing combinations in a combinatorial test set, IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2012.

experimental results show that our approach can effectively and efficiently identify failure-inducing combinations in these programs.

Keywords- Combinatorial Testing, Fault Localization, Debugging.

2.1 INTRODUCTION

Combinatorial testing has been shown to be a very practical and efficient testing strategy [2, 3, 6]. The main idea behind combinatorial testing is the following: while the behavior of a system as a whole may be affected by many parameters, many failures are caused by interactions of only a few parameters [5]. It is, however, not known a priori interactions of which parameters could cause a failure. A t-way combinatorial test set is designed to cover all the t-way interactions, i.e., combinations of values involving t parameters, where t is typically a small integer [2, 6]. If the input parameters are modeled properly, a t-way test set is guaranteed to detect all the failures that are triggered by interactions of no more than t parameters.

Assume that we have executed a t-way test set and some tests have failed. A natural question to ask is: what combinations have caused these failures? Identifying such combinations can facilitate the debugging effort, e.g., by reducing the scope of the code that needs to be inspected.

In this paper, we present an approach to identifying failure-inducing combinations in a combinatorial test set. A failure-inducing combination, or simply an inducing combination, is a combination of parameter values such that all test cases containing this combination fail [8, 10, 13]. Our approach takes as input a combinatorial test set and produces as output a ranking of t-way suspicious combinations in terms of their likelihood to be inducing. Moreover, our approach identifies all the suspicious combinations whose size is smaller than t, if they exist.

Our approach adopts an iterative framework. At each iteration, a set F of test cases is analyzed. (F is a t-way test set at the first iteration.) Our approach first identifies the set π of all t-way suspicious combinations, and then ranks them based on their likelihood to be inducing. Next, our approach generates a set F' of new test cases. The test cases in F', if executed, will be added to F, and will be analyzed in the next iteration to refine the set of suspicious combinations and their ranking. This process is repeated until a stopping condition is satisfied.

The novelty of our approach lies in the fact that we rank suspicious combinations based on two notions: suspiciousness of a combination and suspiciousness of the environment of a combination. Informally, the environment of a combination consists of other parameter values that appear in the same test case. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. Moreover, new test cases are generated for the most suspicious combinations. Let f be a new test case generated for a suspicious combination c. Test f is generated such that the suspiciousness of the environment for c is minimized. If f fails, it is more likely to be caused by c instead of other values in f.

We report an experiment in which we apply our approach to a set of six third-party benchmark programs. Each benchmark program has a number of seeded faults. The results show that our approach is effective in identifying inducing combinations. On one hand, truly inducing combinations are ranked to the top very quickly. On the other hand, combinations that are ranked on the top but are not failure-inducing often have a very high probability to be inducing. Our approach is also very efficient in that only a very small percentage of all possible test cases need to be executed. For example, for one version of the six benchmark program (version 3 of a program named *cmdline*), the only two inducing combinations are ranked to the top 10 after executing 0.034% of all possible test cases.

The remainder of this paper is organized as follows. Section 2.2 represents the definitions and notations used in this paper. Section 2.3 describes our approach. Section 2.4 gives an example to illustrate our approach. Section 2.5 reports an experiment that demonstrates the effectiveness and efficiency of our approach. Section 2.6 discusses existing work on identifying inducing combinations. Section 2.7 provides some concluding remarks.

2.2 PRELIMINARIES

In this section, we introduce the basic definitions and assumptions needed in our approach.

2.2.1 Basic concepts

Assume that the system under test (SUT) has k input parameters, denoted by set $\mathrm{P} = \{\mathrm{p}_1, \mathrm{p}_2, \dots, \mathrm{p}_\mathrm{k}\}$. Let d_i be the domain of parameter p_i. That is, $\mathrm{d}_\mathrm{i}$contains all possible values that p_i could take, and let $\mathrm{D} = \{\mathrm{d}_1 \cup \mathrm{d}_2 \cup \dots \cup \mathrm{d}_\mathrm{k}\}$.

Definition 1. (Test Case) A test case is a function that assigns a value to each parameter. Formally, a test case is a function $\mathrm{f}: \mathrm{P} \to \mathrm{D}$.

We use Γ to represent all possible test cases for the SUT. It is clear that $|\Gamma| = |\mathrm{d}_1| \times |\mathrm{d}_2| \times \dots \times |\mathrm{d}_\mathrm{K}|$.

Definition 2. (Test Oracle) A test oracle determines whether the execution of a test case is “pass” or “fail”. Formally, a test oracle is a function $\mathrm{r}: \Gamma \to \{\mathrm{pass}, \mathrm{fail}\}$.

Definition 3. (Combination) A combination c is a test case f restricted to a non-empty, proper subset M of parameters in P. Formally, $\mathrm{c} = \mathrm{f}|_\mathrm{M}$, where $\mathrm{M} \subset \mathrm{P}$, and $|\mathrm{M}| > 0$.

In the preceding definition, M is a proper subset of P thus a test case is not considered to be a combination in this paper. We use $\mathrm{dom}(\mathrm{c})$ to denote the domain of c, which is a set of parameters involved in c. (Note that $\mathrm{dom}(\mathrm{c})$ is the domain of a function, which is different from the domain of a parameter.) We define the size $[c]$ of a combination c to be the number of parameters involved in c. That is $[c] = |\mathrm{dom}(c)|$.

A combination of size 1 is a special combination, which we refer to as a component. Since there is only one parameter involved, we denote a component o as an assignment, i.e., $o = p \leftarrow v$, where $o(p) = v$.

Definition 4. (Component Containment) A component $o = p \leftarrow v$ is contained in a combination c denoted by $o \in c$, if and only if $p \in \mathrm{dom}(c)$ and $c(p) = v$.

Definition 5. (Combination Containment) A combination c is contained in a test case f, denoted by $c \subset f$, if and only if $\forall p \in \mathrm{dom}(c), f(p) = c(p)$.

Definition 6. (Inducing Combination) A combination c is failure-inducing if any test case f in which c is contained fails. Formally, $\forall f \in \Gamma: c \subset f \Longrightarrow r(f) = \mathrm{fail}$.

Definition 6 is consistent with the definition of inducing combinations in previous work [8, 9, 10, 13].

Definition 7. (Inducing Probability) The inducing probability of a combination c is the ratio of the number of all possible failed test cases containing c to the number of all possible test cases containing c. The inducing probability is computed by

$$\frac{|\{f \in \Gamma | r(f) = \mathrm{fail} \wedge c \subset f\}|}{|\{f \in \Gamma | c \subset f\}|}$$

The computation of inducing probabilities requires all possible test cases containing a combination; such represent is not possible in practice. This notion is mainly used to evaluate the goodness of our experimental results.

Definition 8. (Suspicious Combination) A combination c is a suspicious combination in a test set $F \subseteq \Gamma$ if c is contained only in failed test cases in F. Formally, $\forall f \in F: c \subset f \Rightarrow r(f) = \mathrm{fail}$.

Inducing combinations must be suspicious combinations, but suspicious combinations may or may not be inducing combinations.

2.2.2 Assumption

Assumption 1. The output of the SUT is deterministic. In other words, the SUT always produces the same output from a given test case.

Assumption 2. There exists a test oracle that determines the status of a test execution, i.e., “pass” or “fail”. Assumption 2 is made to simplify the presentation of our approach. The construction of a test oracle is an independent research problem. When a test oracle exists, our approach can be fully automated. When a test oracle does not exist, our approach can still be applied, but the user needs to assist in determining the execution status of a test case.

Assumption 3. Inducing combinations should involve no more than t parameters, where t is the strength of the initial combinatorial test set.

Our approach focuses on detecting inducing combinations that are of size t or less. Such focus is consistent with the implicit assumption held when a tester decides to use a t-way combinatorial test set.

2.3 APPROACH

In this section, we present our approach to identifying inducing combinations.

2.3.1 Framework

As shown in Figure 2-1, the framework consists of three main steps. (1) Rank generation: In this step, we first identify all the t-way suspicious combinations in F (line 4). We then produce a ranking of the suspicious combinations (line 7). (2) Test generation: In this step, we generate a set of new tests, which will be used to refine the ranking of suspicious combinations in the next iteration (line 9). (3) Reduction: In this step, we analyze the final ranking of t-way suspicious combinations to derive suspicious combinations of size smaller than t, if they exist (line 17). The details of these three steps are presented in the following subsections.

In the framework, the two steps, rank generation and test generation, are performed iteratively when the set of suspicious combinations, π, is not empty and the size of π in the current iteration is less than the previous iteration (line 5). Otherwise, the algorithm stops (lines 12, 14).

In addition, another stopping condition happens when a combination is marked as an inducing combination by the test generation step (line 15). The reduction step analyzes π to determine smaller suspicious combinations and produce a ranking for them (line 17).

The user can decide to stop at the end of each iteration, if the resource is limited.

```
Algorithm IdentifyInducingCombinations
Input: sut, $F_0$, t
Output: a set $R = \{R_1, R_2, \dots R_t\}$ of rankings,
        where $R_i$ is the ranking of i-way suspicious combinations

1.  let $F = F_0$ and let π be an empty set
2.  while (true) {
3.    // Step 1. rank suspicious combinations
4.    identify the set π′ of t-way suspicious combinations in F
5.    if ( π′ != empty && ( |π′| < |π| ){
6.      $\pi = \pi'$
7.      produce a ranking R of all the t-way combinations in π
8.      // Step 2. generate new tests
9.      generate a set F' of new tests
10.     $F = F \cup F'$
11.   }
12.   else if ( π′= empty )
13.     return an empty set of rankings;
14.   if (|π′| = |π|
15.       || any combination marked as inducing) {
16.     // Step 3. derive smaller combinations
17.     derive $R_1, R_2, \dots R_t$ based on π′
18.     return $\{R_1, R_2, \dots R_t\}$
19.   }
20. } // end of while
```

Figure 2-1. Algorithm for identifying inducing combinations

2.3.2 Rank generation

In step of rank generation, we first identify the set π' of all t-way suspicious combinations in F. In the first iteration, F is the initial t-way test set, i.e., F_0. Thus, F_0 covers all t-way combinations. Initially, π'contains all the t-way combinations. We then check each t-way combination c in π'. If c appears in at least one passed test, c is removed from π'. In the subsequent iterations, we do not have to re-compute π' from the scratch. Instead, we only need to remove from π' all the combinations contained in newly added, passed tests.

We next discuss how to rank the suspicious combinations in π. First, we introduce three important metrics of suspiciousness, suspiciousness of component, suspiciousness of combination, and suspiciousness of environment.

Suspiciousness of component (ρ): This notion is defined such that the higher ρ a component o has, the more likely o contributes to a failure, and the more likely o appears in an inducing combination. Let F be the test set that is analyzed in the current iteration. In our approach, ρ is computed by the following formula:

$$\rho(\mathrm{o}) = \frac{1}{3}\big(\mathrm{u(o) + v(o) + w(o)}\big) \qquad (1)$$

Where

$$\mathrm{u(o)} = \frac{|\{\mathrm{f} \in \mathrm{F_i} | \mathrm{r(f) = fail} \ \wedge \mathrm{o} \in \mathrm{f}\}|}{|\{\mathrm{f} \in \mathrm{F_i} | \mathrm{r(f) = fail}\}|}$$

$$\mathrm{v(o)} = \frac{|\{\mathrm{f} \in \mathrm{F_i} | \mathrm{r(f) = fail} \ \wedge \mathrm{o} \in \mathrm{f}\}|}{|\{\mathrm{f} \in \mathrm{F_i} | \mathrm{o} \in \mathrm{f}\}|}$$

$$\mathrm{w(o)} = \frac{|\{\mathrm{c} | \mathrm{o} \in \mathrm{c} \ \wedge \mathrm{c} \in \pi\}|}{|\pi|}$$

The first factor of (1), $\mathrm{u(o)}$, shows the ratio of the number of failed test cases in which component o appears over the total number of failed test cases. The second factor, $\mathrm{v(o)}$, shows the ratio of the number of failed test cases in which component o appears over the total number of test cases in which component o appears. The third factor shows the ratio of the number of suspicious combinations in which component o appears over the total number of suspicious combinations. The three factors are averaged to produce a value between 0 and 1.

The motivation behind the first two factors is that the more frequently a component appears in failed test cases, this component is more likely to contribute to a failure.

There is an important difference between the two factors. Since the greater the domain size is, the less frequently each individual value of this parameter appears in a test set and consequently in failed test cases, the first factor, $\mathrm{u(o)}$, has a bias towards smaller domain size parameters. The second factor, $\mathrm{v(o)}$, is brought in to reduce this bias.

The motivation for the third factor is that components of inducing combinations tend to appear more frequently in suspicious combinations. For example, assume that combination $\mathrm{c = (a \leftarrow 0, b \leftarrow 0)}$ is inducing. Let $\mathrm{f = (a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 0)}$ be a test case. Test case f fails as it contains c. Let $\mathrm{f' = (a \leftarrow 1, b \leftarrow 1, c \leftarrow 0, d \leftarrow 0)}$ be another test case, which passed since it does not contain c. The set of suspicious combinations derived from these two test cases is $\mathrm{\pi = \{(a \leftarrow 0, b \leftarrow 0), (a \leftarrow 0, c \leftarrow 0), (a \leftarrow 0, d \leftarrow 0), (b \leftarrow 0, c \leftarrow 0), (b \leftarrow 0, d \leftarrow 0)\}}$

In this set, the frequencies of $\mathrm{a \leftarrow 0}$ and $\mathrm{b \leftarrow 0}$ both are greater than others. The reason is that $\mathrm{(c \leftarrow 0, d \leftarrow 0)}$ appears in $\mathrm{f'}$, which is a passed test case.

Suspiciousness of combination (ρ_c): Suspiciousness of a combination c is defined to be the average of suspiciousness of components that appear in c. Formally suspiciousness of combination c, $\rho_c(\mathrm{c})$ is computed by

$$\rho_c(\mathrm{c}) = \frac{1}{[\mathrm{c}]} \sum_{\forall\, \mathrm{o} \in \mathrm{c}} \rho(\mathrm{o}) \qquad (2)$$

Suspiciousness of Environment (ρ_e): The environment of a combination c in a test f includes all components that appear in f but do not appear in c. The suspiciousness of the environment of a combination c in a test f is the average suspiciousness of the components in the environment of c. If there is more than one (failed) test containing c in a test set, the suspiciousness of the environment of c in this test set is the minimum suspiciousness of environment of c in all the tests containing c. Formally, suspiciousness of the environment is computed by

$$\rho_{\mathrm{e}}(\mathrm{c}) = \mathrm{Min}\left(\sum_{\mathrm{o}\in\mathrm{f}\,\wedge\mathrm{o}\notin\mathrm{c}} \rho(\mathrm{o}),\ \forall \mathrm{f} \in \mathrm{F}\right) \qquad (3)$$

Now we discuss how to actually rank the suspicious combinations based on ρ_{c} and ρ_{e}. Intuitively, the higher the value of ρc, the lower the value of ρe, the higher a combination is ranked.

To produce the final ranking, we first produce two rankings R_{c} and R_{e} of suspicious combinations, where R_{c} is in the non-descending order of ρc and R_{e} is in the non-ascending order of ρe. The final ranking R is produced by combining R_{c} and R_{e} as follows. Let c and c' be two suspicious combinations. Assume that c has ranks r_{c} and r_{e} in R_{c} and R_{e}, respectively, and c' has ranks r'_{c} and r'_{e} in R_{c} and R_{e}, respectively. In the final ranking R, c is ranked before c' if and only if $\mathrm{r}_{\mathrm{c}} + \mathrm{r}_{\mathrm{e}} < \mathrm{r}'_{\mathrm{c}} + \mathrm{r}'_{\mathrm{e}}$.

2.3.3 Test generation

The step of test generation is responsible for generating new test cases for a predefined number of top suspicious combinations. These new test cases are used to refine the ranking of suspicious combinations in the next iteration. Let c be a suspicious combination. A new test f is generated for c such that f contains c and the suspiciousness of the environment for c is minimized in f. When such a test case passes, this combination is removed from the suspicious set. When such a test case fails, the failure is more likely due to this combination since the suspiciousness of its environment is minimized.

One algorithm to find a new test case with minimum ρ_e for a suspicious combination is to generate all possible tests containing this combination, remove tests which already exist in F, and then select one with minimum ρ_e. This algorithm is very expensive. We next describe a more efficient, but heuristic, algorithm.

First, we generate a base test f as follows. For each parameter involved in c, we give the same value in f as in c. Doing so makes sure that f contains c. For each parameter in the environment of c, i.e., each parameter that is not involved in c, we choose a value (or component) whose suspiciousness ρ is the minimum. If there is more than one value with minimum ρ, one of them is selected randomly.

Next, we check whether the base test f is really new, i.e., making sure that f has not been executed before. If so, f is returned as the new test that contains c and has minimum ρ_e. If not, we pick one parameter randomly and change its value to a value with the next minimum ρ. Again, this test is checked to see whether it is a new test. These steps are repeated until a new test is found, or the number of attempts for finding new test case reaches a predefined number. In the latter case, the combination c is marked as an inducing combination, because it is very likely that all the test cases containing this combination have been executed (and all of them must have failed).

2.3.4 Reduction

In the step of reduction, the set of t-way suspicious combinations is analyzed to derive suspicious combinations of smaller size, i.e., size 1 to $t - 1$. A k-way combination c, where $1 \leq k \leq t - 1$, is suspicious if all the (k+1)-way combinations containing c are suspicious.

Our reduction algorithm works as follows. A bucket is assigned to each $(t - 1)$-way combination c to hold t-way suspicious combinations that contain c. For each t-way suspicious combination in π, we put it into t buckets, one for each $(t - 1)$-way combination that it contains. A $(t - 1)$-way combination c is identified to be a suspicious combination if the number of t-way combinations in its bucket is equal to the number of all possible t-way combinations containing c.

After all the $(t - 1)$-way suspicious combinations are identified, they are ranked using the same algorithm for ranking t-way suspicious combinations.

The similar process can be applied to derive suspicious combinations of size $t - 2$, and so on, until we derive suspicious combinations of size 1.

2.3.5 Stopping condition

There are three stopping conditions in Figure 2-1. The first condition is that π becomes empty. This situation occurs when all inducing combinations are of size greater than t. In this situation, assumption 3 is not satisfied. In this situation, no rankings of suspicious combinations are produced.

The second condition is that the size of π does not change from the previous iteration. This situation occurs when all the new tests generated in the previous iteration fail, and thus no suspicious combination is removed from π.

The third stopping condition is that the framework finds a suspicious combination marked as an inducing combination. These combinations are marked in test generation step, when no new test is found for them.

2.3.6 Discussion

Our approach is by nature heuristic. On one hand, suspicious combinations that are ranked top by our approach may not be truly inducing. On the other hand, truly inducing combinations may not be ranked top by our approach.

While our approach focuses on analyzing t-way combinations, it guarantees to identify inducing combinations involving no more than t parameters to be suspicious combinations. Let c be an inducing combination, we consider the following two cases.

Case (1): c is a t-way combination. As the initial test set is a t-way test set, there is at least one test that contains c, and all test cases containing c must fail, since c is inducing. Therefore, c is identified to be a suspicious combination by our approach.

Case (2): The size of c is less than t. All t-way combinations containing c are inducing combinations and are identified to be suspicious combinations. Hence, the reduction step identifies c as a suspicious combination.

Note that when an inducing combination involves more than t parameters, it may not appear in the initial t-way test set, and our algorithm does not identify it to be a suspicious combination.

2.3.7 Complexity analysis

Let k be the number of parameters, d the largest domain size and n the number of test cases in the test set. The maximum number of t-way combinations is $m = \binom{k}{t}d^t$.

The rank generation step needs to sort the set of suspicious combinations for three times, once for each ranking R_c, R_e, and R. The sorting dominates the complexity of this step, which is $O(m * \log m)$.

The test generation step needs to select $(k - t)$ values with minimum ρ first, which takes $(k - t) * O(d)$. Then it needs to check whether it is new, which is $O(k * n)$. In the worst case, a new test is not found after a predefined number of attempts. Thus the complexity for this step is $(k - t) * O(d) * O(k * n)$.

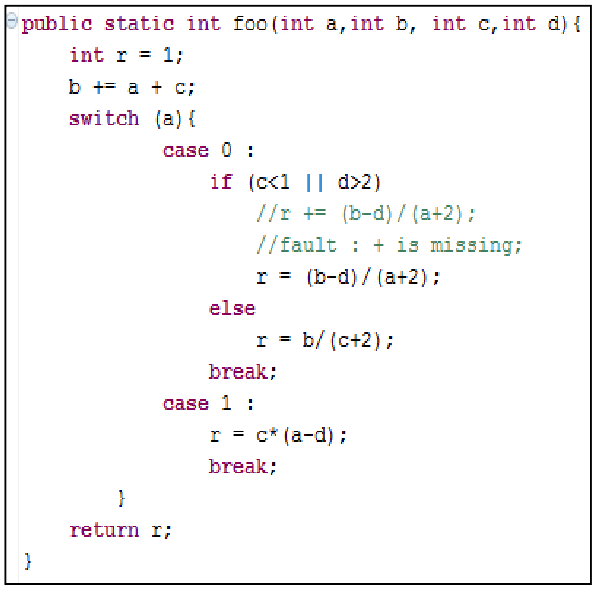
In the reduction step, each t-way suspicious combination is put into t buckets. It takes $O(t)$ to determine whether a t-way combination belongs to a particular bucket. There are $l = \binom{k}{t-1}d^{t-1}$ buckets. So the complexity for all t-way combinations is $O(t * l * m)$. This computation is performed for 1 to $(t - 1)$-way combinations, and the total complexity is

$O(t^2 * l * m)$. The reduction step ranks suspicious combinations, which is however dominated by finding suspicious combinations.

2.4 EXAMPLE

In this section, we illustrate our approach using an example program, which is shown in Figure 2-2. Method *foo* has a fault in line 6. The correct statement should be $r += (b - d)/(a + 2)$, but operator "+" is missing. The input parameter model consists of $P = \{a, b, c, d\}$, and $d_a = \{0,1\}$, $d_b = \{0,1\}$, $d_c = \{0,1,2\}$, and $d_d = \{0,1,2,3\}$. The faulty statement is reachable with a test f such that (1) f(a) = 0; and (2) f(c) = 0 or f(d) = 3. So the inducing combinations are $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.

Suppose that the program is tested by a two-way test set. The result of the test executions is shown in Table 2-1, where 3 out of 12 tests fail. Test cases #1 and #7 fail because they contain combination $(a \leftarrow 0, c \leftarrow 0)$. Test case #10 fails because it contains $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.



```
public static int foo(int a,int b, int c,int d){
    int r = 1;
    b += a + c;
    switch (a){
            case 0 :
                if (c<1 || d>2)
                    //r += (b-d)/(a+2);
                    //fault : + is missing;
                    r = (b-d)/(a+2);
                else
                    r = b/(c+2);
                break;
            case 1 :
                r = c*(a-d);
                break;
        }
    return r;
}
```

Figure 2-2. An example of faulty program

Table 2-1. Two-way test set and their status

Test #	a	b	c	d	Status
1	0	0	0	0	fail
2	1	1	1	0	pass
3	0	1	2	0	pass
4	1	0	0	1	pass
5	0	0	1	1	pass
6	1	1	2	1	pass
7	0	1	0	2	fail
8	1	0	1	2	pass
9	0	0	2	2	pass
10	0	1	0	3	fail
11	1	0	1	3	pass
12	1	0	2	3	pass

Our approach takes Table 2-1 as input. Nine suspicious two-way combinations are identified, and are listed in the first column of Table 2-2. Then our approach computes the suspiciousness of all the components (seven) that appear in a suspicious combination.

For example, component $\mathrm{c} \leftarrow 0$ appears in 3 failed test cases while there are 3 failed test cases, so $\mathrm{u}(\mathrm{c} \leftarrow 0) = 1$. The frequency of $\mathrm{c} \leftarrow 0$ in the test set is 4, so $\mathrm{v}(\mathrm{c} \leftarrow 0) = 3/4$; 5 out of 9 members of suspicious combinations set contain $\mathrm{c} \leftarrow 0$, so $\mathrm{w}(\mathrm{c} \leftarrow 0) = 5/9$.

The computations for all components are as follows:

$$\rho(c \leftarrow 0) = \frac{1}{3} * \left(1 + \frac{3}{4} + \frac{5}{9}\right) = 0.7685$$

$$\rho(d \leftarrow 0) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$

$$\rho(d \leftarrow 2) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$

$$\rho(d \leftarrow 3) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{3} + \frac{3}{9}\right) = 0.3333$$

$$\rho(b \leftarrow 0) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{7} + \frac{1}{9}\right) = 0.1958$$

$$\rho(b \leftarrow 1) = \frac{1}{3} * \left(\frac{2}{3} + \frac{2}{5} + \frac{3}{9}\right) = 0.4667$$

$$\rho(a \leftarrow 0) = \frac{1}{3} * \left(1 + \frac{3}{6} + \frac{2}{9}\right) = 0.5741$$

According to formula (2), ρ_{c} for a suspicious combination c is the average suspiciousness of the components that c contains. For example, in combination$(\mathrm{a} \leftarrow 0, \mathrm{c} \leftarrow 0)$, $\rho_{\mathrm{c}}$is $(0.5741 + 0.7685)/2 = 0.6713$. After computing ρ_{c} for all suspicious combinations, we ranked them based on the non-ascending order of ρ_{c}. The values of ρ_{c} and R_{c} for each suspicious combination are shown in the second and third columns of Table 2-2.

Table 2-2. Suspicious combinations and their corresponding values

Suspicious Combination	$\boldsymbol{\rho_c}$	$\boldsymbol{R_c}$	$\boldsymbol{\rho_e}$	$\boldsymbol{R_e}$	$\boldsymbol{R_c + R_e}$	$\boldsymbol{R}$	New test case	Status
$a \leftarrow 0, c \leftarrow 0$	0.6713	1	0.2460	1	2	1	$(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$	fail
$b \leftarrow 1, c \leftarrow 0$	0.6176	2	0.4352	3	5	2	$(a \leftarrow 1, b \leftarrow 1, c \leftarrow 0, d \leftarrow 1)$	pass
$c \leftarrow 0, d \leftarrow 0$	0.5324	4	0.3849	2	6	3	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 0)$	pass
$c \leftarrow 0, d \leftarrow 3$	0.5509	3	0.5204	4	7	4	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 3)$	pass
$c \leftarrow 0, d \leftarrow 2$	0.5324	4	0.5204	4	8	5	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 2)$	pass
$a \leftarrow 0, d \leftarrow 3$	0.4537	5	0.6176	5	10	6	$(a \leftarrow 0, b \leftarrow 0, c \leftarrow 2, d \leftarrow 3)$	fail
$b \leftarrow 1, d \leftarrow 3$	0.4000	6	0.6713	6	12	7	$(a \leftarrow 1, b \leftarrow 1, c \leftarrow 1, d \leftarrow 3)$	pass
$b \leftarrow 1, d \leftarrow 2$	0.3815	7	0.6713	6	13	8	$(a \leftarrow 1, b \leftarrow 1, c \leftarrow 1, d \leftarrow 2)$	pass
$b \leftarrow 0, d \leftarrow 0$	0.2460	8	0.6713	6	14	9	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 2, d \leftarrow 0)$	pass

Next we compute ρ_e for each suspicious combination using formula (3). For example, there are three test cases, test #1, test #7, and test #10, that contain $(a \leftarrow 0, c \leftarrow 0)$. Therefore,

$$\rho_e(a \leftarrow 0, c \leftarrow 0) = \min\left(\left(\frac{\rho(b\leftarrow 0)+\rho(d\leftarrow 0)}{2}\right) = 0.2460\,, \left(\frac{\rho(b\leftarrow 1)+\rho(d\leftarrow 2)}{2}\right) = 0.3815, \left(\frac{\rho(b\leftarrow 1)+\rho(d\leftarrow 3)}{2}\right) = 0.4000\right) = 0.2460$$

Next we rank suspicious combinations by a non-descending order of ρ_e, as shown in column R_e of Table 2-2.

Finally, the two rankings in columns R_c and R_e are combined to produce a final ranking of the suspicious components (column R). In this final ranking, inducing combination $(a \leftarrow 0, c \leftarrow 0)$ is ranked on the top, and the other $(a \leftarrow 0, d \leftarrow 3)$ is ranked 6th.

Then new tests are generated for the most suspicious combinations. For suspicious combination $(a \leftarrow 0, c \leftarrow 0)$, we assign values to parameters in its environment, i.e., b and d, such that the suspiciousness of each value is minimum. For b, 0 is selected, as$\min(\rho(b \leftarrow 0) = 0.1958\,, \rho(b \leftarrow 1) = 0.4667) = 0.1958$. For d, 1 is selected as $\min(\rho(d \leftarrow 0) = 0.2963, \rho(d \leftarrow 1) = 0, \rho(d \leftarrow 2) = 0.2963, \rho(d \leftarrow 3) = 0.3333) = 0$. So a new test $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$ is generated.

In this example, we generate a new test case for each suspicious combination since there are only nine combinations. As shown in the last column of Table 2-2, all the new test cases pass except two that contain two inducing combinations. In the next iteration, the combinations that appear in a passed test case are not suspicious anymore. Therefore, all combinations except two inducing combinations are removed from the suspicious combinations set, and this set consists of the two combinations, which are both inducing combinations.

Note that this example represents a best case scenario of our approach. In the next section, we provide an experimental evaluation of our approach.

2.5 EXPERIMENT

We built a prototype tool called BEN that implements our approach. (BEN is a Chinese word that means “root cause”.) We used this tool to conduct an experiment on a set of six benchmark programs.

2.5.1 Experimental design

2.5.1.1 Subject programs

We used six C programs, *count*, *series*, *tokens*, *ntree*, *nametbl*, and *cmdline*, as subject programs [7]. Each of these programs contains some faults. To determine whether a test case fails or passes, we created a fault-free version of each program according to the accompanying fault descriptions.

In combinatorial testing, the result may be different by different ways of modeling the input space. To reduce bias, we used the same models for the six programs as in previous work [13].

Table 2-3 shows properties of subject programs and their input models. The second column (LOC) shows the number of lines of uncommented code in these programs. The third column shows the number of faults. The last column (Input Model) shows the

Table 2-3. Subject programs

Program	LOC	# of faults	Input model
count	42	8	$(2^2 \times 3^4)$
series	288	4	$(2^1 \times 4^2 \times 6^1)$
tokens	192	5	$(2^2 \times 3^2)$
ntree	307	8	(4^4)
nametbl	329	8	$(2^1 \times 3^2 \times 5^2)$
cmdline	336	9	$(2^1 \times 3^4 \times 4^1 \times 6^2 \times 15^1)$

input parameter model of each program, which includes the number of parameters and their domain size. We represent it by $(d_1^{k_1} \times d_2^{k_2} \times \ldots)$, where $d_i^{k_i}$ indicates that there are k_i number of parameters with domain size as d_i. Note that $k_1 + k_2 + \cdots = k$, which is the total number of parameters. For example, *count* has six parameters, among which two parameters have a domain size of two, and four parameters have a domain size of three. More details about these models can be found elsewhere [13].

Each subject program contains multiple faults. Generally speaking, the more faults, the more failure-inducing combinations, and the easier it is to find them. To make the problem more challenging, two additional versions for each program are created; a version with about 50% of faults, and a version with a single fault. We refer to these versions by versions1,2, and 3 respectively. Then we run the tool three times for each program, once for each version.

2.5.1.2 Metrics

To measure the effectiveness of our approach, we compute the percentage of truly inducing combinations in the top 10 ranked suspicious combinations. If a combination in top 10 is not inducing, we also compute its inducing probability.

We measure the efficiency of BEN by the percentage of new test cases generated and number of iterations needed.

For the purpose of the evaluation, in order to detect truly inducing combinations, we run the exhaustive test set. A combination is truly inducing if all possible tests containing this combination fail.

2.5.1.3 Test generation

The initial t-way test set is generated using the ACTS tool [1]. When we generate new tests, we generate a new test for each of the top 10 suspicious combinations.

2.5.2 Results and discussion

We conduct the experiment by taking a 2-way test set as the initial test set, except for version 3 of *series*, where both the 2-way and 3-way tests are used. The reason is that there is no 2-way inducing combination for version 3 of *series*.

The results of our experiment are summarized in Table 2-4. We will not explain the column headers one by one, as they are self-explanatory. We point out that the 7th column (ratio of inducing combinations to all combinations) is intended to show the difficulty of the identification problem. Typically, the fewer inducing combinations, the more effort needed to identify them.

For example, in version 3 of *cmdline*, there are only 2 inducing combinations out of 836 possible 2-way combinations. In version 1 of *count*, every combination is inducing. It is easy to see that identifying inducing combinations in version 1 of *count* is much easier than in version 3 of *cmdline*.

Note that the results for versions 1 and 2 of *tokens* are the same. Version 2 was created by removing 3 out of 5 faults of version 1. However, both versions produce the same output. The reason is that we used the same model as previous work [13], which does not capture the difference between these two versions. Columns 8, 9, and 10 (#of iterations, #of new test cases, and percentage of executed tests to the exhaustive test set) are intended to show the efficiency of our approach. In general, a small percentage of tests need to be executed by our approach. There are a few cases where more than 50% tests were executed. One case is for version 3 of *series* with 3-way test set, we ran 60 % of the test cases, i.e., 116 test cases. However, only 10 new test cases were added by our approach and the other 106 tests were in the initial test set. The other cases are for the

Table 2-4. Experimental results for t-way combinations

Program	Version	Size of exhaustive test set	Size of Initial Test Set	# of all t-way combinations	# of inducing combinations	Ratio of inducing combinations to all combinations	# of iterations	# of new test cases	percentage of executed tests to exhaustive test set	# of suspicious combinations	Percentage of inducing combinations in top 10 suspicious combinations
count	1	324	12	106	106	1	2	10	7%	106	100%
	2				30	0.2830	4	30	13%	45	100%
	3				13	0.1226	2	10	7%	20	100%
series	1	192	24	92	10	0.1087	3	20	23%	32	50%
	2				2	0.0217	4	22	24%	4	50%
	3				0	0	4	6	16%	0	-
			106*	224	6	0.0268	2	10	60%	12	60%
tokens	1	36	9	37	14	0.3784	2	10	53%	21	100%
	2				14	0.3784	2	10	53%	21	100%
	3				8	0.2162	2	10	53%	14	80%
ntree	1	256	16	96	24	0.2500	2	10	10%	48	100%
	2				14	0.1458	3	20	14%	44	60%
	3				2	0.0208	4	22	15%	2	100%
nametbl	1	450	25	126	83	0.6587	2	10	8%	105	100%
	2				30	0.2381	2	10	8%	74	100%
	3				6	0.0476	10	83	24%	6	100%
cmdline	1	349920	95	836	252	0.3014	4	30	0.036%	568	50%
	2				197	0.2356	7	60	0.044%	463	70%
	3				2	0.0024	4	24	0.034%	4	50%

* three-way

three versions of *tokens*. This program has a small number of the exhaustive test cases(36 test cases), and there were 9 tests in the initial test set.

In contrast, for the three different versions of *cmdline*, the largest program, at most 0.044% of possible test cases were executed, but as shown in Figure 2-3, later, we can still rank all the inducing combinations to the top 10.

The last two columns (# *of suspicious combinations* and *percentage of inducing combinations in top 10 suspicious combinations*) show the effectiveness of our approach. For 10 (out of 18) versions of these programs, all the top 10 ranked suspicious combinations are truly inducing. For other versions, Table 2-5 shows the ranks and inducing probabilities of the top 10 ranked suspicious combinations that are not truly inducing. All of these combinations have a very high inducing probability. The lowest inducing probabilities happen in version 3 of *cmdline*, where the 3rd and 4th ranked combinations have an inducing probability of 0.75 and 0.7114, respectively. The highest inducing probabilities happen in version 2 of *cmdline*, where the 4th, 7th, and 9th ranked combinations have an inducing probability that is close to 1, even if they are not truly inducing.

In two versions of *series,* 1st and 3rd (with 3-way test set), and the 3rd version of *tokens,* the third stopping condition is satisfied, and truly inducing combinations are found. So the information of other suspicious combinations is excluded from Table 2-5.

As it is shown in Table 2-4, the set of suspicious combinations becomes empty in version 3 of *series* where 2-way test set is applied. In other cases, reaching the stable point, satisfying the second condition, happens.

The reduction step finds 1-way suspicious combinations in 13 (out of 18) versions; in 8 versions of theses 13 versions, all of top ranked combinations are inducing (Table 2-6).

Table 2-5. Inducing probabilities of top 10 suspicious combinations that are not inducing

Program	Version	Rank	# of possible test cases containing combination	# of possible failed test cases containing this combination	Inducing probability
series	2	1	12	10	0.8333
		2	12	10	0.8333
ntree	2	1	16	15	0.9375
		3	16	15	0.9375
		7	16	13	0.8125
		10	16	15	0.9375
cmdline	1	1	29160	27216	0.9333
		7	11664	10674	0.9151
		8	43740	40824	0.9333
		9	11664	10674	0.9151
		10	29160	25704	0.8815
	2	4	11664	11661	0.9997
		7	11664	11661	0.9997
		9	11664	11655	0.9992
	3	3	7776	5832	0.75
		4	3888	2766	0.7114

For other 5 versions, the ranks and inducing probabilities of non-inducing but suspicious combinations are shown in Table 2-7.

The charts in Figure 2-3 show the distribution of inducing and non-inducing combinations in the ranking of suspicious combinations after each iteration. Due to limited space, we only show the distribution for the 3rd version of each program, except for program *series*, where version 2 is shown. The vertical axis shows the number of iterations. The horizontal axis shows the ranks. Inducing and non-inducing combinations are shown by different colors.

The charts show that our approach can quickly rank all the inducing combinations to the top. For example, *nametbl* has 41 suspicious combinations in the first iteration. There are 6 truly inducing combinations, which are ranked 1 to 4, 7, and 11. In the second iteration, we have 33 suspicious combinations, and 5 out of 6 inducing combinations are ranked to the top 5. In the third iteration, all 6 inducing combinations are ranked to the top 6. Although BEN runs 10 iterations to reach to the stable point, all 6 inducing combinations come to the higher ranks sooner than 10 iterations.

Note that version 2 of *series* only has two inducing combinations, and they are ranked in the 3rd and 4th place.

Table 2-6. Experimental results for (t-1)-way combinations

Program	Version	# of all (t – 1)-way combinations	# of inducing combinations	Ratio of inducing combinations to all combinations	# derived combinations	Percentage of inducing in top 10 derived combinations
count	1	16	16	1	16	100%
	2		2	0.125	2	100%
	3		1	0.0625	1	100%
series	1	16	0	0	1	0%
tokens	1	10	2	0.2	2	100%
	2		2	0.2	2	100%
	3		1	0.1	1	100%
ntree	1	16	2	0.125	2	100%
	2		0	0	1	0%
nametbl	1	18	7	0.3889	10	70%
	2		2	0.1111	2	100%
cmdline	1	45	7	0.1556	13	50%
	2		6	0.1333	7	85%

2.5.3 Threats to validity

Threats to internal validity are other factors that may be responsible for the experimental results, without our knowledge. We have tried to automate the experimental procedure as much as possible, as an effort to remove human errors. In particular, we build clean versions for all six subject programs, and a tool that automatically compares the results of the clean version and a faulty version to determine truly inducing combinations. Further, consistency of the results has been carefully checked to detect potential mistakes made in the experiment.

Table 2-7. Inducing probabilities of Top 10 (t-1)-way suspicious combinations that are not inducing

Program	Version	Rank	# of possible test cases containing combination	# of possible failed test cases containing this combination	Inducing probability
series	1	1	32	30	0.9375
ntree	2	1	64	61	0.9531
nametbl	1	5	90	86	0.9556
		8	90	80	0.8889
		9	90	86	0.9555
cmdline	1	2	23328	22338	0.9576
		3	23328	22338	0.9576
		6	23328	22338	0.9576
		7	23328	22338	0.9576
		10	23328	22368	0.9588
	2	5	23328	23302	0.9988

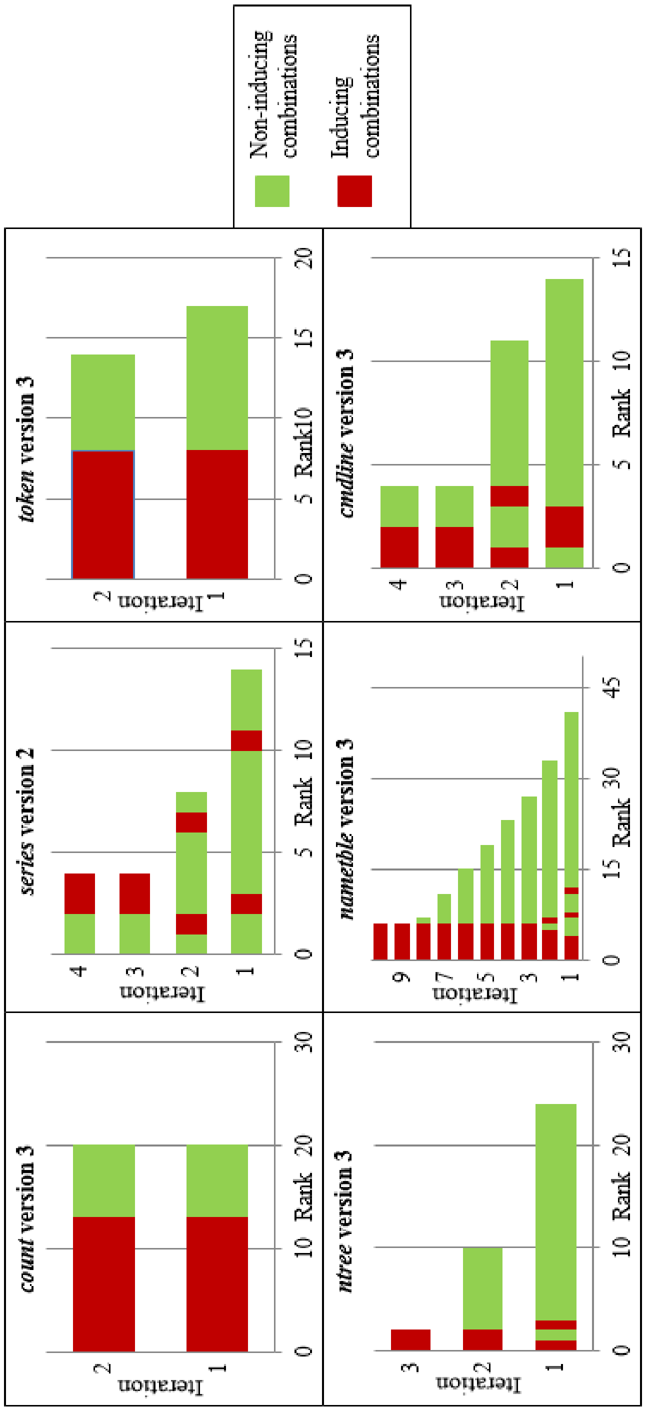


Figure 2-3. Distribution of inducing and non-inducing combinations in suspicious set

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from previous work [7]; these programs are created by a third party and have been used in other studies [13]. But the subject programs are programs of relatively small size with seeded faults. More experiments on larger programs with real faults can reduce external validity of our findings.

The original versions of the subject programs had multiple faults, and thus many inducing combinations. So they could be identified more easily than those for programs with a small number of inducing combinations. To mitigate this threat, we conduct our experiment on 3 versions of each program, with all faults, 50% of faults, and one fault, respectively.

2.6 RELATED WORK

Delta debugging [12] is a technique that tries to find a minimum set of failure-inducing input values in a failed test. It involves systematically changing or removing the values in a failed test to create new tests. Two similar techniques, called FIC and FIC_BS [13], try to identify all the faulty interactions contained in a failed test. The notion of a faulty interaction is the same as the notion of an inducing combination defined in this paper. FIC and FIC_BS assume that no new inducing combinations are introduced when a value is changed to create a new test.

Our approach is different from these previous techniques in that we try to identify inducing combinations in a combinatorial test set, instead of a single failed test. On one hand, a test set contains more information than a single test. On the other hand, doing so makes it possible to identify inducing combinations that appear in different tests. Moreover, the assumption made by FIC and FIC_BS may not hold for many applications, as changing a value in a test introduces many new combinations, and assuming that all of them are non-inducing is over-optimistic.

Yilmaz et al. [11] proposed a machine learning approach to identify likely inducing combinations from a given combinatorial test set. Their approach builds a data structure called classification tree, and assigns a score to each likely inducing combination. A combination is classified to be an inducing combination if its score is greater than a threshold value. This approach is used to guide the generation of new tests in an adaptive combinatorial testing technique [4].

The preceding approach identifies inducing combinations based on a combinatorial test set only, i.e., without adding new tests. Considering that a combinatorial test set is often produced such that it contains as few tests as possible while still achieving t-way coverage, many combinations are covered only once in a combinatorial test set. As a result, a combinatorial test set alone often provides insufficient information for effective classification, especially when there are a large number of inducing combinations and failed test cases.

The work that is mostly related to ours is a technique called AIFL [8, 9]. Given a combinatorial test set, AIFL first identifies all the suspicious combinations, i.e., combinations that only appear in failed tests. Then, for each failed test, AIFL generates k test cases by changing the value of one parameter at a time, where k is the number of parameters. The new value of a changed parameter could be any value in its domain. New tests are then used to refine the set of suspicious combinations. At this point, AIFL stops and outputs the set of suspicious combinations. InterAIFL [10] extends AIFL by adopting an iterative framework. That is, new tests are generated to refine the set of suspicious combinations iteratively until a fixed point is reached, i.e., the set of suspicious combinations becomes stable.

Our approach identifies suspicious combinations in the same way as AIFL and Inter-AIFL. However, our approach goes one step further to produce a ranking of suspicious combinations. This ranking helps the debugging effort to focus on the most suspicious combinations. Our approach also differs significantly in the way of generating new tests: our test generation is based on the notions of suspiciousness combination and suspiciousness of environment.

2.7 CONCLUSION

In this paper, we presented an approach to identifying failure-inducing combinations in a combinatorial test set. Our approach adopts an iterative framework that ranks suspicious combinations and generates new tests repeatedly until a stable condition is reached. The novelty of our approach lies in the fact that we rank suspicious combinations and generate new tests based on the notions of suspiciousness of a combination and suspiciousness of its environment. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. New tests are generated for a user-defined number of most suspicious combinations such that the suspiciousness of the environment of a combination is minimized in each test. Our experimental results show that our approach is very effective in terms of quickly identifying and ranking failure-inducing combinations to the top.

There are two major directions to continue our work. First, we plan to conduct more empirical studies to further evaluate the performance of our approach. In particular, we plan to apply our approach to larger and more complex programs. Second, this work is part of a larger effort to develop fault localization techniques that leverage the result of combinatorial testing. The next step in our project is to go inside the source code and find a particular line or block of code that contains the fault. We believe that failure-inducing combinations provide important insights about how different parameters interact with each other and can be used to reduce the scope of the code that needs to be analyzed in the next step.

2.8 Acknowledgment

We thank Christopher Lott for providing the benchmark programs and Jian Zhang for allowing us to use their combinatorial models. This work is supported by two grants (70NANB9H9178 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST) and a grant (61070013) of National Natural Science Foundation of China.

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

2.9 REFERENCES

1. Advanced Combinatorial Testing System (ACTS), 2010. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.
2. D. Cohen, S. Dalal, M. Fredman, and G. Patton. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 23(7):437–444, 1997.
3. M. B. Cohen, P. B. Gibbons, W.B. Mugridge, C.J. Colbourn. Constructing test suites for interaction testing. In Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pages 38-48, 2003.
4. E. Dumlu, C. Yılmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2011), pages 243-253, 2011.
5. D.R. Kuhn, D.R. Wallace, A.M. Gallo. Software fault interactions and implications for software testing. IEEE Transaction on Software Engineering, 2004, 30: 418–421

6. Y. Lei, R. Kacker, D. Kuhn, V. Okun, J. Lawrence, IPOG/IPOD: Efficient test generation for multi-way software testing, *Journal of Software Testing, Verification, and Reliability*, 18(3):125-148, Sept. 2008.
7. C. Lott. A repeatable software engineering experiment. http://www.maultech.com/chrislott/work/exp.
8. C. Nie, H. Leung, and B. Xu. The minimal failure-causing schema of combinatorial testing. ACM Transactions on Software Engineering and Methodology, Volume 20 Issue 4, September 2011.
9. L. Shi, C. Nie, B. Xu. A software debugging method based on pairwise testing. In Proceedings of the International Conference on Computational Science (ICCS2005), pages 1088-1091, 2005.
10. Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In Proceedings of the 10th International Conference on Quality Software (QSIC 2010), pages 495–502, 2010.
11. C. Yilmaz, M. B. Cohen, A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transaction on Software Engineering, 2006, 32(1): 20-34.
12. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 2002, pages 183–200.
13. Z. Zhang, and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In Proceding of ACM International Symposium on Software Testing and Analysis (ISSTA 2011), pages 331-341, 2011.

Chapter 3. Fault localization based on failure inducing combinations

This chapter contains a paper published in IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), in 2013.

Fault localization based on failure inducing combinations[*]

Laleh Sh. Ghandehari[1], Yu Lei[1], David Kung[1], Raghu Kacker[2], Richard Kuhn[2]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA

laleh.shikhgholamhosseing@mavs.uta.edu, {ylei,kung}@uta.edu

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD, USA raghu.kacker@nist.gov, kuhn@nist.gov

Abstract-Combinatorial testing has been shown to be a very effective testing strategy. After a failure is detected, the next task is to identify the fault that causes the failure. In this paper, we present an approach to fault localization that leverages the result of combinatorial testing. Our approach is based on a notion called failure-inducing combinations. A combination is failure-inducing if it causes any test in which it appears to fail. Given a failure-inducing combination, our approach derives a group of tests that are likely to exercise similar traces but produce different outcomes. These tests are then analyzed to locate the faults. We conducted an experiment in which our approach was applied to the Siemens suite as well as the grep program from the SIR repository that has 10068 lines of code. The experimental results show that our approach can effectively and efficiently localize the faults in these programs.

Keywords- Combinatorial Testing; Fault Localization; Debugging.

[*] Copyright © 2013 IEEE. Reprinted, with permission, from Laleh Sh. Ghandehari, Yu Lei, David Kung, Raghu Kacker and Richard Kuhn, Fault localization based on failure inducing combinations, IEEE International Symposium on Software Reliability Engineering (ISSRE), November 2013.

3.1 INTRODUCTION

Combinatorial testing has been shown to be a very effective testing strategy [7, 14]. The key observation is that most software failures are caused by interactions of only a few parameters. A widely cited NIST study reports that failures in several real-life systems involved no more than six parameters [11, 12]. A t-way combinatorial test set is built to cover all the t-way interactions, where t is typically a small integer. If test parameters and values are correctly modeled, a t-way test set is able to expose all failures involving no more than t parameters. Empirical results have shown that combinatorial testing is very effective for failure detection while significantly reducing the number of tests.

Most research in combinatorial testing has focused on developing efficient combinatorial test generation algorithms and conducting empirical studies to evaluate the failure-detection effectiveness of combinatorial testing [7, 14]. After a failure is detected, the next task is to find the fault that caused the failure. An important research problem to investigate is how to leverage the result of combinatorial testing to locate the faults. Our earlier work in [6] investigated the problem of how to identify failure-inducing combinations in a combinatorial test set. A combination is failure inducing, or simply inducing, if it causes any test in which it appears to fail. In this paper, we address the problem of how to use inducing combinations to locate the faults in the source code.

One common approach to fault localization is based on the notion of a program spectrum. A program spectrum records information about certain aspects of a test execution [20], such as function call counts, program paths, program slices and use-def chains [16]. Examples of spectrum-based methods include Tarantula [10], set union, set intersection, and nearest neighbor [16]. These methods identify faults by comparing the spectrums of passed and failed test executions.

In this paper, we present a spectrum-based approach to fault localization that leverages the notion of an inducing combination. The novelty of our approach is two-fold. First, to the best of our knowledge, our work is the first effort to perform code-based fault localization based on combinatorial testing. Existing work in this area, i.e., fault localization based on combinatorial testing, has only dealt with the problem of how to identify inducing combinations [15, 17, 19, 21].

Second, our approach generates, in a systematic manner, a small group of tests from an inducing combination, such that the execution traces of these tests can be analyzed to quickly locate the faults. This differs from existing spectrum-based approaches which do not deal with the problem of test generation. Instead, they assume the existence of a large number of tests, which are generated randomly and/or using other techniques [10, 16, 20]. In our approach, one of the tests in the group is referred to as the core member, which consists of the inducing combination and produces a failed test execution. The other tests in the group are referred to as the derived members, which are derived from the core member in a way such that they are likely to execute a trace that is very similar to the trace of the core member but produce a different outcome, i.e., a passed execution. The spectrum of the core member is then compared to the spectrum of each derived member to produce a ranking of statements in terms of their likelihood to be faulty.

Our approach is inspired by the notion of nearest neighbor [16]. The key idea of nearest neighbor is that faulty statements are likely to appear in the execution trace of a failed test but not in the execution trace of a passed test that is as similar to this failed test as possible. If two tests are significantly different, they are likely to represent different application scenarios. Thus, the differences in the execution traces of these two tests are likely due to program logic, instead of faults. The novelty of our approach lies in the fact that we generate, in a systematic manner, a failed test, i.e., the core member, and then derive its nearest neighbors from this failed test, i.e., the derived members. This is in contrast with the approach in [16], which executes a large number of tests from which a failed test and its nearest neighbors are selected.

We report an experiment in which we applied our approach to the Siemens suite and the *grep* program in the Software Infrastructure Repository (SIR) [18]. The Siemens suite has been commonly used to evaluate fault localization methods. Each program has a number of faulty versions. The programs in the Siemens suite are, however, relatively small. Thus, we also applied our approach to the *grep* program that has 10068 lines of code [18]. The results show that our approach is effective in localizing faulty statements and also very efficient in that only a small number of tests need to be generated and executed by our approach. For example, one of the programs in the Siemens suite called *replace* has 32 faulty versions. These 32 versions were all killed by a 2-way test set with 192 tests. Our approach identified the faulty statement in each version by generating and executing only about 3 additional tests.

The remainder of this paper is organized as follows. Section 3.2 shows a motivating example. Section 3.3 introduces several definitions and reviews our previous work on identifying inducing combinations. Section 3.4 presents the details of our approach to locating faults based on inducing combinations. Section 3.5 reports the experimental results of applying our approach to the Siemens suite and the *grep* program. Section 3.6 discusses existing work on fault localization. Section 3.7 provides concluding remarks and our plan for future work.

3.2 A MOTIVATING EXAMPLE

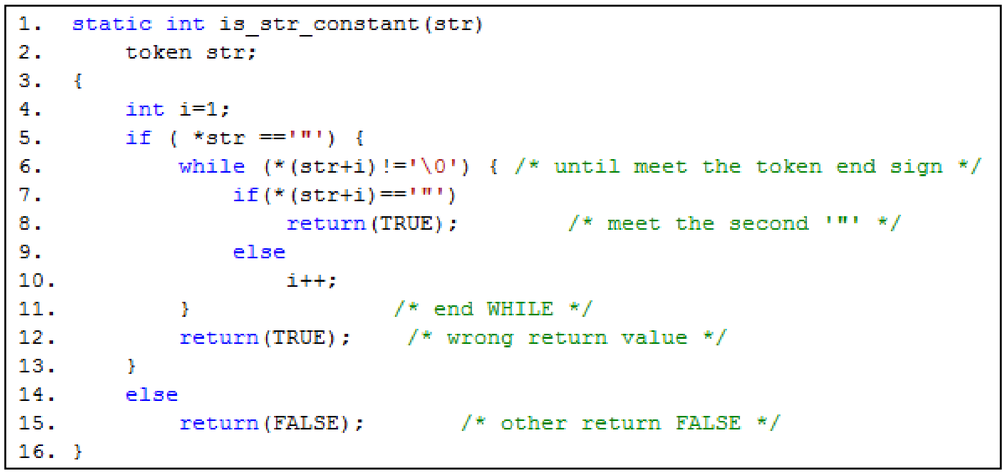
Consider as an example the *printtokens2* program in the Siemens suite [18]. This program is a lexical analyzer that reads an input string and prints out all the tokens in the input string. The types of tokens include *keyword*, *special*, *identifier*, *number*, *comment*, *string_constant* or *character_constant*.

This program works by first extracting all the space-delimited elements in the input string. These elements are then sent to functions that are designed to recognize different types of tokens. One of the faulty versions of this program, i.e., version 5, has the faulty statement in function *is_str_constant*, which is responsible for recognizing whether or not an element extracted from the input string is a *string_constant*. A *string_constant* is a sequence of characters that begins and ends with a double quotation.

Figure 3-1 shows the is_str_constant function. If str does not begin with a double quotation, this function returns false (i.e. the last return statement in Figure 3-1). If str begins with a double quotation, and a second double quotation is found in str, this function returns true (i.e., the first return statement in Figure 3-1). If str begins with a double quotation, and a second double quotation is not found, this function is supposed to return false, but it returns true (i.e., the second return statement in Figure 3-1), which is the faulty statement.

For illustration, two abstract parameters P1 and P2, each of which has two values 0 and 1, are identified to test function *is_str_constant*. P1 indicates whether or not a token begins with a double quotation, and P2 indicates whether or not there exists a second double quotation in a token. Each of these two abstract parameters represents a certain characteristic of the actual parameter *str*. Since the values of these two parameters cannot be directly taken by function *is_str_constant*, they must be mapped to a concrete value of the actual parameter *str*. For example, if both P1 and P2 take value 1, a token such as “test” can be used as the concrete value of *str* to test function *is_str_constant*.

Table 3-1 shows four possible test cases, as well as the actual and expected output of each test case, for function *is_str_constant*. Note that a complete test for the *printtokens2*



```
static int is_str_constant(str)
    token str;
{
    int i=1;
    if ( *str =='"') {
        while (*(str+i)!='\0') { /* until meet the token end sign */
            if(*(str+i)=='"')
                return(TRUE);        /* meet the second '"' */
            else
                i++;
        }              /* end WHILE */
        return(TRUE);    /* wrong return value */
    }
    else
        return(FALSE);       /* other return FALSE */
}
```

Figure 3-1. Function is_str_constant

Table 3-1. Abstract tests for function is_str_constant

P1	P2	Actual Output	Expected Output
1	1	True	True
1	0	True	False
0	1	False	False
0	0	False	False

program would contain values of other parameters, which are not directly related to function *is_str_constant* and thus are not shown in Table 3-1.

When we apply 2-way testing and our earlier approach in [6] to the *printtokens2* program, we identify combination $< \mathrm{p1} \leftarrow 1, \mathrm{p2} \leftarrow 0 >$ to be a failure-inducing combination. This is because this combination represents a string that starts with a double quotation, but does not have a second double quotation. So, every test containing this combination would execute the faulty statement, and thus would fail.

Let f be a (failed) test that contains combination $< \mathrm{P1} \leftarrow 1, \mathrm{P2} \leftarrow 0 >$. All the statements that are executed by f are suspicious statements, which are < 4, 5, 6, 7, 9, 10, 12 >. To locate the true faulty statement, we create two tests, $\mathrm{f}_1$and f_2, which are as similar to f as possible but have a different outcome. Test f_1 is created such that it is the same as f except that we change the value of P1 from 1 to 0. Since combination $< \mathrm{P1} \leftarrow 0, \mathrm{P2} \leftarrow 0 >$ represents a string without any double quotation, test f_1 will execute the last *return* statement (line #15) and thus will pass. The statements that are executed by f_1, are <4, 5, 14 15>. Similarly, test f_2 is created such that it is the same as f except that we change the value of P2 from 0 to 1. Since combination $< \mathrm{P1} \leftarrow 1, \mathrm{P2} \leftarrow 1 >$ represents a string starting with a double quotation and has a second double quotation, test f_2 will execute the first *return* statement (line #8) and thus will also pass. The statements that are executed by f_2, are <4, 5, 6, 7, 8, 9, 10>.

Now we analyze the execution traces of the three tests f, f_1 and f_2. We find that the second *return* statement (line #12) is the only statement that is executed by the failed test f but not by passed tests f_1 and f_2. That is, all the other statements that are executed by f are executed by f_1 and/or f_2. Thus, the second *return* statement (line #12) is identified to be the statement that is most likely to be faulty.

In the rest of this paper, we will describe the details of our approach. We emphasize that the novelty of our work is a systematic approach to generate tests like f, f_1, and f_2 whose execution traces can be analyzed to quickly identify the faulty statement(s).

3.3 PRELIMINARIES

3.3.1 Basic concepts

Assume that the system under test (SUT) has k input parameters, denoted by set $\mathrm{P} = \{\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_\mathrm{k}\}$. Let d_i be the domain of parameter p_i. That is, d_i contains all possible values that p_i could take, and let $\mathrm{D} = \{\mathrm{d}_1 \cup \mathrm{d}_2 \cup \ldots \cup \mathrm{d}_\mathrm{k}\}$. Let S be the set of all the statements in the source code of SUT.

Definition 1. (*Test Case*) A test case, or simply a test, is a function that assigns a value to each parameter. Formally, a test case is a function $\mathrm{f}: \mathrm{P} \to \mathrm{D}$.

We use Γ to represent all possible test cases for the SUT. It is clear that $|\Gamma| = |\mathrm{d}_1| \times |\mathrm{d}_2| \times \ldots \times |\mathrm{d}_\mathrm{K}|$.

Definition 2. (*Test Oracle*) A test oracle determines whether the execution of a test case is "pass" or "fail". Formally, a test oracle is a function $\mathrm{r}: \Gamma \to \{\mathrm{pass}, \mathrm{fail}\}$.

Definition 3. (*Combination*) A combination c is a test f restricted to a non-empty, proper subset M of parameters in P. Formally, $\mathrm{c} = \mathrm{f}|_\mathrm{M}$, where $\mathrm{M} \subset \mathrm{P}$, and $|\mathrm{M}| > 0$.

In the preceding definition, M is a proper subset of P. Thus a test case is not considered to be a combination in this paper. We use $\mathrm{dom}(\mathrm{c})$ to denote the domain of c, which is the set of parameters involved in c. (Note that $\mathrm{dom}(\mathrm{c})$ is the domain of a function, which is different from the domain of a parameter.) We define the size $[\mathrm{c}]$ of a combination c to be the number of parameters involved in c. That is $[\mathrm{c}] = |\mathrm{dom}(\mathrm{c})|$.

A combination of size 1 is a special combination, which we refer to as a component. Since there is only one parameter involved, we denote a component o as an assignment, i.e., $\mathrm{o} = \mathrm{p} \leftarrow \mathrm{v}$, where $\mathrm{o}(\mathrm{p}) = \mathrm{v}$.

Definition 4. (*Component Containment*) A component $\mathrm{o} = \mathrm{p} \leftarrow \mathrm{v}$ is contained in a combination c, denoted by $\mathrm{o} \in \mathrm{c}$, if and only if $\mathrm{p} \in \mathrm{dom}(\mathrm{c})$ and $\mathrm{c}(\mathrm{p}) = \mathrm{v}$.

Definition 5. (*Combination Containment*) A combination c is contained in a test case f, denoted by $c \subset f$, if and only if $\forall p \in dom(c), f(p) = c(p)$.

If a combination c is contained by a test case f, i.e., $c \subset f$, all combinations that are contained by c, are contained by f.

Definition 6. (*Inducing Combination*) A combination c is failure-inducing, or simply inducing, if any test f in which c is contained, fails. Formally, $\forall f \in \Gamma: c \subset f \Longrightarrow r(f) = fail$.

Definition 7. (*Suspicious Combination*) A combination c is a suspicious combination in a test set $F \subseteq \Gamma$ if c is only contained in failed test cases in F. Formally, $\forall f \in F: c \subset f \Rightarrow r(f) = fail$.

If F is a t-way test set, i.e., F covers all the t-way combinations, a t-way inducing combination must be a t-way suspicious combination in F. But a t-way suspicious combination in F may or may not be a t-way inducing combination.

Definition 8. (*Program Spectrum Function*) Let f be a test and $trace(f) \subseteq S$ the set of statements executed by f. (The order in which the statements are executed by f is not significant.) The program spectrum function is a Boolean function γ defined as follows: $\gamma: S \times \Gamma \rightarrow \{true, false\}$, where $\gamma(s, f) = true$ if $s \in trace(f)$ and $\gamma(s, f) = false$ if $s \notin trace(f)$.

In other words, a program spectrum with respect to a test execution is a membership function that determines whether a statement is exercised by this test execution.

3.3.2 Identifying inducing combinations

In our previous work [6], we introduced an approach, BEN, to identify inducing combinations in a combinatorial test set. BEN takes a t-way test set as input and generates a ranking of combinations based on their likelihood to be inducing combinations. The main idea of BEN is based on three notions, suspiciousness of component, suspiciousness of combination, and suspiciousness of environment.

Suspiciousness of Component (ρ): A value between 0 and 1. The higher the suspiciousness value of a component o, denoted as $\rho(o)$, the more likely o contributes to a failure, and the more likely o is contained by an inducing combination.

Suspiciousness of Combination (ρ_c): Suspiciousness of a combination c, $\rho_c(c)$, is defined to be the average of suspiciousness of components that appear in c. Formally,

$$\rho_c(c) = \frac{1}{[c]} \sum \rho(o) \quad \text{for } \forall\, o \in c \qquad (1)$$

Suspiciousness of Environment (ρ_e): The environment of a combination c in a test f includes all components that appear in f but do not appear in c. The suspiciousness of the environment of a combination c in a test f is the average suspiciousness of the components in the environment of c. If there is more than one (failed) test containing c in a test set, the suspiciousness of the environment of c in this test set is the minimum suspiciousness of environment of c in all the tests containing c . Formally , suspiciousness of the environment is computed by

$$\rho_e(c) = \mathrm{Min}\left(\frac{1}{[f]-[c]} \sum \rho(o)\right) \qquad (2)$$

$$\text{for } \forall\, o \in f \wedge o \notin c$$

The final ranking is produced such that the higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked.

Experimental results show that BEN is very effective in identifying inducing combinations. On the one hand, truly inducing combinations are ranked to the top very quickly. On the other hand, combinations that are ranked on the top but are not truly inducing have a high probability of being inducing, i.e., they are very likely to cause a failure if they appear in a test. Our approach is also very efficient in that only a very small percentage of all possible tests need to be executed.

3.4 APPROACH

The input of our approach is taken from the output of our earlier tool called BEN, including a suspicious combination c that is ranked at the top and the suspiciousness value, ρ, of every component (of every parameter) of the subject program. The top-ranked suspicious combination may not be a truly inducing combination. However, as mentioned in the previous section, it is very likely to cause a failure if it appears in a test case. The output is a ranking of statements with their likelihood of being faulty.

Our approach consists of two major steps: (1) Test Generation: In this step, we generate a group of tests. This group consists of one failed test, which is referred to as the core member, and at most t passed tests, which are referred to as the derived members. Each derived member is expected to produce a similar trace as the core member. (2) Rank Generation: In this step, we compare the spectrum of the core member to the spectrum of each derived member, and then produce a ranking of statements in terms of their likelihood of being faulty.

3.4.1 Test generation

Let c be the top-ranked suspicious t-way combination taken as input by our approach. In this step, a group of tests is generated which contains a core member and at most t derived members. The core member f is created such that it contains c and the suspiciousness of environment of c in f is minimized. That is, for each parameter p involved in c, f has the same value for p as c, i.e. $c \subset f$; and for each parameter p that does not appear in c, f takes a value that has the minimum suspiciousness value among all the values of p. As discussed later, the reason why we want to minimize the suspiciousness of the environment of c is to maximize the likelihood of a derived member to be a passing test.

The core member f is very likely to fail, since it contains c, and c is, or very likely to be an inducing combination. In case that f does not fail, we pick a test from the initial t-way

f	$\{\overbrace{o_1, o_2, \dots, o_t}^{c}, o_{t+1}, o_{t+2} \dots, o_k\}$	Core
f_1	$\{\boldsymbol{o'_1}, o_2, \dots, o_t, o_{t+1}, o_{t+2} \dots, o_k\}$	Derived
f_2	$\{o_1, \boldsymbol{o'_2}, \dots, o_t, o_{t+1}, o_{t+2} \dots, o_k\}$	Derived
…	…	…
f_t	$\{o_1, o_2, \dots, \boldsymbol{o'_t}, o_{t+1}, o_{t+2} \dots, o_k\}$	Derived

Figure 3-2. An illustration of how to generate derived members

test set that contains c as the core member. Since c is identified as an inducing combination, there must exist at least one failed test that contains c in the initial test set. (Otherwise, c would not even be a suspicious combination.)

Next we generate t derived members $f_1, f_2 \dots f_t$. A derived member f_i is generated such that it has the same value as f for all parameters except one component of c, which is replaced with another component of the same parameter with the minimum suspiciousness value.

Figure 3-2 shows how derived members are generated from the core member f. Core member f contains k components, $o_1, o_2 \dots, o_k$, where k is the number of parameters. Without loss of generality, assume that the first t components in f, i.e., $o_1, o_2 \dots, o_t$, are in the inducing combination c. Each derived member is different from the core member f, only in one component in the inducing combination c.

On the one hand, a derived member f_i is likely to pass for three reasons. First, the replacement effectively removes combination c from f. Second, it is not likely for f_i to contain other suspicious combinations like c because the new component has the minimum suspiciousness value. Finally the suspiciousness of the environment of c is minimized. If a derived member does fail, we ignore this member. In case that all derived members fail, we

Table 3-2. Suspiciousness of components of an example system

Parameter	value	ρ_c	Parameter	value	ρ_c
a	0	0.57	b	0	0.2
	1	0		1	0.47
c	0	0.77	d	0	0.3
	1	0		1	0
	2	0		2	0.3
				3	0.33

pick a passed test f_1 from the initial t-way test set such that the number of components that differ between f_1 and the core member is minimized.

On the other hand, the execution trace of a derived member is likely to be very similar to the execution trace of the core member, as these two tests only differ in one value. Therefore the faulty statement is very likely to be one of the statements that appear in the execution trace of f but do not appear in the execution trace of $f_1, f_2 \dots f_t$.

Example: Consider a system P with four parameters a, b, c, and d, where a and b takes two values 0 and 1, c takes three values 0, 1, and 2, and d takes four values 0, 1, 2, and 3. Assume that BEN ranked combination $(a \leftarrow 0, c \leftarrow 0)$ as the most suspicious combination, and reported the suspiciousness of each component as shown in Table 3-2. In our approach, the core member, f = $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$, is generated. This core member contains $(a \leftarrow 0, c \leftarrow 0)$, components $b \leftarrow 0$ and $d \leftarrow 1$ have the minimum suspiciousness value.

Two derived members f_1= $(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$ and f_2= $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1)$ are generated from the core member f. Derived member f_1 replaces $a \leftarrow 0$ in f with $a \leftarrow 1$, as $\rho(a \leftarrow 1) = 0$, and f_2 replaces $c \leftarrow 0$ in f with $c \leftarrow 1$, as $\rho(c \leftarrow 1) = 0$. Note that for f_2, we could also replace $c \leftarrow 0$ in f with $c \leftarrow 2$, as $\rho(c \leftarrow 2) = \rho(c \leftarrow 1) = 0$. In this case, a random choice is made.

3.4.2 Rank generation

In this section, we produce a ranking of statements in terms of their likelihood to be faulty by analyzing the spectrums of the core member and derived members. The suspiciousness of statement s is computed by the following formula:

$$\rho(s) = \sum_{f_i \in g_d} \rho(s, f_i)/(|g_d|) \qquad (3)$$

In the above formula, g_d represents all the derived members, and f_i is a derived member in the group, $\rho(s)$ is the average of $\rho(s, f_i)$, for all the derived members f_i. The value of $\rho(s, f_i)$ is computed by the following formula:

$$\rho(s, f_i) = \begin{cases} 1 & \text{if } \gamma(s, f) = \text{ true and } \gamma(s, f_i) = \text{ false} \\ 0.5 & \text{if } \gamma(s, f) = \gamma(s, f_i) = \text{ true} \\ 0 & \text{if } \gamma(s, f) = \text{ false} \end{cases} \qquad (4)$$

The idea behind this formula is the following. Statements that are only executed by the core member f are most suspicious and are given 1 as their suspiciousness value. Statements that are executed by both core and derived members are less suspicious, and are given 0.5 as their suspiciousness value. Note that the execution of a faulty statement by a test does not necessarily make the test fail. For example, if there exists a fault in a conditional expression, this fault can be executed by all tests but only cause some to fail. Finally statements that are not executed by f are not suspicious.

For example if there are three tests in the test group f, f_1 and f_2, where f is the core member, f_1 and f_2 are the derived members. Assume that a statement s is executed by f and f_2. The suspiciousness value of s for each derived member is: $\rho(s, f_1) = 1$ and $\rho(s, f_2) = 0.5$. And, the overall suspiciousness value for statement s is:

$$\rho(s) = \frac{1 + 0.5}{2} = 0.75$$

The higher the suspiciousness value of a statement, the more likely this statement is faulty. We rank statements by a non-ascending order of their suspiciousness value. To locate the faulty statement, statements in the top rank are examined first, and then statements in the next rank, until the faulty statement is found.

3.4.3 Discussion

The effectiveness of our approach depends to some extent on the quality of the top-ranked suspicious combination identified by BEN. If the top-ranked combination is truly inducing, the core member generated by our approach, i.e., the one that contains this combination and minimizes the suspiciousness of its environment, must fail. If the top-ranked combination is not truly inducing, but has a high probability to be inducing, the core member generated by our approach still has a high probability to fail. If the core member generated by our approach does not fail, we have to pick from the initial test set a failed test as the core member. This failed test contains this top-ranked combination, but may not minimize the suspiciousness of its environment. This may reduce the probability for the derived members to pass.

After finding the core member, the derived members are generated. The derived members are passed tests which have a similar trace to the core member. If a derived member fails, we discard it. If all the derived members fail, we pick a passed test from the initial test set that is as similar to the core member as possible. In this case, the difference between the core member and this derived member may not be minimized, which might affect the effectiveness of our approach. We believe the chance for this case, i.e., all the derived members fail, to occur is small, which is consistent with our experiments in which no such case occurred to the total of 102 versions of our subject programs.

3.4.4 Complexity analysis

Let k be the number of parameters, t the strength of the initial test set and n the number of statements in the subject program. To generate the core member, it is necessary to find the component with the minimum suspiciousness value for all the parameters that are not involved in the inducing combination (so that the suspiciousness of the environment is minimized). This takes $(k - t) * O(d)$ where d is the largest domain size.

To generate a derived member, we replace a component in the inducing combination with a component with the minimum suspiciousness value (of the same parameter). This takes $t * O(d)$ for all the derived members.

In the rank generation part, the complexity of assigning a suspiciousness value to each statement with respect to the t derived members is $O(t)$. So for all the n statements of the program, it takes $n * O(t)$. Then the rank generation part needs to sort all the statements in a non-ascending order of their suspiciousness value, which is $O(n * \log(n))$. Since t is typically much smaller than n, this sorting operation dominates the complexity of this part.

3.5 EXPERIMENT

In our experiment, we applied our approach to the Siemens suite and the *grep* program in SIR [18]. The Siemens suite has been used to evaluate several fault localization techniques [9, 16, 20]. The *grep* program is a significantly larger program than the Siemens programs and is designed to obtain some initial evidence on how our approach works on larger programs.

3.5.1 The Siemens suite

The Siemens suite contains 7 programs and each of them contains a number of faulty versions. The Siemens suite also provides an error-free version and a test set for each program. Table 3-3 represents properties of subject programs. To show that our approach works effectively when the program under test has more than one fault, one faulty version is created for each program with all faults available in the Siemens suite.

Since some faults may conflict with each other, combining all of them in one version is not possible. For example *tcas* has 41 faulty versions, but we could only apply 36 of them in one version. The column #of compatible faults in Table 3-3 shows the total number of faults that can be combined in the multiple-fault version of each program.

3.5.2 *Initial test set*

The input model of each program is shown in Table 3-4. The detailed model is also available for review in [8]. The model column in the table shows the number of parameters and their domain size. We represent it by $(d_1^{k_1} \times d_2^{k_2} \times \dots)$, where $d_i^{k_i}$ indicates that there are k_i number of parameters with domain size as d_i. Note that $k_1 + k_2 + \cdots = k$, which is the total number of parameters. For example *totinfo* has six parameters, among which three parameters have a domain size of 3, two parameters have a domain size of 5, and one parameter has a domain size of 6.

The constraint column shows the number of constraints in each model. Consider the input model of the *printtokens* program which contains different positions for different tokens. For example, *keyword* and *identifier* are two types of tokens that could appear at

Table 3-3. Characteristics of subject programs

Programs	LOC	# of faulty versions	# of compatible faults
pinttokens	472	7	7
printtokens2	399	10	9
replace	512	32	24
schedule	292	9	8
schedule2	301	10	9
tcas	141	41	36
totinfo	440	23	20

the beginning, middle or end of the input stream. A constraint is needed to prevent having more than one type of token at the same position.

Note that programs *printtokens* and *printtokens2* share the same model, and so do programs *schedule* and *schedule2*. The model in *tcas* is the same as [11]. Also note that the models are built based on the specification of the programs, i.e., independent from their implementations.

We used the ACTS tool [2] to generate t-way test sets. For each program, we first test it with a 2-way test set. (We assume that boundary testing is done before combinatorial testing is applied. Combinatorial testing is mainly used to test interaction faults involving more than one parameter.) If a program is not killed by a 2-way test set, we increase the test strength and then test the program with a 3-way test set. This process is repeated until we reach strength 4.

Table 3-5, shows the number of versions in each fault category and the number of versions that are killed by our test set. For example, in two versions of *printtokens* program the fault is missing code, and both of them are killed by our combinatorial test set. The maximum strength is used for testing is 4.

Table 3-4. Programs model

Programs	Model	#Constraints
printtokens	$(2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2)$	8
printtokens2	$(2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2)$	8
replace	$(2^4 \times 4^{16})$	36
schedule	$(2^1 \times 3^8 \times 8^2)$	0
schedule2	$(2^1 \times 3^8 \times 8^2)$	0
tcas	$(2^7 \times 3^2 \times 4^1 \times 10^2)$	0
totinfo	$(3^3 \times 5^2 \times 6^1)$	0

Table 3-5. Test results

Program	Extra code		Missing code		Incorrect code		Definition		All	
	Total	*Kill*	*Total*	*Kill*	*Total*	*Kill*	*Total*	*Kill*	*Total*	*Kill*
printtokens	1	0	2	2	2	0	2	1	7	3
printtokens2	0	0	3	2	7	7	0	0	10	9
replace	1	1	2	2	28	28	1	1	32	32
schedule	1	1	2	1	6	5	0	0	9	7
schedule2	1	0	6	2	3	1	0	0	10	3
tcas	0	0	0	0	31	30	10	6	41	36
totinfo	0	0	1	1	18	8	4	3	23	12

3.5.3 Trace collection

We used Gcov to collect execution trace. Gcov reports the number of times statements are executed by a given test. A statement is included in the execution trace of a given test if and only if it is executed by the test one or more times.

Gcov distinguishes between statements which are executable but do not execute and statements which are not executable. We used this information to compute the percentage of executable code that must be inspected to find the faulty statement. If a program crashes, Gcov does not report any coverage. To deal with this problem, we add a statement to call function *gcov_flush* before every statement. Note that this is only done after a program crashes.

3.5.4 Metrics

Recall that the output of our approach is a ranking of statements in terms of their likelihood to be faulty. In order to find the faulty statement, we inspect statements in the first rank, and then statements in the second rank, and continue to do so until we find the faulty statement. We record the number of statements that must be inspected to find the faulty statement in each program to measure the effectiveness of our approach.

The efficiency of the approach is measured by the number of tests that are executed. We show the number of tests executed in different stages of our approach, i.e., number of tests needed to kill a program, number of tests needed to identify inducing combinations, and number of tests needed to produce the ranking of faulty statements.

3.5.5 Results on single-fault versions

Table 3-6 shows the results of our approach for each program. We will not explain the column headers one by one, as they are self-explanatory. The experiments start by a 2-way test set and if needed the strength of the test set is increased up to 4. Note that in the last five columns average is used, since the data could be different in different versions. In some versions that are killed by a t-way test set, BEN does not find any t-way inducing combination, i.e. the strength of the fault is more than t. In these cases we used the top-ranked suspicious combination in the initial test set to locate the fault. It is very likely that the core member does not fail in these versions and we must select the core member from the test set. We do not increase the strength of the test set in these version. This is to show that our approach works even truly inducing combination is not found. For example, the fault in all versions of the *tcas* program has strength more than six, so the top-ranked combination identified by BEN, is not truly inducing. In all versions of *tcas*, the test that contains top-ranked combination and minimum environment passed. And, the core test is selected from the test set, therefore the column six is equal to 1 for *tcas* program.

Also, in some versions, e.g. 12 versions of *replace*, BEN could not find any t-way suspicious combination. In these cases, BEN reports all failed tests in the test set as suspicious combinations. We select one of the failed tests randomly and assume that whole test is inducing combination, i.e. all parameters (in this case 20 parameters) are involved in the inducing combination.

Table 3-6. Experimental results for single fault versions

Program	Test strength (t)	# of tests in t-way test set	# of killed versions	Avg # of tests for identifying inducing combination	Avg # of times the core member does not fail	Avg # of tests for ranking suspicious statements	Avg #of statements inspected to find faults	Avg percentage of statement inspected to locate faults
printtokens	2	170	3	33	0.67	2.33	28.67	15.24
printtokens2	2	170	9	35.88	0.55	2.44	24.44	12.53
replace	2	192	32	12.66	0	3.62	35.25	14.57
schedule	2	64	7	23.86	0.28	3	9.14	5.94
schedule2	2	64	3	42	0	3	55	43.30
tcas	2	100	17	1.70	1	2.47	15.58	22.53
	3	405	12	5.67	1	3.5	12.08	18.58
	4	1434	7	13.14	1	4.14	11.71	18.02
totinfo	2	30	5	9.4	1	2	17.6	14.30
	3	156	7	7.14	0.86	3.57	11.43	9.29

By definition, a group contains one core member and t derived members, so t+1tests are in the group. But it is likely that some derived members fail and are discarded from the group. So the maximum number of tests in the group is t+1 and the minimum is 2 (a core test and one derived test). The sixth column shows the average number of tests in the groups. Note that in the *replace* program the average number of tests in the groups is 3.62, more than t+1. This is because in 12 out of 32 versions the suspicious combination is more than 2-way. But in these versions 15 derived tests failed and their groups have 6 members.

The last two columns show the average number and the percentage of code that must be inspected to locate the fault. To compute this number, we started from statements that were ranked at the top and for statements ranked at the same rank, we started from the first statement as output by our approach. We did not perform any dependency analysis. As discussed later, dependency analysis could further reduce the percentage of code that needs to be inspected. As shown in the table, in 7 versions of *schedule* less than 6% of the code must be inspected to locate the fault and only about 28 tests are generated after testing. In the worst case 43% of the code in *schedule2* must be inspected.

Another point is that, the number of executable code in *tcas* is 65, less than 100. In this program, when only one statement is needed to inspect, it is 1.54% of executable code. So for *tcas* program number of statements gives better insight than the percentage of code.

3.5.6 Results on multiple-fault versions

In this section we show the behavior of our approach when the program under test has multiple faults. The faulty version is created such that it includes all compatible faults. The result is summarized in Table 3-7 columns are the same as Table 3-6. The *replace* program is removed from this table since all the tests in the initial 2-way test set fail; this suggests that the fault should be fairly easy to locate even without help of any advanced method.

Table 3-7. Experimental results for multiple faults versions

Program	Test strength (t)	# of tests in t-way test set	# of tests for identifying inducing combination	# of times the core member does not fail	# of tests for ranking suspicious statements	#of statements inspected to find faults	percentage of statement inspected to locate faults
Printtokens	2	170	45	0	2	1	0.53
printtokens2	2	170	110	0	2	1	0.51
schedule	2	64	140	0	3	5	3.24
schedule2	2	64	42	0	3	1	0.79
tcas	2	100	71	0	3	1	1.54
totinfo	2	30	28	1	3	3	2.44

3.5.7 The grep program

We applied our approach to the *grep* program from SIR [18], which has 10068 lines of code. The *grep* program has two input parameters, patterns and files. It prints lines in the files that contain a match of any of the patterns. While the *grep* program can take multiple patterns and files, we only used a single pattern and file in this experiment. Also different options can be used to control the behavior of the *grep* program. For example, option "–w" causes the program to print only lines containing whole-word matches.

The *grep* program can take four different types of patterns: (1) *basic-regexp*: a basic regular expression. (2) *extended-regexp*: an extended regular expression. (3) *fixed-strings*: a list of fixed strings. (4) *perl-regexp*: a Perl regular expression. In this experiment, we focused on *extended-regexp*. There are five versions of *grep* in the benchmark, and each of them has a number of seeded faults. We selected the first version, which has 18 seeded faults. Thus 18 faulty versions were built, each of which contains only one fault.

The *grep* program was written in C, and has ten header files and one C file. The benchmark does not provide any specification for the *grep* program. So we used the *grep*'s manual from [22] as the program specification. We modeled the input space focusing on extended regular expression. The input model can be represented using the exponential notation as $(2^7 \times 4^1 \times 5^1 \times 6^3 \times 8^1 \times 9^1 \times 13^1)$ and has one constraint related to the repetition operator. The 2-way test set created from the model has 121 tests and killed 4 versions, 3, 8, 11 and 14. We executed all the tests that come with this program in SIR, which also only killed these 4 versions.

For version 11, all tests failed. In this case, our approach cannot be applied. In practice, the fact that all the test cases failed suggests that this fault can probably be found easily even without help from tools like ours.

For version 3, BEN generated 40 tests and identified 826 2-way suspicious combinations. We generated a group of tests from the most suspicious combination ranked by BEN. One derived member in the group failed. Thus, the group had 2 tests, one core member and one derived member. The statements were ranked and the faulty statement was in the second rank. To locate the faulty statement, 19% of the executable code needs to be inspected.

For version 8, BEN generated 7 tests and identified one 2-way suspicious combination. A group of 3 tests was generated, and the faulty statement was ranked in the second rank. To locate the faulty statement, 0.9% of the executable code needs to be inspected.

For version 14, BEN generated 41 tests and identified two 2-way suspicious combinations. A group of 3 tests was generated and the faulty statement was in the second rank. To locate the faulty statement, 8.5% of the executable code needs to be inspected.

On the average BEN identified 2-way inducing combination by generating 13.67 tests for the three killed versions, and the faulty statement as located by inspecting 9% of the executable code. Recall from sub-section Metrics that we did not perform any manual analysis when we determine the percentage of code that needs to be inspected. This percentage can be significantly reduced even with some simple dependency analysis, which we believe is what people typically do in practice.

3.5.8 Threats to validity

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. One of the key steps in our experiments is modeling the input parameters, that may affect the correctness of the result. To reduce this threat, we have done this step by using the program specifications and error-free versions, without having any knowledge about the faults. Further, consistency of the results has been carefully checked to detect potential mistakes made in the experiments.

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from the Siemens suite [18]; these programs are created by a third party and have been used in other studies [9, 16, 20]. But the subject programs are programs of relatively small size with seeded faults. To mitigate this threat, the *grep* program was added to the experiments, but more experiments on larger programs with real faults can further reduce this threat.

Each of the Siemens program has multiple versions, each of which has a single fault. However, programs in practice could have multiple faults. To mitigate this threat, we created a version that combined all the compatible faults and conducted an experiment on this version. More experiments on programs with real faults can further reduce this threat.

3.6 RELATED WORK

Our approach is a spectrum-based approach based on combinatorial testing. In this section, we first discuss two areas of work: (1) fault localization based on combinatorial testing; and (2) spectrum-based fault localization.

Fault localization based on combinatorial testing: Several recent efforts have been reported aiming to develop fault localization techniques that utilize the result of combinatorial testing. Two techniques, called FIC and FIC_BS [21], take as input a single failed test from a combinatorial test set, and identify as output a minimal inducing combination that causes the test to fail. The main idea of the two techniques consists of changing, in a systematic manner, the parameter values in the failed test. A parameter value is considered to be involved in an inducing combination if changing it to a different value causes the failed test to pass. It is assumed that changing a parameter value does not introduce any new inducing combination.

Other techniques have also been reported that take as input the results of an entire combinatorial test set, i.e., not only a single failed test, and identify as output one or more combinations that are likely to be inducing. These techniques include AIFL and InterAIFL [15, 17], and our earlier work BEN [6]. The key idea behind these techniques is that an inducing combination is more likely to appear in a failed test than in a passed test. InterAIFL and BEN may generate and execute additional tests to refine the results.

To the best of our knowledge, all the existing work in this area has focused on the identification of inducing combinations. Our work presented in this paper is the first effort to leverage the notion of inducing combination to locate the faults inside the source code. In this respect, our work is the natural, next step of the above existing work.

Spectrum-Based Fault Localization: Tarantula uses the coverage of statements in the execution traces of failed and passed tests to compute suspiciousness of each statement [10]. The suspiciousness score of each statement is the ratio of failed test cases that execute the statement divided by the ratio of failed test cases that execute the statement plus the ratio of passed test cases that execute the statement. The statements with the highest suspiciousness score must be examined first when looking for the fault. If the fault is not found, the remaining statements are examined in a non-increasing order of their suspiciousness scores. Other approaches such as Pinpoint, AMPLE and Ochiai [1] are reported that adopt the general framework of Tarantula but use different metrics to compute suspiciousness of statements. Experiments reported by [13], shows that no other spectrum-based approaches statistically significantly outperform Tarantula.

Renieris and Reiss in [16] proposed three different spectrum-based approaches, set union, set intersection and nearest neighbor. They assume the existence of one failed run and a large number of passed runs. The input of the approach is a group of program runs, instead of a test set. The set union method computes $f - \bigcup_S s$, where f is a program spectra of a failing run and $\bigcup_S s$ is the union spectra of a set of passed runs. The intersection model computes, the intersection spectra, $\bigcap_S s - f$, of a set of passed runs. The statements in these spectra are then checked to find the actual faults.

The nearest neighbor method [16] chooses one passed run whose spectrum is the closest one to the failed spectrum. Then it searches for a fault in the difference set of these two spectra. If the fault is not found in the difference set, a ranking technique based on the program dependence graph is proposed. In the program dependence graph, the nodes corresponding to the difference set are identified and called as blamed nodes. A breadth-first search from the blamed nodes is performed along dependency edges in both forward and backward directions. All adjacent nodes to the blamed nodes are grouped in the next rank and checked as blamed. This process is repeated until the faulty statement is found.

Our approach is fundamentally different from the existing spectrum-based methods in the following aspects. First, the existing spectrum-based methods do not deal with the problem of test generation. Instead, they assume the existence of a large number of test runs and then analyze the spectra with respect to these test runs. In contrast, our approach proposes a systematic strategy to generate a group of tests, whose spectra are then used to produce a ranking of statements. Second, the existing spectrum-based methods do not make a clear distinction between the testing stage and the fault localization stage. In these methods, all the tests are generated up front and then executed and traced to record the program spectra. In contrast, our approach is to be applied after combinatorial testing is performed. The tests used in our approach for the purpose of fault localization are generated based on the result of combinatorial testing. Furthermore, only tests that are generated by our approach are traced to record program spectra, whereas the original combinatorial tests are not traced. Finally, we point out that the formulas used in our approach to compute suspiciousness of statement are different from those used in the existing spectrum-based methods. This is to accommodate the fact that our approach uses a small group of tests among which there is only a single failed test, i.e., the core member.

The effectiveness of a spectrum-based method such as ours also depends on the quality of the initial set of failed and passed tests. Baudrey et al. in [5] proposed a criterion to evaluate the efficiency of a test set for fault localization. They introduced a concept, i.e., dynamic basic block that contains a set of statements that is covered by the same test cases in a test set. All statements in the same basic block typically have the same rank. The more dynamic basic blocks in the program a test set could distinguish, the more efficient the test set is for fault localization. They then use an adoption of a genetic algorithm to optimize a test set and maximize the number of dynamic basic blocks.

3.7 CONCLUSION

In this paper, we presented an approach to localizing faults that leverages the result of combinatorial testing. The key idea of our approach is that we systematically generate a group of tests from an inducing combination such that the spectra of these tests can be analyzed quickly to identify the faulty statement. This group of tests consists of a core member that is a failed test run and a number of derived members that are passed test runs but are very similar to the core member. The suspiciousness values of statements are computed by analyzing the spectra of the core member and the derived members. We applied our approach to the Siemens suite and also the *grep* program. Our experimental results show that our approach requires a very small number of tests while significantly reducing the number of statements to be inspected for fault localization.

We plan to conduct more empirical studies to further evaluate the performance of our approach. In particular, we plan to apply our approach to more programs like *grep* that are larger and/or more complex than the Siemens programs. Our current approach assumes that a combinatorial test set is used to test a program. We plan to investigate how to adapt our approach to work with an arbitrary test set. That is, we will try to identify inducing combinations from an arbitrary test and then use them to generate tests for fault localization. This will further increase the applicability of our approach.

3.8 Acknowledgment

This work is supported by two grants (70NANB12H175 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

3.9 REFERENCES

1. R. Abreu, P. Zoeteweij and A. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, 39-46, 2006.
2. Advanced Combinatorial Testing System (ACTS), http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html, 2013.
3. H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. Proceedings of Software-Practice & Experience. 23(6):589-616, 1993.
4. H. Agrawal, J. Horgan, S. London, and E. Wong. Fault localization using execution slices and dataflow tests. Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, 143-151, 1995.
5. B. Baudry, F. Fleurey, and Y. Traon. Improving test suites for efficient fault localization. Proceedings of the 28th ACM International Conference on Software Engineering (ICSE). 82-91, 2006.

6. L.S.G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. Identifying Failure-Inducing Combinations in a Combinatorial Test Set. Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), 370-379, 2012.

7. M. Grindal, J. Offutt, and S. Andler. Combination Testing Strategies: A Survey. Journal of Software Testing, Verification and Reliability, 15(3), 167-199, 2005.

8. Input models for the subject programs, http://barbie.uta.edu/~laleh/BEN/models.html. 2013.

9. J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE). 273-282, 2005.

10. J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. Proceedings of the 24th ACM International Conference on Software Engineering (ICSE), 467-477, 2002.

11. D.R. Kuhn and V. Okum. Pseudo-Exhaustive Testing for Software. Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW). 153-158, 2006.

12. D.R. Kuhn, D.R. Wallace, and A.M. Gallo. Software Fault Interactions and Implications for Software Testing. IEEE Transaction on Software Engineering 30(6):418-421, 2004.

13. Lucia, D. Lo, L. Jiang, A. Budi, Comprehensive evaluation of association measures for fault localization. Proceedings of the IEEE International Conference on Software Maintenance, 1-10, 2010.

14. C. Nie and H. Leung. A survey of combinatorial testing. ACM Computing Surveys (CSUR).43(2):11, 2011.

15. C. Nie and H. Leung. The Minimal Failure-Causing Schema of Combinatorial Testing. ACM Transactions on Software Engineering and Methodology (TOSEM), 20(4):15 , 2011.

16. M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. Proceedings of the i8th IEEE International Conference on Automated Software Engineering (ASE). 30-39, 2003.

17. L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. Proceedings of the 5th International Conference on Computational Science (ICCS), 1088-1091, 2005.

18. 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.

19. Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. Proceedings of the 10th International Conference on Quality Software (QSIC), 495–502, 2010.

20. E. Wong and V. Debroy. A survey on software fault localization. Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, 2009.

21. Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA), 331-341, 2011.

22. GNU Grep 2.14 http://www.gnu.org/software/grep/manual/grep.html, 2012.

Chapter 4. A Combinatorial Testing-Based Approach to Fault Localization

This chapter contains a paper which is submitted in January 2016, and currently under review.

A Combinatorial Testing-Based Approach to Fault Localization

Laleh Sh. Ghandehari, Yu Lei, Raghu Kacker, Richard Kuhn

Abstract— Combinatorial testing has been shown to be a very effective strategy for software testing. After a failure is detected, the next task is to identify one or more faulty statements in the source code that have caused the failure. In this paper, we present a fault localization approach, called BEN, which produces a ranking of statements in terms of their likelihood of being faulty by leveraging the result of combinatorial testing.

BEN consists of two major phases. In the first phase, BEN identifies a combination that is very likely to be failure-inducing. A combination is failure-inducing if it causes any test in which it appears to fail. In the second phase, BEN takes as input a failure-inducing combination identified in the first phase and produces a ranking of statements in terms of their likelihood to be faulty. We conducted an experiment in which our approach was applied to the Siemens suite and two relatively large programs, grep and gzip, from Software Infrastructure Repository (SIR). The experimental results show that our approach can effectively and efficiently localize the faulty statements in these programs.

Index Terms— Combinatorial Testing, Fault Localization, Debugging

4.1 INTRODUCTION

Combinatorial testing is based on the observation that a large number of software failures are caused by interactions of only a few input parameters [20]. A t-way

• *Laleh Sh. Ghandehari and Yu Lei are with the Department of Computer Science and Engineering, University of Texas at Arlington, Texas, USA. E-mail: laleh.shikhgholamhosseing@mavs.uta.edu and ylei@uta.edu.*
• *Raghu Kacker, and Richard Kuhn are with the Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, Maryland, USA. E-mail: raghu.kacker@nist.gov and kuhn@nist.gov.*

combinatorial test set, or simply a t-way test set, is designed to cover all the t-way combinations, i.e., combinations involving any t parameters [5][6][21]. Typically, t is a small number and is referred to as the strength of a combinatorial test set [19][20]. When the input parameters are properly modeled, a t-way test set triggers all or most failures caused by interaction of at most t parameters. Empirical studies have shown that combinatorial testing is very effective in practice [4][11][19].

After a failure is detected during combinatorial testing, the next task is locating the fault that caused the failure. In this paper, we present a fault localization approach called BEN that leverages the result of combinatorial testing. BEN takes as input a combinatorial test set and the execution status, i.e., pass or fail, of each test, and produces as output a ranking of statements in terms of their likelihood to be faulty.

Most research in combinatorial testing has focused on developing efficient combinatorial test generation algorithms [5][21][25], or demonstrating the effectiveness of combinatorial testing in different application domains [4][10][29][32]. Several approaches have been developed to identify failure-inducing combinations in a combinatorial test set [37][33]. A failure-inducing combination, or simply an inducing combination, is a combination that causes all tests containing this combination to fail [37][26]. These approaches, however, are not designed to locate faulty statements in the source code.

A significant amount of research has been reported on spectrum-based approaches to fault localization [1][17][27][34]. A program spectrum records information about certain aspects of a test execution [34], such as function call counts, program paths, program slices and use-def chains [27]. Examples of spectrum-based methods include Tarantula [18], set union, set intersection, and nearest neighbor [27]. These approaches identify faulty statements by analyzing the spectra of passed and failed test executions [18][27][23]. These approaches are not designed to work with combinatorial testing.

However, they can be applied to analyze test executions obtained from combinatorial testing, provided that the test executions were traced. In case that a combinatorial test set is already executed without being traced, which is often the case in practice considering that testing and debugging are fundamentally different activities and are often performed separately, the test set must be re-executed before these approaches could be applied. In contrast, our approach does not require every test execution to be traced and is designed to be applied after normal testing is performed where test executions are not traced. We will compare our approach, i.e., BEN, to these approaches both analytically (Section 4.6.2) and experimentally (Section 4.5.2.3).

Our approach consists of two major phases, consisting of *inducing combination identification* and *faulty statement localization*. In the first phase, BEN takes as input a t-way combinatorial test set, and it adopts an iterative framework to identify an inducing combination of size t or larger. At each iteration, a set F of tests is analyzed. Initially F is the t-way combinatorial test set taken as input by BEN. BEN first identifies the set π of all t-way suspicious combinations in F, and ranks them based on their likelihood to be inducing. Suspicious combinations are candidates of inducing combinations.

Next, our approach generates a set F' of new test. If all the tests containing a suspicious combination c in F' fail, c is marked as an inducing combination, and the process stops. Otherwise, all the tests in F' are added to F and the process is repeated until a t-way combination is marked as an inducing combination or a stopping condition is satisfied. In the latter case, no t-way inducing combination is identified and we increase the size of inducing combination. That is, we try to identify a (t+1)-way inducing combination. This process is repeated until an inducing combination is found. Note that this process must terminate, as a failed test is by definition an inducing combination.

The novelty of our approach in this phase lies in the fact that we rank suspicious combinations based on two notions, including suspiciousness of a combination and suspiciousness of the environment of a combination. Informally, the environment of a combination consists of other parameter values that appear in the same test case. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. Moreover, new tests are generated for the most suspicious combinations. Let f be a new test generated for a suspicious combination c. Test f is generated such that it contains c and the suspiciousness of the environment for c is minimized. If f fails, it is more likely to be caused by c instead of other values in f.

In the second phase of our approach, i.e., faulty statement localization, BEN systematically generates a small group of tests from an inducing combination such that the execution traces of these tests can be analyzed to quickly locate the faults. One of the tests in the group is referred to as the core member, which consists of the inducing combination and produces a failed test execution. The other tests in the group are referred to as the derived members, which are derived from the core member in a way such that they are likely to execute a trace that is very similar to the trace of the core member but produce a different outcome, i.e., a passed execution. The spectrum of the core member is then compared to the spectrum of each derived member to produce a ranking of statements in terms of their likelihood to be faulty.

Our approach differs from existing spectrum-based approaches, which do not deal with the problem of test generation. Instead, they assume the existence of a large number of tests, which are generated randomly and/or using other techniques [18][27][34].

The second phase of BEN is inspired by the notion of nearest neighbor [27]. The key idea of nearest neighbor is that faulty statements are likely to appear in the execution trace of a failed test but not in the execution trace of a passed test that is as similar to this failed test as possible. If two tests are significantly different, they are likely to represent different application scenarios. Thus, the differences in the execution traces of these two tests are likely due to program logic, instead of faults. The novelty of our approach lies in the fact that we generate, in a systematic manner, a failed test, i.e., the core member, and then derive its nearest neighbors from this failed test, i.e., the derived members. This is in contrast with the approach in [27], which executes a large number of tests from which a failed test and its nearest neighbors are selected.

We report an experiment in which we applied our approach to the Siemens suite and two relatively large programs, *grep* and *gzip*, in the Software Infrastructure Repository (SIR) [31]. The Siemens suite has been used in several studies to evaluate fault localization methods [17] [27] [34]. It contains seven relatively small programs, each of which has a number of faulty versions. The two larger programs, i.e., *grep* and *gzip*, have 10068 and 5680 lines of code respectively, and they also have a number of faulty versions in SIR. The faulty versions in SIR contain a single fault. In order to evaluate the performance of BEN with multiple faults, we created several faulty versions that contain multiple faults.

The results show that our approach is effective in localizing faulty statements and also efficient in that only a small number of tests need to be executed and instrumented. For example, one of the implementations of the *grep* program called *grep3* has 18 faulty versions. Among them four versions were killed by a 2-way test set consisting of 121 tests. On average, BEN generated and executed 25 additional tests and instrumented 7 tests for theses 4 versions. One needs to examine 0.64% (on average) of the code to locate the faulty statement.

Moreover, we compared the results of BEN and two other spectrum based approaches, Tarantula [18] and Ochiai [23]. Since Tarantula and Ochiai do not deal with test generation, they were applied to the initial combinatorial test set. Our experimental results show that BEN performed better than or as well as Tarantula and Ochiai for all the programs, but it requires a significant smaller number of test executions to be traced and analyzed. In particular, BEN works better than the two other approaches, for the two larger programs and when they have multiple faults. For instance, *gzip1* has 13 multiple-fault versions, among which BEN outperforms Tarantula for nine versions with an average improvement of 13.58%. That is, in these nine versions BEN on average inspects 13.58% less lines of code than Tarantula.

The approach presented in this paper is the extension of our previous work, which has been presented in [14] and [13]. To the best of our knowledge, our work is the first to deal with code-based fault localization based on combinatorial testing. Existing work in this area, i.e., fault localization based on combinatorial testing, has mainly dealt with the problem of how to identify inducing combinations [37][30][26][33].

The remainder of this paper is organized as follows. Section 4.2 explains basic concepts and assumptions of our approach. Section 4.3 presents the details. Section 4.4 gives an example to illustrate the approach. Section 4.5 reports the experimental results of applying our approach to the subject programs. Section 4.6 discusses existing work on fault localization. Section 4.7 provides the concluding remarks plans for future work.

4.2 PRELIMINARIES

In this section, we introduce the basic concepts and assumptions needed in our approach.

4.2.1 Basic concepts

Assume that the system under test (SUT) has a set P of k input parameters, denoted by $P = \{p_1, p_2, \ldots, p_k\}$. Let d_i be the domain of parameter p_i. That is, d_i contains all possible values that $\mathrm{p_i}$ could take. Let $\mathrm{D} = \{\mathrm{d_1} \cup \mathrm{d_2} \cup \ldots \cup \mathrm{d_k}\}$. Let $\Pi = \mathrm{d_1} \times \mathrm{d_2} \times \ldots \times \mathrm{d_k}$. Let S be the set of program statements.

Definition 1. (*Test Case*) A test case or simply a test is a function that assigns a value to each parameter. Formally, a test is a function $\mathrm{f}: \mathrm{P} \to \mathrm{D}$.

Definition 2. (*Constraint*) A constraint ψ is a function that maps a test case to a Boolean value true or false, formally, $\psi: \Pi \longrightarrow \{\mathrm{true}, \mathrm{false}\}$.

The system under test, SUT, includes a set $\Psi = \{\psi_1, \psi_2, \ldots, \psi_{|\Psi|}\}$ of constraints. We use Γ to represent all valid tests for the SUT, formally $\Gamma \subseteq \Pi$. A test $\mathrm{f} \in \Gamma$ is valid if and only if $\forall \psi \in \Psi$, $\psi(\mathrm{f}) = \mathrm{true}$. To simplify the presentation, we assume that each test is a valid test in the remainder of the paper unless otherwise specified.

Definition 3. (*Test Oracle*) A test oracle determines whether the execution of a test is “pass” or “fail”. Formally, a test oracle is a function $\mathrm{r}: \Gamma \to \{\mathrm{pass}, \mathrm{fail}\}$.

Definition 4. (*Combination*) A combination c is a test f restricted to a non-empty, subset M of parameters in P. Formally, $\mathrm{c} = \mathrm{f}|_{\mathrm{M}}$ where $\mathrm{M} \subseteq \mathrm{P}$, and $|\mathrm{M}| > 0$.

In the preceding definition, M is a subset of P. Thus, a test is a combination where $\mathrm{M} = \mathrm{P}$. We use $\mathrm{dom(c)}$ to denote the domain of c, which is a set of parameters involved in c. (Note that $\mathrm{dom(c)}$ is the domain of a function, which is different from the domain of a parameter.)

A combination of size one is a special combination, which we refer to as a component. Since there is only one parameter involved, we denote a component o as an assignment, i.e., $\mathrm{o} = \mathrm{p} \leftarrow \mathrm{v}$, where $\mathrm{o(p)} = \mathrm{v}$.

Definition 5. (*Component Containment*) A component $\mathrm{o} = \mathrm{p} \leftarrow \mathrm{v}$ is contained in a combination c denoted by $\mathrm{o} \in \mathrm{c}$, if and only if $\mathrm{p} \in \mathrm{dom(c)}$ and $\mathrm{c(p)} = \mathrm{v}$.

Definition 6. (*Combination Containment*) A combination c is contained in a test f, denoted by $\mathrm{c} \subseteq \mathrm{f}$, if and only if $\forall \mathrm{p} \in \mathrm{dom(c)}, \mathrm{f(p)} = \mathrm{c(p)}$.

Definition 7. (*Inducing Combination*) A combination c is failure-inducing, or simply inducing, if any test f in which c is contained fails. Formally, $\forall \mathrm{f} \in \Gamma: \mathrm{c} \subseteq \mathrm{f} \Longrightarrow \mathrm{r(f)} = \mathrm{fail}$.

Definition 7 is consistent with the definition of inducing combinations in previous work [37][30][26][33].

Definition 8. (*Inducing Probability*) The inducing probability of a combination c is the ratio of the number of all possible failed tests containing c to the number of all possible tests containing c. The inducing probability is computed by

$$\frac{|\{\mathrm{f} \in \Gamma | \mathrm{r(f)} = \mathrm{fail} \ \wedge \mathrm{c} \subseteq \mathrm{f}\}|}{|\{\mathrm{f} \in \Gamma | \mathrm{c} \subseteq \mathrm{f}\}|}$$

The computation of inducing probabilities requires all possible tests containing a combination, which is often not possible in practice. This notion is mainly used to evaluate the goodness of our experimental results. By Definition 7, an (truly) inducing combination is a combination whose inducing probability is one.

Definition 9. (*Suspicious Combination*) A combination c is a suspicious combination in a test set $\mathrm{F} \subseteq \Gamma$ if c is contained only in failed tests in F. Formally, $\forall \mathrm{f} \in \mathrm{F}: \mathrm{c} \subseteq \mathrm{f} \Rightarrow \mathrm{r(f)} = \mathrm{fail}$.

Inducing combinations must be suspicious combinations, but suspicious combinations may or may not be inducing combinations.

Definition 10. (*Test Spectrum*) A test spectrum is a membership function γ that determines whether a statement is exercised by a test (or precisely the execution of a test). Formally, $\gamma: \mathrm{S} \times \Gamma \rightarrow \{\mathrm{true}, \mathrm{false}\}$, where $\gamma(\mathrm{s}, \mathrm{f}) = \mathrm{true}$ if $s \in S$ is executed by $\mathrm{f} \in \Gamma$, and $\gamma(\mathrm{s}, \mathrm{f}) = \mathrm{false}$ otherwise.

In the rest of the paper, we also use $\gamma(\mathrm{f})$ to represent all the statements that are executed by f. Formally, $\gamma(\mathrm{f}) = \{s \in S \mid \gamma(\mathrm{s}, \mathrm{f}) = true\}$.

4.2.2 Assumptions

In this section, we present several assumptions that must hold to apply BEN.

Assumption 1. The output of the SUT is deterministic. In other words, the SUT always produces the same output for a given test.

Assumption 2. There exists a test oracle that determines the status of a test execution, i.e., “pass” or “fail”.

Assumption 2 is made to simplify the presentation of our approach. The construction of a test oracle is an independent research problem. When a test oracle exists, our approach can be fully automated. When a test oracle does not exist, our approach can still be applied, but the user needs to assist in determining the execution status of a test case.

Assumption 3. There are at least one failed and one passed tests in the initial test set.

If there is no failed test, no fault is detected. Fault localization is typically performed when at least one fault is detected. If there is no passed test, the fault is likely easy to locate.

4.3 APPROACH

In this section, we present the BEN approach. BEN consists of two major phases, inducing combination identification and faulty statement localization. BEN assumes that a combinatorial test set has been executed on the subject program. Thus, the execution status of each test is known. Also, it assumes that the input parameter model used to generate the combinatorial test set is known. An input parameter model includes a set of parameters, each of which has a set of values, and a set of constraints that must be satisfied for a test to be valid.

The output of our approach is the ranking of statements such that the higher a statement is ranked, the more likely it is faulty. In the rest of this section, we explain the details of BEN.

4.3.1 *Phase 1: Inducing combination identification*

This phase takes three inputs, including an input parameter model Ω, a combinatorial test set F_0 created based on Ω, and the strength t of F_0. It produces as output an inducing combination, or more precisely a highly suspicious combination.

4.3.1.1 Framework

As shown in Figure 4-1 our approach adopts an iterative framework in this phase. At each iteration, the *identify* algorithm is used to analyze a set F of test cases and identify an l-way inducing combination. Initially, F is the initial combinatorial test set and l, the size of inducing combination, is the strength of the initial test set.

If the *identify* algorithm identifies an l-way inducing combination, c, the while loop stops and reports c as an inducing combination (line 5). If no l-way inducing combination is found, i.e. the *identify* algorithm returns null (line 2), l will be incremented. In the next iteration, the framework searches for inducing combination of size l+1. As shown in Figure 4-2 new tests may be added into F by the identify algorithm each time it is called.

Based on assumption 3, there is at least one failed test in the initial test set. Recall that a failed test is an inducing combination by definition. Therefore, there is at least one inducing combination in the initial test set. Thus, the framework must terminate.

```
The Framework
1   $l \leftarrow t$ and $F \leftarrow F_0$
2   while($(c \leftarrow \text{identify}(\Omega, l, F)) = \text{null}$) {
3       $l \leftarrow l + 1$
4   }
5   return c
```

Figure 4-1. The framework for identifying inducing combination

4.3.1.2 Algorithm identify

Algorithm *identify* is shown Figure 4-2, and is designed to find an l-way inducing combination in the test set F. It takes as input the input parameter model, Ω, test set F and l. Algorithm *identify* consists of two main steps. (1) Rank generation: In this step, we first identify all the l-way suspicious combinations in F (line 3). Then, the suspiciousness value of each component and suspicious combination is computed (line 6 and line 8), and finally a ranking of the suspicious combinations is produced (line 10). (2) Test generation: In this step, for a user-specified number of top-ranked suspicious combinations, a set of new tests is generated (line 14). Note that the user could specify the number of top-ranked

```
Algorithm identify
1  while ( c = null) {
2     // Step 1. rank suspicious combinations
3     $\pi \leftarrow l$-way suspicious combinations in F
4     if ($\pi$ = empty) then return null; //No l-way inducing combination is found
5     let $\Theta$ be the set of suspicious components that appear in $\pi$
6     compute the suspiciousness of each component in $\Theta$
7     for each combination $\tau \in \pi$ {
8        compute $\rho_c(\tau)$ and $\rho_e(\tau)$ based on formula 2 and 3
9     }
10    produce a ranking of $l$-way combinations in $\pi$ based on $\rho_c$ and $\rho_e$
11    // Step 2. generate new tests
12    let T be the set containing a user-specified number of top-ranked combinations
13    for every combination $\tau \in T$ {
14       generate a set $F'$of a user-specified number of new tests that contain $\tau$
15       if ($|F'| == 0$ || ($\forall f \in F', r(f) = fail$) )
16            $c \leftarrow \tau$     // l-way inducing combination is found
17       else
18             F $\leftarrow$ F $\cup F'$
19    }
20 }
21 return c
```

Figure 4-2. The Identify algorithm

suspicious combinations and the number of tests generated for each top-ranked combination. If an inducing combination is not found, all the new tests in F' are added to the test set F to refine the ranking of suspicious combinations in the next iteration (line 18).

The two steps, rank generation and test generation, are performed iteratively until one of the following two stopping conditions is satisfied:

(1) The set π of l-way suspicious combinations becomes empty (line 4); or.

(2) An l-way inducing combination is found (line 16 and line 21). An l-way suspicious combination τ is considered to be an inducing combination if no new test containing τ can be generated, or all newly generated tests containing τ fail (line 15). In the former case, it is very likely that all tests containing τ have been executed, and all of them must have failed (otherwise, τ is not suspicious). Thus, τ is the inducing combination. In the latter case, τ is likely to be inducing due to the way the new tests are generated as explained in Section 4.3.1.4. Later, we will discuss how BEN works when a non-inducing combination is reported as an inducing combination.

In the following subsections, we will explain the two major steps, rank generation and test generation.

4.3.1.3 Rank generation

In this step, we first identify the set π of all l-way suspicious combinations in F. Initially, π contains all the l-way combinations covered by F. We then check each l-way combination τ in π. If τ appears in at least one passed test, τ is removed from π, since it is not suspicious anymore. In the subsequent iterations, we do not re-compute π from the scratch. Instead, we only remove from π all the combinations contained by newly added tests that passed.

If there is no l-way suspicious combination, there is no l-way inducing combination. In this case, the *identify* algorithm returns null. The main framework, as shown in

Figure 4-1, then increases the size of inducing combination by one, and calls the *identify* algorithm again.

In the first iteration, where $\mathrm{F} = \mathrm{F}_0$ and $\mathrm{l} = \mathrm{t}$, all the t-way combinations are covered by F, as F_0 is a t-way test set. But, when $\mathrm{l} > \mathrm{t}$, F does not contain all the l-way combinations. Therefore our approach focuses on l-way combinations that appear F.

We next discuss how to rank the suspicious combinations in π. First, we introduce three important notions of suspiciousness, including *suspiciousness of component*, *suspiciousness of combination*, and *suspiciousness of environment*.

Suspiciousness of component (ρ): This notion is defined such that the higher ρ a component o has, the more likely o contributes to a failure, and the more likely o appears in an inducing combination. Let F be the test set that is analyzed in the current iteration. In our approach, ρ is computed by the following formula:

$$\rho(\mathrm{o}) = \frac{1}{3}\big(\mathrm{u}(\mathrm{o}) + \mathrm{v}(\mathrm{o}) + \mathrm{w}(\mathrm{o})\big) \qquad (1)$$

Where

$$\mathrm{u}(\mathrm{o}) = \frac{|\{\mathrm{f} \in \mathrm{F} | \mathrm{r}(\mathrm{f}) = \mathrm{fail} \ \land \mathrm{o} \in \mathrm{f}\}|}{|\{\mathrm{f} \in \mathrm{F} | \mathrm{r}(\mathrm{f}) = \mathrm{fail}\}|}$$

$$\mathrm{v}(\mathrm{o}) = \frac{|\{\mathrm{f} \in \mathrm{F} | \mathrm{r}(\mathrm{f}) = \mathrm{fail} \ \land \mathrm{o} \in \mathrm{f}\}|}{|\{\mathrm{f} \in \mathrm{F} | \mathrm{o} \in \mathrm{f}\}|}$$

$$\mathrm{w}(\mathrm{o}) = \frac{|\{\tau | \mathrm{o} \in \tau \ \land \tau \in \pi\}|}{|\pi|}$$

The first factor, $\mathrm{u}(\mathrm{o})$, shows the ratio of the number of failed test cases in which component o appears over the total number of failed test cases. The second factor, $\mathrm{v}(\mathrm{o})$, shows the ratio of the number of failed test cases in which component o appears over the total number of test cases in which component o appears. The third factor shows the ratio of the number of suspicious combinations in which component o appears over the total number of suspicious combinations. The three factors are averaged to produce a value between 0 and 1.

The motivation behind the first two factors is that the more frequently a component appears in failed test cases, this component is more likely to contribute to a failure.

There is an important difference between the first two factors. Since the greater the domain size is, the less frequently each individual value of this parameter appears in a test set and consequently in failed test cases, the first factor, $\mathrm{u(o)}$, has a bias towards smaller domain size parameters. The second factor, $\mathrm{v(o)}$, is used in to reduce this bias.

The motivation for the third factor is that components of inducing combinations tend to appear more frequently in suspicious combinations. For example, assume that combination $\mathrm{c} = (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0)$ is inducing. Let $\mathrm{f} = (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0, \mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 0)$ be a test case. Test case f fails as it contains c. Let $\mathrm{f}' = (\mathrm{a} \leftarrow 1, \mathrm{b} \leftarrow 1, \mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 0)$ be another test case, which passes since it does not contain c. The set of suspicious combinations derived from these two test cases is

$$\pi = \{(\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0), (\mathrm{a} \leftarrow 0, \mathrm{c} \leftarrow 0), (\mathrm{a} \leftarrow 0, \mathrm{d} \leftarrow 0), (\mathrm{b} \leftarrow 0, \mathrm{c} \leftarrow 0), (\mathrm{b} \leftarrow 0, \mathrm{d} \leftarrow 0)\}$$

In this set, the frequencies of $\mathrm{a} \leftarrow 0$ and $\mathrm{b} \leftarrow 0$ both are greater than others. The reason is that $(\mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 0)$ appears in f', which is a passed test case.

Suspiciousness of combination (ρ_c): Suspiciousness of a combination τ is defined to be the average of suspiciousness of components that appear in τ. Formally suspiciousness of combination τ, $\rho_c(\tau)$ is computed by

$$\rho_c(\tau) = \frac{1}{|\tau|} \sum_{\forall\, \mathrm{o} \in \tau} \rho(\mathrm{o}) \qquad (2)$$

Suspiciousness of Environment (ρ_e): The environment of a combination τ in a test f includes all the components that appear in f but do not appear in τ. The suspiciousness of the environment of a combination τ in a test f is the average suspiciousness of the components in the environment of τ. If there is more than one (failed) test containing τ in a test set, the suspiciousness of the environment of τ in this test set is the minimum suspiciousness of environment of τ in all the tests containing τ. Formally, suspiciousness of the environment ρ_e is computed by

$$\rho_e(\tau) = \min_{\substack{f \in F \wedge \tau \subseteq f \\ \wedge\, r(f)=\text{fail}}} \sum_{o \in f \wedge o \notin \tau} \rho(o) \qquad (3)$$

Now we discuss how to actually rank the suspicious combinations based on ρ_c and ρ_e. Intuitively, the higher the value of ρc, the lower the value of ρe, the higher a combination is ranked.

To produce the final ranking, we first produce two rankings R_c and R_e of suspicious combinations, where R_c is in the non-ascending order of ρc and R_e is in the non-descending order of ρe. The final ranking R is produced by combining R_c and R_e as follows. Let τ and τ' be two suspicious combinations. Assume that τ has ranks r_c and r_e in R_c and R_e, respectively, and τ' has ranks r'_c and r'_e in R_c and R_e, respectively. In the final ranking R, τ is ranked before τ' if and only if $r_c + r_e < r'_c + r'_e$.

4.3.1.4 Test generation

This step is responsible for generating new test cases for user-specified top-ranked suspicious combinations. Let τ be a top-ranked suspicious combination. A new test f is generated for τ such that f contains τ and the suspiciousness of the environment for τ is minimized in f. When such a test case passes, this combination is removed from the suspicious set. When such a test fails, the failure is more likely due to this combination since the suspiciousness of its environment is minimized. Therefore, the suspicious combination should be marked as an inducing combination. To increase the confidence, a user-specified number of tests can be generated for a top-ranked suspicious combination.

One approach to generate a given number n of new tests with minimum ρ_e for a suspicious combination is to generate all possible tests containing this combination, remove tests which already exist in F, and then select n tests which have the lowest ρ_e. This algorithm is very expensive. We next describe a more efficient but heuristic algorithm.

First, we generate a base test f as follows. For each parameter involved in τ, we give the same value in f as in τ. Doing so makes sure that f contains τ. For each parameter in the environment of τ, i.e., each parameter that is not involved in τ, we choose a value (or component) whose suspiciousness ρ is the minimum. If there is more than one value with minimum ρ, one of them is selected randomly.

Next, we check whether the base test f is new, i.e., making sure that f has not been executed before. If so, f is returned as the new test that contains τ and has minimum ρ_e. If not, we pick one parameter randomly and change its value to a value with the next minimum ρ. Again, this test is checked to see whether it is a new test. These steps are repeated until a new test is found, or the number of attempts for finding a new test case reaches a predefined number. The process is repeated until a desired number of new tests are generated.

If BEN does not find any new test, the combination is marked as an inducing combination, because it is likely that all the test cases containing this combination have been executed (and all of them must have failed).

The newly generated tests, i.e., those in set F', are executed. If all the tests fail, the suspicious combination, τ, is marked as an inducing (line 16 -Figure 4-2). If not, F' is added to the test set (line 18 - Figure 4-2) to refine the suspicious combinations set in the next iteration. By adding F' to the test set the suspicious combination τ and all other suspicious combinations appear in passed tests of F' are removed from the suspicious combinations set. Therefore, the number of suspicions combinations could be reduced by the new tests added into the test set.

4.3.1.5 Discussion

To successfully identify an inducing combination, BEN must first identify the combination to be a suspicious combination. Assume that c is an inducing combination. Let t be the strength of the initial test set. We consider the following three cases.

Case (1): c is a t-way combination. As the initial test set is a t-way test set, there is at least one test that contains c, and all test cases containing c must fail, since c is inducing. Therefore, c is identified to be a suspicious combination.

Case (2): The size of c is less than t. All t-way combinations containing c are inducing combinations, and are identified to be suspicious combinations.

Case (3): The size of c is more than t. The initial t-way test set is not guaranteed to cover every combination whose size is more than t. If c appears in the initial t-way test set or the newly generated tests, and thus causes a test containing it to fail, it is identified to be a suspicious combination when l is equal to the size of c.

Let c be an inducing combination that has been identified as a suspicious combination. If it is in the top-ranked set, i.e., the set of a user-specified number of top-ranked combinations, all the tests generated for c fail since they contain c. Therefore, c is identified to be an inducing combination.

Now consider the case that c is not in the top-ranked set. Without loss of generality, assume that every combination c’ in the top-ranked set is not inducing. If any new test generated for c’ passes, c’ is no longer suspicious and is thus removed from π. This will cause c to move up in the ranking. With a sufficient number of iterations, c will be moved into the top-ranked set and will be identified to be an inducing combination.

If all the tests generated for *c*' fail, *c*' will be reported as an inducing combination. As discussed earlier, a new test for *c*' is generated such that if it fails, it is likely due to *c*' Thus, if all the tests generated for *c*' fail, *c*' is likely to have a high inducing probability even if it is not truly inducing.

BEN provides the user with several options to control the cost and effectiveness of the process. First, BEN allows the user to specify the number of new tests generated for each top-ranked suspicious combination. The more tests generated, the more effort it takes to execute them, but the more confidence we have about the identified inducing combinations.

Second, BEN allows the user to specify the size of the top-ranked set for which new tests will be generated. The bigger the top-ranked set, the more effort to generate and execute the new tests, but the faster an inducing combination may be identified. This is because if an inducing combination c is included in the top-ranked set, c is identified to be an inducing combination in the first iteration. Otherwise, it may take multiple iterations for c to move up into the top-ranked set.

Finally, BEN allows the user to stop the first phase (and move to the second phase) in the following three ways if there is limited resource:

The user could define the maximum number of iterations for the *identify* algorithm. That is, if none of the two stopping conditions is satisfied after a specified number of iterations, the *identify* algorithm stops and returns null. Returning null shows that there is no inducing combination of the current size; therefore, the main framework increments the size in the next iteration.

The user could decide to stop at the end of each iteration of the framework. In this case, the top ranked suspicious combination would be reported as an inducing combination.

The user could define the maximum size of inducing combination. If the maximum size is reached but BEN still does not find any inducing combination, the top ranked suspicious combination in the last iteration is reported as an inducing combination. Recall that in the worst case, the size of inducing combination is equal to the number of parameters.

4.3.2 Phase 2: Faulty statement localization

Figure 4-3 shows the algorithm used by BEN to localize faulty statements. It consists of two major steps: (1) Test Generation: In this step, we generate a small group of tests. The group contains one failed test, which is referred to as the core member, and at most l passed tests, where l is the size of the inducing combination. The passed tests are referred to as the derived members. Each derived member is expected to produce a similar execution trace as the core member. (2) Rank Generation: In this step, we compare the spectrum of the core member to the spectrum of each derived member, and then produce a ranking of statements in terms of their likelihood of being faulty. More details of these two steps are explained in the following sections.

4.3.2.1 Test generation

In this step, as shown in Figure 4-3 (lines 2-9), a group of tests, M, which includes the core member f and at most l derived members, are generated. Let c be the l-way inducing combination identified in Phase 1. The core member f is created such that it contains c and the suspiciousness of environment of c in f is minimized (line 3). To generate such a test, the same algorithm used for test generation in Phase 1 is applied: For each parameter p involved in c, f has the same value for p as c, i.e. $c \subset f$, and for each parameter p that does not appear in c, f takes a value that has the minimum suspiciousness value among all the values of p. As discussed later, the reason why we want to minimize the

```
Algorithm localize
1   // Step 1. Generate core and derived members
2   let c be the inducing combination identified in Phase 1
3   let M be an empty set
4   generate core member $f \in \Gamma$ such that $c \subset f$ and for all $o \in f$ and $o \notin c$,
    $\rho(o: p \leftarrow v) = \min_{v_i \in d}\{\rho(p \leftarrow v_i)\}$
5   for (each component $o \in c$) {
6       generate the derived member candidate set $M_o$ for component o based on $\Theta$
        and $\Omega$
7   select derived member $m_o \in M_o$ where $r(m_o) = pass$ and $|\gamma(f) - \gamma(m_o)| >$
    $0$ and $|\gamma(f) - \gamma(m_o)| = \min_{m \in M_o}\{|\gamma(f) - \gamma(m)|\}$
8       $M = M \cup \{m_o\}$
9   }
10  // Step 2. Rank statements
11  for each statement $s \in S$ {
12      for all derived members $in\ m \in M$)
13          compute $\rho(s, m)$ with respect of core member f, based on formula 5
14      $\rho(s) = \sum_{m \in M} \rho(s, m) / |M|$
15  }
16  Let $R$ be the ranking of statement in the non-increasing order of $\rho(s)$
17  return $R$
```

Figure 4-3. The Localize algorithm

suspiciousness of the environment of c is to maximize the likelihood of a derived member to be a passing test.

The core member f is likely to fail, since it contains the inducing combination c identified in the first phase. Next, for each component $o \in c$, a set of derived member candidates, M_o, is generated. A derived member candidate $m_i \in M_o$ is generated such that it has the same values as f for all parameters except for one component $o \in c$. The component o is replaced with another component, o', of the same parameter with the minimum suspiciousness value. Note that a parameter may have multiple least suspicious components, i.e., multiple components with the minimum suspiciousness value. So, all the tests in M_o are different from the core member and from each other in one component, o.

$$f \qquad \{\overbrace{o_1, o_2, \dots, o_l}^{c}, o_{l+1}, o_{l+2} \dots, o_k\} \qquad \text{Core}$$

$$M_{o_1}\left\{\begin{array}{c} \{\mathbf{o_1^1}, o_2, \dots, o_l, o_{l+1}, o_{l+2} \dots, o_k\} \\ \{\mathbf{o_1^2}, o_2, \dots, o_l, o_{l+1}, o_{l+2} \dots, o_k\} \\ \dots \end{array}\right\}$$

Figure 4-4. Generation of the candidate set M_{o_1}

Figure 4-4 shows how the derived member candidate set, or simply candidate set M_{o_1} is generated from the core member f. (In the remainder of this paper, we will refer to a derived member candidate set as a candidate set if there is no ambiguity.) The core member f contains k components, $o_1, o_2 \dots, o_k$, where k is the number of parameters. Without loss of generality, assume that the first l components in f, i.e., $o_1, o_2 \dots, o_l$, are in the inducing combination c. As shown Figure 4-4, each test in candidate set M_{o_1} is different from the core member f in component $o_1 \in c$. The o_1 component is replaced with $o_1^j = p_1 \longleftarrow v_j$ where o_1^j is a least suspicious component of p_1. For each least suspicious component p_1, one derived candidate test is generated. Formally:

$$\rho(o_1^1 = p_1 \longleftarrow v_1) = \rho(o_1^2 = p_1 \longleftarrow v_2) \dots = \min_{\forall j \in d_1} \rho\,(p_1 \longleftarrow v_j)$$

The number of tests in M_{o_1} depends on the number of least suspicious components of parameter p_1. Candidate tests are likely to pass. First, the replacement effectively removes inducing combination c from tests. Second, the use of a least suspicious component for the replacement, and having the suspiciousness of the environment minimized reduce the chance of introducing another inducing combination to the test.

Next, a derived member m_o is selected from each candidate set M_o (line 7). There are two criteria for derived member m_o. First, it must pass. Second, it has the minimum positive spectrum difference with the core member f among all the passed tests in M_o. Formally, $|\gamma(f) - \gamma(m_o)| = \min_{\substack{m \in M_o \wedge \\ r(m)=pass}} \{|\gamma(f) - \gamma(m)|\}$ and $|\gamma(f) - \gamma(m_o)| > 0$.

If there is more than one test that satisfies the two criteria, one of them is selected randomly. All the derived members are stored in a set called M (line 8). Figure 4-5 shows the core member f and the set M of derived members.

The execution trace of a derived member $m_i \in M$ is likely to be very similar to the execution trace of the core member, because these two tests only differ in one value, and they have the minimum spectrum differences among other similar tests. Since all the derived members m_i pass whereas the core member f fail, the faulty statement is very likely to be one of the statements that appear in the execution trace of f but do not appear in the execution trace of $m_1, m_2 \dots m_l$.

4.3.2.2 Rank generation

In this step, BEN computes the suspiciousness of statements and then ranks them in terms of their likelihood to be faulty by analyzing the spectrums of the core member and derived members. The suspiciousness of statement s is denoted by $\rho(s)$ and computed by analyzing the spectrums of the core member and derived members. The suspiciousness of statement s is the average of suspiciousness of s with respect to every derived members.

$$\begin{array}{ll}
f & \{\overbrace{o_1, o_2, \dots, o_l}^{c}, o_{l+1}, o_{l+2} \dots, o_k\} \quad \text{Core} \\
M & \left\{ \begin{array}{l}
\{\mathbf{o'_1}, o_2, \dots, o_l, o_{l+1}, o_{l+2} \dots, o_k\} \\
\{o_1, \mathbf{o'_2}, \dots, o_l, o_{l+1}, o_{l+2} \dots, o_k\} \\
\qquad\qquad \dots \\
\{o_1, o_2, \dots, \mathbf{o'_l}, o_{l+1}, o_{l+2} \dots, o_k\}
\end{array} \right.
\end{array}$$

Figure 4-5.The core and derived members

Formally:

$$\rho(s) = \sum_{m_i \in M} \rho(s, m_i)/(|M|) \qquad (4)$$

where $\rho(s, m_i)$ is the suspiciousness of s with respect to a derived member m_i and is computed by the following formula:

$$\rho(s, m_i) = \begin{cases} 1 & \text{if } \gamma(s, f) = \text{ true and } \gamma(s, m_i) = \text{ false} \\ 0.5 & \text{if } \gamma(s, f) = \gamma(s, m_i) = \text{ true} \\ 0 & \text{if } \gamma(s, f) = \text{false} \end{cases} \qquad (5)$$

The idea behind formula (5) is the following. Statements that are only executed by the core member f are most suspicious and are given 1 as their suspiciousness value. Statements that are executed by both the core member and a derived member are less suspicious, and are given 0.5 as their suspiciousness value. Note that the execution of a faulty statement by a test does not necessarily make the test fail. For example, if there exists a fault in a conditional expression, this fault can be executed by all tests but only cause some to fail. Finally, statements that are not executed by f are not suspicious.

For example, if there are two derived members in M, m_1 and m_2, and the core member is f. Assume that a statement s is executed by f and m_2, but not by m_1 The suspiciousnes $\rho(s)$ of s would be 0.75. This is because $\rho(s, m_1) = 1$ and $\rho(s, m_2) = 0.5$, and the average of $\rho(s, m_1)$ and $\rho(s, m_2)$ would be 0.75.

The higher the suspiciousness value of a statement, the more likely this statement is faulty. We rank statements by a non-ascending order of their suspiciousness value. To locate the faulty statement, statements in the top rank are examined first, and then statements in the next rank, until the faulty statement is found.

4.3.2.3 Discussion

The effectiveness of our approach in this phase depends to some extent on the quality of the inducing combination c identified in the first phase. If combination c is truly inducing, the core member generated by our approach, i.e., the one that contains this combination and minimizes the suspiciousness of its environment, must fail. However, if c is not truly inducing, but with a high inducing probability, the core member still has a high probably to fail. The experimental results in Section 4.5.2.1.1 and 4.5.2.2.1 show that Phase 1 of our approach can identify truly inducing combinations or combinations that have a high inducing probability.

If the core member generated in the second phase does not fail, we pick a test from the initial t-way test set that contains c as the core member. Since c is identified as an inducing combination, there must exist at least one failed test that contains c in the initial test set. (Otherwise, c would not even be a suspicious combination.) In this case, the suspiciousness of environment of c in this test may not be minimized. This may reduce the probability for the derived members to pass.

If BEN could not find any passed test in a candidate set M_o for a component o (in the inducing combination), it ignores the candidate set and thus no derived member is generated for component o. In case that no derived member is generated for all the components in the inducing combination, BEN picks a passed test from the test set such that the number of components that differ between the passed test and the core member is minimized. In this case, the difference between the core member and this derived member may not be minimized, which might affect the effectiveness of our approach. We believe the chance for this case, i.e., all the tests in all the candidate sets for all the components fail, to occur is small, which is consistent with our experiments in which it occurred in 3 versions to the total of 124 versions of our subject programs.

4.3.3 Complexity analysis

In our analysis, we do not consider the complexity of constraint solving and the cost of test execution. Our approach uses a third-party solver for constraint solving. The cost of test execution depends on the subject program.

Let k be the number of parameters, t the strength of the initial test set and d the largest domain size of the parameters. Let N be the number of tests in the current iteration, which includes the tests in the initial test set and the tests generated at the previous iterations. Note that the number of test generated at each iteration depends on two user-specified numbers, i.e., the size of the top-ranked set consisting of suspicious combinations for which tests are to be generated, and the number of tests to be generated for each suspicious combination in the top-ranked set. Assume that the inducing combination is of size l which is greater than or equal to t. The maximum number of l-way combinations contained in the test set is $\eta = \binom{\mathrm{k}}{l}\mathrm{N}$.

To determine whether a combination is suspicious, the identify algorithm needs to check if the combination appears in any passed test, which takes $\mathrm{O}(\mathrm{N} \times l)$. Therefore, building the suspicious combinations set takes $\eta \times \mathrm{O}(\mathrm{N} \times l)$. Next, the identify algorithm computes the suspiciousness value for all the components, which includes computing the frequency of each component in the suspicious combination set, test set and failed tests. Computing the frequency in the suspicious combination set dominates the other two, which takes $\mathrm{O}(\eta)$ for each component. The maximum number of components is $\mathrm{k} \times \mathrm{d}$. Thus, computing suspiciousness values for all the components takes $\mathrm{k} \times \mathrm{d} \times \mathrm{O}(\eta)$.

After having suspiciousness values of all the components, computing suspiciousness of each combination (ρ_c) takes l, and thus $l \times \mathrm{O}(\eta)$ for all the combinations. To compute ρ_e of a combination, BEN first searches in the test set to find all the failed tests that contain this combination, which takes $l \times \mathrm{O}(\mathrm{N})$. Next, for each of these failed tests, it computes the average suspiciousness value of $\mathrm{k} - l$ components in the environment. Therefore, it takes in total $l \times (\mathrm{k} - l) \times \mathrm{O}(\eta) \times \mathrm{O}(\mathrm{N})$. Finally, BEN finds the minimum suspiciousness of the environment among all these failed tests, which takes $\mathrm{O}(\mathrm{N})$.

Therefore, the complexity of computing ρ_e for all the combination is $l \times (k - l) \times O(\eta) \times O(N)$.

The identify algorithm sorts the set of suspicious combinations three times, once for each ranking R_c, R_e, and R, taking $O(\eta \times \log(\eta))$. This dominates the complexity of the rank generation step, if the number of tests N is far less than the number of combinations, η.

The test generation step needs to select $(k - l)$ values with minimum ρ first, which takes $(k - l) \times O(d)$. Then it needs to check whether it is new, which is $O(N)$. Since k, l and d are smaller than η, $O(\eta \times \log(\eta))$ dominates the complexity of the rank generation and test generation steps. Therefore the complexity of the identify algorithm is $O(\eta \times \log(\eta))$. In the worst case, the identify algorithm is called $(k - t)$ times. Thus, the complexity of this phase is $(k - t) \times O(\eta \times \log(\eta))$.

In Phase 2, in order to generate the core member, we need to select values with minimum suspiciousness for $(k - l)$ components, which takes $(k - l) \times O(d)$. There are l candidate sets, and for each it takes $O(d)$ to find components with minimum ρ. Therefore, generating all candidate sets takes $l \times O(d)$.

Each candidate set at most contains $d - 1$ derived members. Selecting a test with minimum difference in the spectrum with the core member takes $[l \times (d - 1)] \times |S|$, where $|S|$ is the number of statements of the program. The complexity of selecting a test, $[l \times (d - 1)] \times |S|$, dominates the complexity of this step.

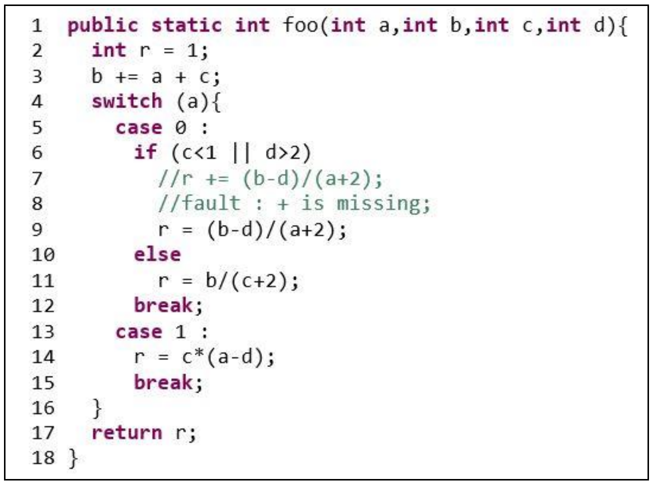
In the rank generation step, the complexity of assigning a suspiciousness value to each statement with respect to the l derived members is $O(l)$. So for all the statements S of the program, it takes $|S| \times O(l)$. Then all the statements need to be sorted to rank the statements, which is $O(|S| \times \log(|S|))$. Since l is typically much smaller than the program size $|S|$, this sorting operation dominates the complexity of this part. The complexity of the rank generation step, $O(|S| \times \log(|S|))$, dominates the complexity of this phase.

Depending on the programs size, $|S|$ and the number of suspicious combinations, η, the complexity of Phase 1 or Phase 2 may dominate the complexity of BEN.

4.4 EXAMPLE

In this section, we illustrate our approach using an example program shown in Figure 4-6. Method *foo* has a fault in line 9. The correct statement should be $r += (b - d)/(a + 2)$, but operator "+" is missing. The input parameter model consists of $P = \{a, b, c, d\}$, and $d_a = \{0,1\}$, $d_b = \{0,1\}$, $d_c = \{0,1,2\}$, and $d_d = \{0,1,2,3\}$. The faulty statement is reached when a is 0 and c is 0 or d is 3. So there are two inducing combinations $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.

Assume that the program is tested by a two-way test set. The test result is shown in Table1, where 3 out of 12 tests fail. Test cases #1 and #7 fail because they contain combination $(a \leftarrow 0, c \leftarrow 0)$. Test case #10 fails because it contains $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.



```
1  public static int foo(int a,int b,int c,int d){
2    int r = 1;
3    b += a + c;
4    switch (a){
5      case 0 :
6        if (c<1 || d>2)
7          //r += (b-d)/(a+2);
8          //fault : + is missing;
9          r = (b-d)/(a+2);
10       else
11         r = b/(c+2);
12       break;
13     case 1 :
14       r = c*(a-d);
15       break;
16   }
17   return r;
18 }
```

Figure 4-6. An example faulty program

Table 4-1. two-way Test Set and Status

Test #	a	b	c	d	Status
1	0	0	0	0	fail
2	1	1	1	0	pass
3	0	1	2	0	pass
4	1	0	0	1	pass
5	0	0	1	1	pass
6	1	1	2	1	pass
7	0	1	0	2	fail
8	1	0	1	2	pass
9	0	0	2	2	pass
10	0	1	0	3	fail
11	1	0	1	3	pass
12	1	0	2	3	pass

4.4.1 Phase 1: Inducing combination identification

Table 4-1 shows a t-way test set with test execution statuses for the example program. In the first iteration, the identify algorithm identifies nine suspicious combinations (Figure 4-2, line 3) which are listed in the first column of Table 4-2Table 4-2. Then the algorithm computes the suspiciousness values of all the (seven) components that appear in one or more of these suspicious combinations.

Table 4-2. Suspicious combinations and their corresponding values

Suspicious Combination	$\boldsymbol{\rho_c}$	$\boldsymbol{R_c}$	$\boldsymbol{\rho_e}$	$\boldsymbol{R_e}$	$\boldsymbol{R_c + R_e}$	$\boldsymbol{R}$
$a \leftarrow 0, c \leftarrow 0$	0.6713	1	0.2460	1	2	1
$b \leftarrow 1, c \leftarrow 0$	0.6176	2	0.4352	3	5	2
$c \leftarrow 0, d \leftarrow 0$	0.5324	4	0.3849	2	6	3
$c \leftarrow 0, d \leftarrow 3$	0.5509	3	0.5204	4	7	4
$c \leftarrow 0, d \leftarrow 2$	0.5324	4	0.5204	4	8	5
$a \leftarrow 0, d \leftarrow 3$	0.4537	5	0.6176	5	10	6
$b \leftarrow 1, d \leftarrow 3$	0.4000	6	0.6713	6	12	7
$b \leftarrow 1, d \leftarrow 2$	0.3815	7	0.6713	6	13	8
$b \leftarrow 0, d \leftarrow 0$	0.2460	8	0.6713	6	14	9

For example, component $c \leftarrow 0$ appears in all of the three failed test cases, so $u(c \leftarrow 0) = 1$. Also, it appears a total of four tests three of which are failed tests, so $v(c \leftarrow 0) = 3/4$; 5 out of 9 members of suspicious combinations set contain $c \leftarrow 0$, so $w(c \leftarrow 0) = 5/9$. The computations for all the seven components are as follows:

$$\rho(c \leftarrow 0) = \frac{1}{3} \times \left(1 + \frac{3}{4} + \frac{5}{9}\right) = 0.7685$$
$$\rho(d \leftarrow 0) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$
$$\rho(d \leftarrow 2) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$
$$\rho(d \leftarrow 3) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{3} + \frac{3}{9}\right) = 0.3333$$
$$\rho(b \leftarrow 0) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{7} + \frac{1}{9}\right) = 0.1958$$
$$\rho(b \leftarrow 1) = \frac{1}{3} \times \left(\frac{2}{3} + \frac{2}{5} + \frac{3}{9}\right) = 0.4667$$
$$\rho(a \leftarrow 0) = \frac{1}{3} \times \left(1 + \frac{3}{6} + \frac{2}{9}\right) = 0.5741$$

Table 4-3 illustrates the suspiciousness values of all the components. The suspiciousness values for the components that do not appear in any suspicious combination are zero.

According to formula (2), ρ_c for a suspicious combination τ is the average suspiciousness of the components that τ contains. For example, in combination

Table 4-3. Suspiciousness of components

Parameter	Value	ρ_c	Parameter	Value	ρ_c
a	0	0.5741	b	0	0.1958
	1	0		1	0.4667
c	0	0.7685	d	0	0.2963
	1	0		1	0
	2	0		2	0.2963
				3	0.3333

$(a \leftarrow 0, c \leftarrow 0)$, ρ_c is $(0.5741 + 0.7685)/2 = 0.6713$. After computing ρ_c for all suspicious combinations, we rank them based on the non-ascending order of ρ_c. The values of ρ_c and R_c for each suspicious combination are shown in the second and third columns of Table 4-2Table 4-2.

Next, we compute ρ_e for each suspicious combination using formula (3). For example, there are three test cases, test #1, test #7, and test #10, that contain $(a \leftarrow 0, c \leftarrow 0)$. Therefore,

$$\rho_e(a \leftarrow 0, c \leftarrow 0) = \min\left(\left(\frac{\rho(b \leftarrow 0) + \rho(d \leftarrow 0)}{2}\right) = 0.2460, \left(\frac{\rho(b \leftarrow 1) + \rho(d \leftarrow 2)}{2}\right)\right.$$

$$\left. = 0.3815, \left(\frac{\rho(b \leftarrow 1) + \rho(d \leftarrow 3)}{2}\right) = 0.4000\right) = 0.2460$$

Next we rank suspicious combinations by a non-descending order of ρ_e, as shown in column R_e of Table 4-2.

Finally, the two rankings in columns R_c and R_e are combined to produce a final ranking of the suspicious components (column R). In this final ranking, inducing combination $(a \leftarrow 0, c \leftarrow 0)$ is ranked on the top, and the other $(a \leftarrow 0, d \leftarrow 3)$ is ranked 6th.

Then, a new test is generated for the top ranked suspicious combination $(a \leftarrow 0, c \leftarrow 0)$. We assign values to parameters in its environment, i.e., b and d, such that the suspiciousness of each value is minimum. For b, 0 is selected, as $\min(\rho(b \leftarrow 0) = 0.1958, \rho(b \leftarrow 1) = 0.4667) = 0.1958$. For d, 1 is selected as $\min(\rho(d \leftarrow 0) = 0.2963, \rho(d \leftarrow 1) = 0, \rho(d \leftarrow 2) = 0.2963, \rho(d \leftarrow 3) = 0.3333) = 0$. So a new test $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$ is generated.

The newly generated test, $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$, fails. For simplicity of presentation, assume that only one test is generated for this combination. (If more tests are generated, all of them would fail too in this example.) Therefore, suspicious

$$\begin{array}{lll} \mathrm{f} & (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0, \mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 1) & \text{Fail} \\ \mathrm{M}_{\mathrm{a}\leftarrow 0} & \left\{ \mathrm{m}_{\mathrm{a}\leftarrow 0} = (\mathbf{a} \leftarrow \mathbf{1}, \mathrm{b} \leftarrow 0, \mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 1) \right. & \left. \text{Pass} \right\} \end{array}$$

Figure 4-7. Candidate set of $\mathrm{M}_{\mathrm{a}\leftarrow 0}$

combination $(\mathrm{a} \leftarrow 0, \mathrm{c} \leftarrow 0)$ is marked as inducing combination and returned by the identify algorithm. The main framework of the first phase stops at the end of the first iteration and reports $(\mathrm{a} \leftarrow 0, \mathrm{c} \leftarrow 0)$ as the inducing combination.

4.4.2 Phase 2: Faulty statement localization

In the test generation step of the second phase, the core member $\mathrm{f} = (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0, c \leftarrow 0, \mathrm{d} \leftarrow 1)$ is generated. It contains the inducing combination $(\mathrm{a} \leftarrow 0, \mathrm{c} \leftarrow 0)$, and two components $\mathrm{b} \leftarrow 0$ and $\mathrm{d} \leftarrow 1$ which have the minimum suspiciousness value (among components of the same parameter) as shown in Table 4-3. The core member fails.

As shown in Figure 4-7 the candidate set $\mathrm{M}_{\mathrm{a}\leftarrow 0}$ of component $\mathrm{a} \leftarrow 0$ contains only one test, $(\mathrm{a} \leftarrow 1, \mathrm{b} \leftarrow 0, \mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 1)$, since $\mathrm{a} \leftarrow 1$ is the only component with minimum suspiciousness. The test passes and therefore is selected as a derived member, $\mathrm{m}_{\mathrm{a}\leftarrow 0}$.

The second candidate set $\mathrm{M}_{\mathrm{c}\leftarrow 0}$, shown in Figure 4-8 has two tests, where component $\mathrm{c} \leftarrow 0$ from the core member is replaced with $\mathrm{c} \leftarrow 1$ and $\mathrm{c} \leftarrow 2$, since $\min(\rho(\mathrm{c} \leftarrow 0) = 0.7685, \rho(\mathrm{c} \leftarrow 1) = 0, \rho(\mathrm{c} \leftarrow 2) = 0) = 0$ and both components $\mathrm{c} \leftarrow 1$ and $\mathrm{c} \leftarrow 2$ have the minimum suspiciousness value, 0.

To select a derived member $\mathrm{m}_{\mathrm{c}\leftarrow 0}$ from candidate set $\mathrm{M}_{\mathrm{c}\leftarrow 0}$, both tests $\mathrm{m}^1_{c\leftarrow 0}$ and $\mathrm{m}^2_{c\leftarrow 0}$ are executed and their execution traces are recorded. A test is selected as a derived member if it passes and it has minimum spectrum difference with the core member.

$$\begin{array}{lll} \mathrm{f} & (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0, \mathrm{c} \leftarrow 0, \mathrm{d} \leftarrow 1) & \text{Fail} \\ \mathrm{M}_{\mathrm{c}\leftarrow 0} & \left\{ \begin{array}{l} \mathrm{m}^1_{c\leftarrow 0} = (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0, \mathbf{c} \leftarrow \mathbf{1}, \mathrm{d} \leftarrow 1) \\ \mathrm{m}^2_{c\leftarrow 0} = (\mathrm{a} \leftarrow 0, \mathrm{b} \leftarrow 0, \mathbf{c} \leftarrow \mathbf{2}, \mathrm{d} \leftarrow 1) \end{array} \right. & \left. \begin{array}{l} \text{Pass} \\ \text{Pass} \end{array} \right\} \end{array}$$

Figure 4-8. Candidate set of $\mathrm{M}_{\mathrm{c}\leftarrow 0}$

Both tests $m^1_{c\leftarrow 0}$ and $m^2_{c\leftarrow 0}$ pass. The spectra of the core member, f, and two members of candidate set $M_{c\leftarrow 0}$ are shown in Table 4-4. The second column of Table 4-4 shows the program statements. The third column shows the spectra of the core member f. The fourth column shows the program spectrum of $m^1_{c\leftarrow 0}$. The fifth column contains 1 if a statement is executed by the core member but not by $m^1_{c\leftarrow 0}$. Otherwise it contains 0. The sixth column show the program spectrum of $m^2_{c\leftarrow 0}$. The last column is assigned to 1 iff the corresponding statement is executed by the core member and not by $m^2_{c\leftarrow 0}$. The fifth and

Table 4-4. Program spectra of core and candidate set $\mathbf{M_{c\leftarrow 0}}$

	Subject Program	$\gamma(s,f)$	$\gamma(s, m^1_{c\leftarrow 0})$	$\gamma(f)-\gamma(m^1_{c\leftarrow 0})$	$\gamma(s, m^2_{c\leftarrow 0})$	$\gamma(f)-\gamma(m^2_{c\leftarrow 0})$
1	public static int foo(int a,int b, int c,int d){	True	True	0	True	0
2	int r = 1;	True	True	0	True	0
3	b += a + c;	True	True	0	True	0
4	switch (a){	True	True	0	True	0
5	case 0 :	True	True	0	True	0
6	if (c<1 || d>2)	True	True	0	True	0
7	//r += (b-d)/(a+2);	-	-	0	-	0
8	//fault:+is missing;	-	-	0	-	0
9	r = (b-d)/(a+2);	True	False	1	False	1
10	else	False	True	0	True	0
11	r = b/(c+2);	False	True	0	True	0
12	break;	True	True	0	True	0
13	case 1 :	False	False	0	False	0
14	r = c*(a-d);	False	False	0	False	0
15	break;	False	False	0	False	0
16	}	True	True	0	True	0
17	return r;	True	True	0	True	0
18	}	True	True	0	True	0
$\lvert\gamma(f)-\gamma(m_{c\leftarrow 0})\rvert$		-	-	1	-	1

$$\begin{array}{ll} \mathrm{f} & (a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1) \text{ Fail} \\ \mathrm{M} & \left\{ \begin{array}{l} m_{a\leftarrow 0} = (\mathbf{a} \leftarrow \mathbf{1}, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1) \text{ Pass} \\ m_{c\leftarrow 0} = (a \leftarrow 0, b \leftarrow 0, \mathbf{c} \leftarrow \mathbf{1}, d \leftarrow 1) \text{ Pass} \end{array} \right\} \end{array}$$

Figure 4-9. Core and derived members of the example program

seventh columns are used to compute the spectrum differences of the core and $m^1_{c\leftarrow 0}$ or $m^2_{c\leftarrow 0}$. The last row of Table 4-4 shows the spectrum difference of the core and each member of $M_{c\leftarrow 0}$, which are computed by the summation of fifth and last columns.

Since two tests $m^1_{c\leftarrow 0}$ and $m^2_{c\leftarrow 0}$ both pass and have the same spectrum difference with the core member. Test $m^1_{c\leftarrow 0}$ is selected randomly as the derived member $m_{c\leftarrow 0}$. Figure 4-9 shows the output of the test generation step, the core member, f, in the first row and the derived members set M, which contains two tests.

In the rank generation step, the spectrum of the core member is compared to that of each derived member $m \in M$ and the statement suspiciousness with respect to m is computed. Table 4-5 shows the program spectra for the core member and two derived members in columns 3 to 5. The suspiciousness values for each statement with respect to derived tests $m_{a\leftarrow 0}$ and $m_{c\leftarrow 0}$ are shown in columns 6 and 7 ($\rho(s, m_{a\leftarrow 0})$ and $\rho(s, m_{c\leftarrow 0})$) of Table 4-5, respectively. The last two columns of Table 4-5 show the statement suspiciousness and ranks. The faulty statement in line 9 is in the first rank.

Note that this example represents a best-case scenario of our approach. In the next section, we provide an experimental evaluation of our approach.

Table 4-5. Program spectra and statements suspiciousness values

	Subject Program	$\gamma(s,f)$	$\gamma(s,m_{a\leftarrow 0})$	$\gamma(s,m_{c\leftarrow 0})$	$\rho(s,m_{a\leftarrow 0})$	$\rho(s,m_{c\leftarrow 0})$	$\rho(s)$	Rank
1	**public static int** foo(**int** a,**int** b, **int** c,**int** d){	True	True	True	0.5	0.5	0.5	3
2	**int** r = 1;	True	True	True	0.5	0.5	0.5	3
3	b += a + c;	True	True	True	0.5	0.5	0.5	3
4	**switch** (a){	True	True	True	0.5	0.5	0.5	3
5	**case** 0 :	True	False	True	1	0.5	0.75	2
6	**if** (c<1 || d>2)	True	False	True	1	0.5	0.75	2
7	//r += (b-d)/(a+2);	-	-	-	-	-	-	-
8	//fault:+is missing;	-	-	-	-	-	-	-
9	r = (b-d)/(a+2);	True	False	False	1	1	1	1
10	**else**	False	False	True	0	0	0	4
11	r = b/(c+2);	False	False	True	0	0	0	4
12	**break**;	True	False	True	1	0.5	0.75	2
13	**case** 1 :	False	True	False	0	0	0	4
14	r = c*(a-d);	False	True	False	0	0	0	4
15	**break**;	False	True	False	0	0	0	4
16	}	True	True	True	0.5	0.5	0.5	3
17	**return** r;	True	True	True	0.5	0.5	0.5	3
18	}	True	True	True	0.5	0.5	0.5	3

4.5 EXPERIMENT

We built a tool called BEN [12] that implements our approach. (BEN is a Chinese word that means “root cause”.) BEN is available for public download [3]. For our experiment, we used the command line version of BEN on a set of nine benchmark programs.

Subject programs are selected from the SIR [31] benchmark, including seven small programs in the Siemens suite and two large real-life programs grep and gzip.

Furthermore, we conducted an experimental comparison between our approach and two well-known spectrum based approaches, Tarantula [18] and Ochiai [23].

4.5.1 Experimental design

4.5.1.1 Subject programs

BEN is applied to the Siemens suite and the *grep* and *gzip* programs from SIR [31]. The Siemens suite has been used to evaluate several fault localization techniques [17][15][34]. The two programs, *grep* and *gzip*, are significantly larger programs than the Siemens programs and are included to evaluate how our approach works on larger programs.

THE SIEMENS SUITE - The Siemens suite contains seven programs and each of these programs contains a number of faulty versions. The Siemens suite also provides an error-free version and a test set for each program. Table 4-6 represents properties of subject programs. The second column shows the number of lines of code for each program [31], including comments. The third column shows the size of executable code computed by Gcov 4.1.2 [9], and the last column indicates the number of faulty versions provided for each program. Note that the number lines of executable code is different from the number of lines code reported in [31], This is because the number of lines of executable code does not include commented lines, declaration lines, nor code in header files.

Both of the two programs, *printtokens* and *printtokens*2, are used to tokenize the input file and determine the type of each token. A token could be one of the following types: *identifier*, *special*, *keyword*, *number*, *comment*, *character constant* or *string constant*. Tokens of type *keyword* include *and*, *or*, *if*, *xor*, and *lambda*. Tokens of type *special* include *lparen*, *rparen*, *lsquare*, *rsquare*, *quote*, *bquote*, *comma* and *equalgreater*. *Comment* begins with semicolon and ends when a new line character is seen. *String constant* is a string enclosed by two double quotations. *Character* is a token starting with #.

Table 4-6. Characteristics of Siemens programs

Programs	LOC	#Lines of Executable Code	# of faulty versions
printtokens	726	188	7
printtokens2	570	201	10
replace	564	242	32
schedule	412	154	9
schedule2	374	127	10
tcas	173	65	41
totinfo	565	123	23

The *replace* program has three inputs, *pattern*, *substitute* and *input text*. The program finds every match of *pattern* in the *input text* and replaces it with *substitute*. The *pattern* is a restricted form of regular expression. The *substitute* is a string that allows three meta-characters to be used. These include "@t", which matches a tab; @n, which matches the end of a line, and &, which represents the string that matches the *pattern*. For example, if the string that matches *pattern* is *ab* and substitute is *a&c*, all *ab* strings in the file are replaced with *aabc*.

Two programs, *schedule* and *schedule2*, take the same input and produce the same output, but use different scheduling algorithms. The input includes: (1) three non-negative integers representing the number of processes in three different priority queues, *low*, *medium* and *high*; and (2) a list of commands that must be done on queues. There are seven commands, *new job*, *upgrade_prio*, *block*, *unblock*, *quantum_expire*, *finish* and *flush*. The output of these two programs is a list of numbers indicating the order in which the processes exit (from the scheduling system).

The *tcas* program is an aircraft collision avoidance system. It takes 12 numbers that represent different flight parameters of two aircrafts as input and generates as output a resolution advisory, which can be *unresolved*, *upward* and *downward*.

The *totinfo* program takes as input a file containing one or more tables. The program uses the notions of chi-square and degree of freedom to calculate whether the distribution of the numbers in these tables is logarithm gamma distribution. The output is the total degree of freedom of rows and columns and chi-square.

THE GREP PROGRAM - The *grep* program from SIR, has 10068 lines of code [31]. The *grep* program has two input parameters, patterns and files. It prints lines in each file that contain a match of any of the patterns. While the *grep* program can take multiple patterns and files, we only used a single pattern and file in this experiment. In addition, different options can be used to control the behavior of the *grep* program. For example, option "–w" causes the program to print only lines containing whole-word matches.

The grep program can take four different types of patterns: (1) basic-regexp: a basic regular expression. (2) extended-regexp: an extended regular expression. (3) fixed-strings: a list of fixed strings. (4) perl-regexp: a Perl regular expression. In this experiment, we focused on extended-regexp.

There are five versions of *grep* in the benchmark, and each of them has a number of seeded faults. All versions were written in C, and have ten header files and one C file. Table 4-7 shows the size of executable code computed by Gcov 4.1.2 and number of faulty versions for each version.

Table 4-7. Characteristics of *grep* versions

Programs	#Lines of Executable Code	# of faulty versions	Grep version
grep1	3078	18	2.2
grep2	3224	8	2.3
grep3	3294	18	2.4
grep4	3313	12	2.4.1
grep5	3314	1	2.4.2

The last column of Table 4-7 indicates the release version of each program. Note that all the faults in a given version are different from the faults of the other versions, and reside in the code that has been modified from the previous version. For example, for *grep*2, all the faults residing on the code modified from *grep* 2.2 to *grep* 2.3.

THE GZIP PROGRAM - The *gzip* program from SIR has 5680 lines of code [31], which includes all header files, comments and declarations. The *gzip* program is used for file compression and decompression. The *gzip* input includes 13 options and a list of file. For example "-S" option uses to define the suffix of the result file, where the default is ".gz".

There are five versions of *gzip* in the benchmark, and each of them has a number of seeded faults. All versions were written in C, and have 6 header files and one C file. Table 4-8 shows the number of lines of executable code computed by Gcov 4.1.2 and number of faulty versions for each (correct) version, in the second and third columns, respectively. The last column indicates the release version for each program. The base version is *gzip* 1.0.7. The faults for different *gzip* versions are different from each other except for one case where the first fault of *gzip5* is the same as the first fault of *gzip2*. In addition, all the faults reside in the code that has been modified from the previous version, except the fault mention above. For example, for *gzip2*, all the faults reside in the code modified from *gzip* 1.1.2 to *gzip* 1.2.2.

Table 4-8. Characteristics of *gzip* versions

Programs	#Lines of Executable Code	# of faulty versions	Gzip version
gzip1	1705	16	1.1.2
gzip2	2006	7	1.2.2
gzip3	1866	10	1.2.3
gzip4	1892	12	1.2.4
gzip5	1993	14	1.3

4.5.1.2 Initial test set

The input parameter model of each program is shown in Table 4-9. The detailed models are made available for review in [8]. Also, in [10], we explained how we modeled the input parameters of the Siemens programs to apply combinatorial testing.

The model column of Table 4-9 shows the number of parameters and their domain size. We represent it by $(d_1^{k_1} \times d_2^{k_2} \times \ldots)$, where $d_i^{k_i}$ indicates that there are k_i number of parameters with domain size as d_i. Note that $k_1 + k_2 + \cdots = k$, where k is the total number of parameters. For example, *totinfo* has six parameters, among which three parameters have a domain size of 3, two parameters have a domain size of 5, and one parameter has a domain size of 6.

The constraint column shows the number of constraints in each model. Constraints exclude invalid combinations from the resulting test set. Consider the input model of the *printtokens* program, which contains different positions for different tokens. For example, *keyword* and *identifier* are two types of tokens that could appear at the beginning, middle or end of the input stream. A constraint is needed to prevent having more than one type of token at the same position.

Table 4-9. Programs model

Programs		Model	#Constraints
Siemens Suite	printtokens	$(2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2)$	8
	replace	$(2^4 \times 4^{16})$	36
	schedule	$(2^1 \times 3^8 \times 8^2)$	0
	tcas	$(2^7 \times 3^2 \times 4^1 \times 10^2)$	0
	totinfo	$(3^3 \times 5^2 \times 6^1)$	0
grep		$(2^7 \times 4^1 \times 5^1 \times 6^3 \times 8^1 \times 9^1 \times 13^1)$	1
gzip		$(2^{11} \times 4^2)$	8

Note that programs *printtokens* and *printtokens2* share the same model, and so do programs *schedule* and *schedule2*. The model of *tcas* is the same as [19]. Also note that the models are built based on the specification of the programs, i.e., independent from their implementations. The SIR repository does not provide any specification for the *grep* and *gzip* programs. So we used the manual document from [15] and [16] as their specification.

We assume that boundary testing is done before combinatorial testing is applied. Combinatorial testing focuses on failures caused by interactions between parameters, while boundary testing focuses on failure caused by boundary values of individual parameters. We used the ACTS tool [2] to generate t-way test sets. For each program, we first test it with a 2-way test set. If a program is not killed by a 2-way test set, we increase the test strength and then test the program with a 3-way test set. This process is repeated until we reach strength 4.

Table 4-10 shows the number of versions killed by our test sets of different strengths for Siemens suite. Note that the column of t-way test set indicates all versions that are killed by t-way test set and not by $(t-1)$-way test set. For example, 17, 12 and 7 versions of *tcas* are killed by the 2-way, 3-way and 4-way test sets, respectively. The 12 versions that are killed by 3-way test set are different from 17 and 7 versions that are killed by 2-way and 4-way test set, respectively. Therefore, in total, 36 versions of *tcas* are killed. The same information for the *grep* and *gzip* programs is shown in Table 4-11 and Table 4-12.

We also executed all the tests in the test pool that come with each program in SIR. (We will refer to the test pools in SIR as the SIR test pools.) These test pools are created initially in a black box manner based on the tester’s understanding of the program’s functionality and knowledge of special and boundary values. Then, white-box tests are

Table 4-10. Test results for Siemens suite

Programs	#faulty versions	#Killed versions			
		2-way	*3-way*	*4-way*	*All*
Printtokens	7	3	0	0	3
Printtokens2	10	9	0	0	9
replace	32	32	0	0	32
schedule	9	7	0	0	7
schedule2	10	3	0	0	3
tcas	41	17	12	7	36
totinfo	23	5	7	0	12

Table 4-11. Test results for *grep*

Programs	#faulty versions	#Killed versions				#Killed versions by SIR test pool
		2-way	*3-way*	*4-way*	*All*	
grep1	18	4	0	0	4	4
grep2	8	0	0	0	0	4
grep3	18	4	0	0	4	7
grep4	12	2	0	0	2	2
grep5	1	0	0	0	0	0

Table 4-12. Test results for *gzip*

Programs	#faulty versions	#Killed versions				#Killed versions by SIR test pool
		2-way	*3-way*	*4-way*	*All*	
gzip1	16	6	0	0	6	7
gzip2	7	3	0	0	3	3
gzip3	10	0	0	0	0	0
gzip4	12	1	0	0	1	3
gzip5	14	3	0	0	3	4

created and added into the pools to ensure that each executable statement, branch, and definition-use pair in the error-free version was exercised [31]. All the faulty versions of the Siemens programs are killed by the test pools, except version 9 of *schedule2*.

Combinatorial testing does not kill it either. The results of executing the test pools on the *grep* and *gzip* programs are shown in the last column of Table 4-11 and Table 4-12.

For the *grep*1 program, both test sets, our combinatorial test set and the SIR test pool, killed four versions. Three out of these four versions are the same, and one is different. The combinatorial test set killed version 8 while the test pool killed version 7. The combinatorial test set did not kill version 7 because the particular value that triggers the fault was not modeled in our model.

Moreover, version 2 of *grep*4 was killed by the combinatorial test set but not by the test pool. However, the test pool killed version 10 which is due to a boundary value that is not handled correctly.

Note that 4 versions out of 18 versions of *grep*1 were killed by 2-way test set. However, in one of the killed versions, i.e., version 11, all the tests failed. Based on Assumption 3, BEN was not applied to this version.

4.5.1.3 Multiple-fault versions

To evaluate the effectiveness of our approach when the program under test has more than one fault, we create several multiple-fault versions for each program. To increase the diversity, different multiple-fault versions have different numbers of faults. Table 4-13 shows the number of faulty versions with the number of faults created for each program. For example, we created three versions with 2 faults and one version with 3 faults for *printtokens*.

To create multiple-fault versions, we randomly pick faults from faults that are detected by our combinatorial test sets. Consider the *schedule* program. There are nine faulty versions and each version has one fault. The combinatorial test set kills seven of them (Table 4-10), versions 1 to 7, and the other 2 versions, versions 9 and 10, were not

Table 4-13. Multiple-fault versions

Programs		# multiple-fault versions									
		2 faults	3 faults	4 faults	5 faults	6 faults	7 faults	8 faults	9 faults	10 faults	ALL
Siemens Suite	printtokens	3	1	0	0	0	0	0	0	0	4
	printtokens2	1	1	1	1	1	1	1	0	0	7
	replace	1	1	1	1	1	1	1	1	1	9
	schedule	1	1	1	1	1	0	0	0	0	5
	schedule2	2	0	0	0	0	0	0	0	0	2
	tcas	1	1	1	1	1	1	1	1	1	9
	totinfo	1	1	1	1	1	1	1	1	1	9
grep	grep1	3	1	0	0	0	0	0	0	0	4
	grep3	3	3	1	0	0	0	0	0	0	7
	grep4	1	0	0	0	0	0	0	0	0	1
gzip	gzip1	3	3	3	3	1	0	0	0	0	13
	gzip2	3	1	0	0	0	0	0	0	0	4
	gzip5	3	1	0	0	0	0	0	0	0	4

killed. To create multiple-fault versions with 2 faults, two faulty version from 1 to 7 are selected randomly.

For each program, we typically generate one multiple-fault version for a certain number of faults. The maximum number of multiple-fault versions for each program depends on the number of killed versions. When the total number of killed versions is large, e.g., *replace* and *tcas*, we create multiple-fault versions with a maximum number of 10 faults. When the total number of killed versions is small, e.g., *printtokens* and *schedule*2, more than one multiple-fault version is created for the same number of faults. Since the two large programs, *grep* and *gzip*, have a small number of killed versions, three multiple-fault versions created for each number of faults, if possible.

Since some faults may conflict with each other, combining them in one version is not possible. For example, the *schedule2* program has three killed versions, version 2, 3, and 7. Two faulty versions, version 3 and 7, conflict with each other. In version 7, the condition of an if statement is changed, while version 3, the whole block that contains the same if statement is missing. Therefore, having these two versions in one multiple-fault version is not possible. For the *schedule2* program, two multiple-fault versions with 2 faults are created. One of them contains the faults of versions 2 and 3, and the other contains the faults of versions 2 and 7.

Table 4-14 shows the result of combinatorial testing on multiple-fault versions. All of them are killed by a 2-way test set except one version of program *printtokens2* and one version of program tcas that are killed by a 3-way test set. In addition, all the tests in the 2-way test set failed for the version with 8 faults of the *replace* program, and therefore this version is ignored.

Table 4-14. Test results for multiple-fault versions

Programs		#faulty versions	#Killed versions		
			2-way	*3-way*	*All*
Siemens Suite	printtokens	4	4	0	4
	printtokens2	7	6	1	7
	replace	9	9	0	9
	schedule	5	5	0	5
	schedule2	2	2	0	2
	tcas	9	8	1	9
	totinfo	9	9	0	9
grep	grep1	4	4	0	4
	grep3	7	7	0	7
	grep4	1	1	0	1
gzip	gzip1	13	13	0	13
	gzip2	4	4	0	4
	gzip5	4	4	0	4

4.5.1.4 Trace collection

We used Gcov 4.1.2 [9] to collect execution trace. Gcov reports the number of times a statement is executed by a given test. A statement is included in the execution trace of a given test if and only if it is executed by the test for one or more times.

Gcov distinguishes between statements that are executable but are not executed and statements that are not executable. We used this information to compute the percentage of executable code that must be inspected to find the faulty statement. If a program crashes, Gcov does not report any coverage. To deal with this problem, we add a statement to call function gcov_flush before every statement. Note that this is only done after a program crashes.

4.5.1.5 BEN configuration

For our experiments, we configure BEN to generate five tests for each of the two top ranked suspicious combinations at each iteration. In addition, because of resource limitation, we limit the size of inducing combination to 6. If BEN does not find an inducing combination of size 5, BEN reports the top 6-way suspicious combination as an inducing.

4.5.1.6 Metrics

Recall that the output of BEN is a ranking of statements in terms of their likelihood to be faulty. In order to find the faulty statement, we inspect statements in the first rank, and then statements in the second rank, and continue to do so until we find the actual faulty statement. Statements in the same rank are inspected in the order that they appear in the program. We record the number of statements that must be inspected to find the actual faulty statement in each program to measure the effectiveness of our approach.

Moreover, the effectiveness of the first phase, i.e., identifying inducing combination, is measured by the inducing probability (Definition 8) of the identified combination. The higher inducing probability the identified inducing combination has, the more precise the approach is.

The efficiency of our approach is measured by two factors: the number of tests that are executed and the number of tests that are instrumented for trace collection. We show the number of tests executed in different stages of our approach, i.e., number of tests of the initial combinatorial test set, number of tests needed to identify inducing combinations (Phase 1), and number of tests needed to produce the ranking of faulty statements (Phase 2).

We also compare our approach to two approaches Tarantula and Ochiai in terms of effectiveness, i.e., the number of statements that must be inspected to find the actual faulty statement, and efficiency, i.e., the number of tests executed and the number of tests whose execution traces must be collected.

4.5.2 Results and discussion

In this section, we discuss the results of applying BEN to the subject programs. We first report the results of BEN on the single-fault programs, then on the multiple-fault programs. Next, we compare the results of BEN to two techniques, Tarantula and Ochiai. Finally, the threats to validity are discussed.

4.5.2.1 Results on single-fault versions

This section is divided into two subsections. The first subsection reports the result of the first phase, identifying inducing combination. The second subsection discusses the result of the second phase, faulty statement localization.

4.5.2.1.1 Phase 1: Identifying inducing combination

Table 4-15 shows the inducing probabilities of inducing combinations identified in the first phase. To compute the inducing probability for combination c, we generated and

Table 4-15. Inducing probabilities for single-fault versions

Programs		Test strength (t)	# of killed versions	Avg size of inducing combinations	Avg inducing probability of inducing combinations
Siemens Suite	printtokens	2	3	3	1
	printtokens2	2	9	2.56	0.93
	schedule	2	7	2.86	0.86
	schedule2	2	3	2	1
	tcas	2	17	5.82	0.09
		3	12	5.92	0.11
		4	7	6	0.06
	totinfo	2	5	4.8	1
		3	7	4.86	1
gzip	gzip1	2	6	2.33	1
	gzip2	2	3	2.33	1
	gzip4	2	1	2	1
	gzip5	2	3	2	1

executed all the tests containing c. Then, inducing probability is computed using the formula explained in Section 4.2.1.

Depending on the input parameter model of the program, number of parameters, their domain size and constraints, generating all the tests containing a combination can be a very expensive task. This is the case for the inducing combinations identified for the two programs, replace and grep. Thus, inducing probabilities are not computed for these two programs.

In Table 4-15 the “test strength” column shows the strength of the initial test set, and the next column, i.e., “#of killed versions”, indicates the number of versions killed using the corresponding test set. The last two columns show the average size of the identified inducing combinations and the average of their inducing probabilities.

As shown in Table 4-15, in most cases, the inducing probability is one, which means that the identified inducing combination is truly inducing. For *printtokens2* and *schedule*, the inducing probability is close to one. However, the inducing probability is very low in the tcas program.

Recall that for our experiments, we limit the size of inducing combination to six. BEN reports the top ranked suspicious combination of size six, if the inducing combination of a smaller size was not identified. For *tcas*, the average size of inducing combination is or close to 6, 5.82 and 5.92 (Table 4-15). This shows that BEN does not find the truly inducing combination, but it stops as it reaches the size of 6.

4.5.2.1.2 Phase 2: Faulty statement localization

Table 4-16 shows the results of our approach on each program. We will not explain the column headers one by one, as they are self-explanatory. Note that in the last eight columns, average values are used, since the data could be different in different versions.

Column “Avg size of inducing combination” indicates the average size of inducing combinations for versions that are killed by the t-way test set. For example, the sizes of the inducing combinations for three versions, 3, 5 and 6, of *printtokens* that are killed by the 2-way test set, are 2, 4 and 3, respectively. Therefore, the average size of inducing combinations is 3. As explained in Section 4.3.1, the size of an inducing combination could be greater than the strength of the initial test set.

The next column, “Avg # of tests for identifying inducing combination”, shows the average number of tests generated in the first phase, i.e., inducing combination identification. For *gzip4*, only one version is killed for which no new test is generated in the first phase. This is because BEN could not find new test containing the top suspicious combination in the first iteration.

If a combination c identified in the first phase is not inducing, there is a probability that the core member does not fail. The higher the inducing probability, the more likely that the core member fails. If the inducing probability is 1, the core member will definitely fail.

Table 4-16. Results for single-fault versions

Programs		Test strength (t)	# of tests in t-way test set	# of killed versions	Avg size of inducing combination	Avg # of tests for identifying inducing combination	Avg # of times the core Member does not fail	Avg # of tests executed for generating derived members	Avg # of times derived members selected from initial test set	Avg # of tests instrumented for selecting derived members	Avg #of statements inspected to find actual faults	Avg percentage of statement inspected to locate actual faults
Siemens Suite	printtokens	2	170	3	3	20	0	10	0	11	25.66	13.65
	printtokens2	2	170	9	2.56	16.67	0	10.89	0	11.89	13.55	6.74
	replace	2	193	32	3.66	19.37	0.41	4.16	0	5.16	30.91	12.77
	schedule	2	64	7	2.86	17.14	0.14	6.43	0	7.43	18.71	12.15
	schedule2	2	64	3	2	10	0	4.33	0	5.33	59.67	46.98
	tcas	2	100	17	5.82	32.23	0.94	21.35	0	22.35	14	21.54
		3	405	12	5.91	25	0.92	20.83	0	21.83	14.67	22.57
		4	1434	7	6	20	1	18.57	0	19.57	11.14	17.14
	totinfo	2	30	5	4.8	40	0	11.5	0	12.5	20.8	16.91
		3	156	7	4.86	27.43	0	13.5	0	14.5	11.71	9.52
grep	grep1	2	121	3	3.33	20	0	9	0	10	327.3 3	10.63
	grep3	2	121	4	4	25	0.5	6	0	7	21.25	0.64
	grep4	2	121	2	2	10	0	5	0	6	172.5	5.21
gzip	gzip1	2	17	6	2.33	10.33	0	1.33	0.16	2.5	170.6 7	9.46
	gzip2	2	17	3	2.33	13.33	0	1.33	0	2.33	92.67	4.62
	gzip4	2	17	1	2	0	0	3	1	5	4	0.21
	gzip5	2	17	3	2	6.67	0	0.67	0.33	2	245.6 7	12.33

However, our approach can still apply if the core member does not fail. We select as the core member a failed test that contains the inducing combination from the initial test set. Column “Avg # of times the core member does not fail” shows the average number of such cases. For all the seven versions of *tcas*, when the initial test set is 4-way, the core member is selected from the initial test set. This is consistent with the fact that the inducing probabilities of the identified inducing combinations were very small (Table 4-15).

For each version, we compute the total number of tests in all the derived member sets, i.e., all the tests executed for generating the derived members. The average of this number for all versions is shown in the ninth column, “Avg # of tests executed for generating derived members”. The number includes all the tests, although later some of them are discarded since they do not pass. The maximum value of this column, 21.35, is for the tcas program and 2-way test set. The minimum value, 0, happens for gzip4. Note that the number of tests executed for generating derived members depends on the size of inducing combination, the domain size of inducing components, and also system constraints.

The column, “Avg # of times derived members are selected from initial test set”, shows the number of cases that all the derived member candidates failed, and a derived member is selected from the initial test set.

The column, “Avg # of tests instrumented for selecting derived members”, shows the average number of derived members whose traces are collected. Recall from Section 4.3.2, the tests of a candidate set are instrumented for trace collection. Note that BEN also needs the execution trace of the core member. Therefore the total number of tests instrumented by the coverage tool is the summation of the following three numbers: 1) number of tests executed for generating derived members (column nine of Table 4-16); 2) number of derived members selected from initial test set (column ten of Table 4-16); and 3) one which represents the core member.

The last two columns show the average number and percentage of statements that must be inspected to locate a fault. To compute this number, we include statements that are ranked higher and statements that are ranked at the same rank but appear before the faulty statement, in the order as produced by our approach. We did not perform any dependency analysis, which could reduce the number of statements that must be inspected.

We point out that, the number of executable statements in tcas is 65, less than 100. In this program, when only one statement is needed to inspect, it is 1.54% of executable code. Therefore, for the tcas program the number of statements gives better insight than the percentage of code.

As shown in Table 4-16 our approach works better for the *grep* and *gzip* programs than the Siemens programs, i.e. small programs. The best case happens with *gzip4* where only 0.21% of code must be inspected to locate the fault. The worst case happens with *gzip5* where 12.33% of the code must be inspected. For the Siemens programs, the best and worst cases happen with *printtokens* and *schedule2*, where 6.74% and 46.98% of the code must be inspected, respectively.

4.5.2.2 Results on multiple-fault versions

In this section, we discuss the result of our experiments on the subject programs that have multiple faults.

4.5.2.2.1 Phase 1: Identifying inducing combination

Table 4-17 shows the inducing probabilities for the inducing combinations identified in the first phase. To compute inducing probability, the same procedure used in Section 4.5.2.1 for single fault versions is performed. Again, two programs, *grep* and *replace*, are ignored as it is very expensive to compute inducing probabilities for these programs.

As shown in Table 4-17, the inducing probabilities for all programs are one or close to one, except for the tcas program. In the five faulty versions (four versions killed by 2-way test set and one killed by 3-way) of the *tcas* program, BEN does not find any inducing combination of size of five or less. Therefore, the most suspicious combination whose size is six is reported as an inducing combination.

Table 4-17. Inducing probabilities for multiple-fault versions

Programs		Test strength (t)	# of killed versions	Avg size of inducing combination	Avg of inducing probability of inducing combination
Siemens Suite	printtokens	2	4	2.75	0.95
	printtokens2	2	7	2.14	1
	schedule	2	5	2	0.86
	schedule2	2	2	2	1
	tcas	2	8	5.12	0.33
		3	1	6	0.02
	tot_info	2	9	4.67	1
gzip	gzip1	2	13	2.07	1
	gzip2	2	4	2.25	1
	gzip5	2	4	2	0.84

4.5.2.2.2 Phase 2: Faulty statement localization

The results are summarized in Table 4-18, where the columns are the same as in Table 4-16. The last two columns, “Avg #of statements inspected to find actual faults” and “Avg percentage of statement inspected to locate actual faults”, show respectively the number of statements and percentage of statements that should be inspected to locate the first faulty statement.

Similar to the single-fault versions, BEN works better for *grep* and *gzip*, than for the Siemens programs. For *grep* and *gzip*, the worst case happens in *gzip1*, where 8.00% of executable code must be inspected to locate the fault. However, the worst case for the Siemens programs happens with *schedule2*, where 25.83% of the executable code must be inspected.

The results in Table 4-16 and Table 4-18, suggest that BEN works better when there are multiple faults. For all the programs, BEN is more effective for multiple-fault versions than single-fault versions, except *grep3*, in terms of percentage of code that needs

Table 4-18. Results for multiple-fault versions

Programs		Test strength (t)	# of tests in t-way test set	# of killed versions	Avg size of inducing combination	Avg # of tests for identifying inducing combination	Avg # of times the core member does not fail	Avg # of tests executed for generating derived members	Avg # of times derived members selected from initial test set	Avg # of tests instrumented for selecting derived members	Avg #of statements inspected to find actual faults	Avg percentage of statement inspected to locate actual faults
Siemens Suite	printtokens	2	170	4	2.75	17.5	0	5	0	6	1.25	0.66
	printtokens2	2	170	7	2.14	11.43	0	3.14	0	4.14	1.86	0.92
	replace	2	193	8	2.5	13	0.12	1.87	0	2.87	12.25	5.06
	schedule	2	64	5	2	10	0.2	2.60	0	3.60	8.2	5.32
	schedule2	2	64	2	2	10	0	4	0	5	45.5	25.83
	tcas	2	100	8	5.12	31.75	0.50	14.37	0	15.37	3.62	5.57
		3	405	1	6	22	1	23	0	24	11	16.92
	tot_info	2	30	9	4.67	36.67	0	9.78	0	10.78	8.67	7.05
grep	grep1	2	121	4	2.5	15	15	5.5	0	6.5	107.5	3.49
	grep3	2	121	7	4.29	27.14	27.14	5.43	0	6.43	32.86	1
	grep4	2	121	1	2	10	10	3	0	4	23	0.69
gzip	gzip1	2	17	13	2.07	10	10	0.61	0.38	2	136.46	8.00
	gzip2	2	17	4	2.25	10	10	1.5	0	2.5	51.25	2.55
	gzip5	2	17	4	2	2.5	2.5	0.75	0.75	2.5	100.25	5.03

to be inspected. Moreover, BEN is more efficient for multiple-fault versions than single-fault versions, in terms of the total number of tests generated in phases 1 and 2 and the number of tests instrumented by the coverage tool for multiple-fault versions. This can be explained as follows.

The more faults a program has, the more likely that a test fails. When there are more failed tests in the initial test set, it is likely to have more inducing combinations or the size of inducing combination is smaller. Inducing combination of smaller size is less expensive to identify compare with those of larger size. This is because the smaller the inducing combination is, the fewer times the *identify* algorithm is called to identify the combination. Moreover, the number of candidate sets equals the size of inducing combination. Thus, the smaller the inducing combination is, the fewer derived candidate sets and therefore the fewer tests are generated in the second phase.

4.5.2.3 Comparison with Tarantula and Ochiai

We compared BEN to two spectrum-based approaches, Tarantula and Ochiai, in terms of effectiveness and efficiency. Experiments suggest that Tarantula and Ochiai perform best among other spectrum based approaches [1][17][23]. Recall that effectiveness is measured by the percentage of executable code that must be examined to guide the programmer to the faulty statement, and efficiency is measured by the number of tests executed and the number of tests instrumented to collect the trace.

Since Tarantula and Ochiai do not deal with test generation, we applied them using the initial combinatorial test set.

In Table 4-19 and Table 4-20, we compare the size of the test sets used in Tarantula, Ochiai and BEN for each program. Table 4-19 and Table 4-20 show the information for single-fault versions and multiple-fault versions, respectively. We used average to aggregate the results of all the killed versions for each program. The third column shows the average size of the combinatorial test sets used in the testing stage for each program. The fourth column shows the average number of tests instrumented for coverage collection, for Tarantula and Ochiai. Since every test needs to be traced for Tarantula and Ochiai, columns three and four are equal.

The same information for BEN is shown in the last column. As shown in Table 4-19 and Table 4-20, BEN needs to instrument only a very small number of tests in comparison with the other approaches. However, BEN generates and executes a number of tests (in addition to the initial test set) to identify the inducing combination. This cost is shown in the fifth column of Table 4-19 and Table 4-20, and it equals to the seventh column of Table 4-16

Table 4-19. Efficiency comparison results for single-fault versions

Programs		Avg # of tests executed in the testing stage	Tarantula and Ochiai	BEN	
			Avg # of tests instrumented for coverage collection	Avg # of tests generated and executed in fault localization stage	Avg #tests instrumented for coverage collection
Siemens Suite	print_tokens	170	170	20	11
	print_tokens2	170	170	16.67	11.89
	replace	193	193	19.37	5.16
	schedule	64	64	17.14	7.43
	schedule2	64	64	10	5.33
	tcas	461.05	461.05	27.44	21.64
	tot_info	103.5	103.5	32.67	13.67
grep	grep1	121	121	20	10
	grep3	121	121	25	7
	grep4	121	121	10	6
gzip	gzip1	17	17	10.33	2.5
	gzip2	17	17	13.33	2.33
	gzip4	17	17	2	5
	gzip5	17	17	2	2

Table 4-20. Efficiency comparison results for multiple-fault versions

Programs		Avg # of tests executed in the testing stage	Tarantula and Ochiai	BEN	
			Avg # of tests instrumented for coverage collection	Avg # of tests generated and executed in fault localization stage	Avg #tests instrumented for coverage collection
Siemens Suite	print_tokens	170	170	17.5	6
	print_tokens2	170	170	11.43	4.14
	replace	193	193	13	2.87
	schedule	64	64	10	3.60
	schedule2	64	64	10	5
	tcas	133.89	133.89	30.67	16.33
	tot_info	30	30	36.67	10.78
grep	grep1	121	121	15	6.5
	grep3	121	121	27.14	6.43
	grep4	121	121	10	4
gzip	gzip1	17	17	10	2
	gzip2	17	17	10	2.5
	gzip5	17	17	2.5	2.5

and Table 4-18. So the last two columns show the cost of applying BEN and the fourth column shows the cost of applying Tarantula and Ochiai.

In [17][27], a score is used to compare different fault localization methods. The score is defined based on the percentage of code that must be examined to find the faulty statement. The percentage is based on executable code, i.e., non-executable code is excluded. Table 4-21 and Table 4-22 show the percentage of all the program versions that achieve each score for single fault and multiple-fault versions, respectively. The results of BEN, Tarantula and Ochiai for the Siemens programs are aggregated and shown in the "Siemens Suite" rows, and the results of these three approaches for the *grep* and *gzip* programs are aggregated in their corresponding rows.

For single fault versions (Table 4-21), on the first score, i.e., 99-100%, which means only 1% or less than 1% of code must be inspected to find the first faulty statement, BEN outperforms Tarantula for the Siemens programs and the *grep* program, while both have the same results for the *gzip* program.

BEN achieves a higher score than Ochiai for the Siemens programs and the same score for the *grep* program. However, Ochiai outperforms BEN for the *gzip* program, in terms of the first score of single-fault versions. We analyzed all the versions of the *gzip* program as an effort to explain this phenomenon. The *gzip* program has a very complex input parameter model with eight constraints. As a result, the derived member candidates which are the nearest neighbor of the core member, i.e., have minimum differences from the core member, are not valid in a number of cases, i.e., they do not satisfy all the constraints. In these cases, a passed test is selected from the initial test set as a derived member, as shown in coulmn 10 of Table 4-16. Therefore, BEN could not benefit from the notion of nearest neighbor. In these cases, BEN uses the core member and a single derived member which is not the nearest neighbor to rank the statements, while Ochiai and

Tarantula uses all tests of the initial test set. Thus, BEN is less effective than Ochiai and Tarantula for the *gzip* program.

For multiple-fault versions of all the programs (Table 4-22), the first score with BEN is higher than Tarantula and Ochiai. Moreover, the difference between them is greater for

Table 4-21. Comparison results for single-fault versions

Programs	Approach	Score									
		99-100%	90-99%	80-90%	70-80%	60-70%	50-60%	40-50%	30-40%	20-30%	10-20%
Siemens Suite	BEN	23.53	30.39	22.55	4.90	3.92	9.80	1.96	1.96	0.98	0
	Ochiai	20.59	34.31	14.71	11.76	4.90	5.88	5.88	1.96	0	0
	Tarantula	18.63	33.33	16.67	11.76	3.92	3.92	8.82	0.98	1.96	0
grep	BEN	66.67	11.11	22.22	0	0	0	0	0	0	0
	Ochiai	66.67	11.11	22.22	0	0	0	0	0	0	0
	Tarantula	55.56	11.11	22.22	11.11	0	0	0	0	0	0
gzip	BEN	38.46	38.46	0	23.08	0	0	0	0	0	0
	Ochiai	46.15	38.46	7.69	7.69	0	0	0	0	0	0
	Tarantula	38.46	38.46	0	15.38	0	7.69	0	0	0	0

Table 4-22. Comparison results for multiple-fault versions

Programs	Approach	Score									
		99-100%	90-99%	80-90%	70-80%	60-70%	50-60%	40-50%	30-40%	20-30%	10-20%
Siemens Suite	BEN	38.64	40.91	18.18	0	0	0	0	0	2.27	0
	Ochiai	31.82	52.27	13.64	0	0	0	0	2.27	0	0
	Tarantula	31.82	61.36	4.55	0	0	0	0	0	2.27	0
grep	BEN	91.67	0	8.33	0	0	0	0	0	0	0
	Ochiai	75.00	16.67	8.33	0	0	0	0	0	0	0
	Tarantula	58.33	33.33	8.33	0	0	0	0	0	0	0
gzip	BEN	23.81	57.14	4.76	14.29	0	0	0	0	0	0
	Ochiai	14.29	71.43	0	9.52	4.76	0	0	0	0	0
	Tarantula	19.05	47.62	19.05	0	4.76	9.52	0	0	0	0

the multiple-fault version compared to the single-fault versions. The reason is that BEN first identifies one inducing combination and it is likely that each inducing combination corresponds to one faulty statement. In the second phase, BEN generates a group of tests with one failed test, i.e., the core member, which likely includes one inducing combination and executes only one faulty statement. Therefore, even when there is more than one fault in the program, BEN focuses on one of them. However, when Tarantula and Ochiai are applied on multiple-fault programs, they use the initial test set that likely includes several failed tests corresponding to different faulty statements. Moreover, Tarantula and Ochiai do not perform any nearest neighbor analysis. Thus, it is likely that very different execution traces are compared to each other, which reduces their effectiveness of locating the faulty statement.

Table 4-23 and Table 4-24 also show the comparison between BEN and Tarantula and Ochiai for single-fault and multiple-fault versions, respectively. There are two groups of columns that show the comparison between BEN and Tarantula and the comparison between BEN and Ochiai, respectively.

In each group, the first two columns show cases that BEN outperforms the other approach, Tarantula or Ochiai (positive numbers). The first column shows the number of killed versions that BEN outperforms the other approach, and the next one shows the average percentage of improvement. For example in the 19 out of 36 killed single-fault versions of the tcas program, BEN inspects 7.94% (of executable code) less than Tarantula.

The third column of each group shows the number of killed versions that BEN and the other approach, Tarantula or Ochiai, produce the same results. In addition, the last two columns of each group show the number of versions that the other approach outperforms BEN and the average percentage of the differences (negative numbers). For example in 5 out of 36 killed single-fault versions of the *tcas* program, BEN inspects about 3.38% (of executable code) more than Tarantula.

Three rows, Siemens suite, *grep* and *gzip*, are added to represent the total results of all the Siemens programs, all *grep* and all *gzip* versions, respectively.

Table 4-23. Differences between BEN, Tarantula and Ochiai for single-fault versions

Programs		#of killed versions	Tarantula					Ochiai				
			BEN > Tarantula		BEN = Tarantula	BEN < Tarantula		BEN > Ochiai		BEN = Ochiai	BEN < Ochiai	
			#of versions	Average of Difference Percentages		#of versions	Average of Difference Percentages	#of versions	Average of Difference Percentages		#of versions	Average of Difference Percentages
Siemens Suite	printtokens	3	+1	+8.51	1	-1	-0.53	+1	+4.79	1	-1	-0.53
	printtokens2	9	+3	+5.97	2	-4	-3.98	+1	+9.45	4	-4	-7.21
	replace	32	+14	+8.56	4	-14	-9.80	+14	+8.21	4	-14	-11.16
	schedule	7	+2	+1.30	1	-4	-13.47	+2	+1.30	1	-4	-13.47
	schedule2	3	+2	+5.91	1	0	0	0	0	1	-2	-3.54
	tcas	36	+19	+7.94	12	-5	-3.38	+19	+7.61	12	-5	-3.38
	totinfo	12	+4	+27.85	6	-2	-13.82	+3	+4.07	6	-3	-10.03
Siemens Suite		102	45	9.40	27	-30	-8.40	40	7.21	29	-33	-8.90
grep	grep1	3	+2	+2.84	1	0	0	+1	+1.56	1	-1	-1.92
	grep3	4	+1	+6.80	1	-2	-0.09	+1	+0.24	1	-2	-0.09
	grep4	2	+1	+3.08	0	-1	-0.12	0	0	0	-2	-0.98
grep		9	+4	+3.89	2	-3	-0.10	+2	0.90	2	-5	-0.81
gzip	gzip1	6	+4	+5.41	0	-2	-2.52	+2	+0.26	0	-4	-7.71
	gzip2	3	+3	+7.98	0	0	0	+1	+12.86	0	-2	-0.80
	gzip4	1	0	0	1	0	0	0	0	1	0	0
	gzip5	3	+2	+0.40	0	-1	-24.53	+1	0.05	0	-2	-13.07
gzip		13	+9	+5.15	1	-3	-9.86	+4	3.36	1	-8	-7.32

For single-fault versions (Table 4-23), BEN outperforms Tarantula in all the three cases, Siemens Suite, *grep* and *gzip*, which is consistent with Table 4-21. According to Table 4-23, BEN outperforms Ochiai for the Siemens programs, while Ochiai works better than BEN for the *gzip* and *grep* programs, for single-fault versions. The difference between BEN and Ochiai is very small (less than one percent), and thus it is not reflected in Table 4-21. As explained, Ochiai outperforms BEN for the gzip programs, because BEN could not benefit from the notion of nearest neighbor in this program.

Table 4-24. Differences between BEN, Tarantula and Ochiai for multiple-fault versions

Programs		#of killed versions	Tarantula					Ochiai				
			BEN > Tarantula		BEN = Tarantula	BEN < Tarantula		BEN > Ochiai		BEN = Ochiai	BEN < Ochiai	
			#of versions	Average of Difference Percentages		#of versions	Average of Difference Percentages	#of versions	Average of Difference Percentages		#of versions	Average of Difference Percentages
Siemens Suite	printtokens	4	0	0	3	-1	-0.53	0	0	4	0	0
	printtokens2	7	+6	+3.73	1	0	0	+3	+1.33	4	-2	-0.50
	replace	8	0	0	1	-7	-4.25	+2	+3.10	1	-5	-4.96
	schedule	5	+1	+2.60	1	-3	-5.41	+4	+2.27	0	-1	-9.74
	schedule2	2	+1	+3.94	1	0	0	0	0	1	-1	-0.79
	tcas	9	+1	+4.62	5	-3	-4.62	+2	+2.31	4	-3	-4.10
	totinfo	9	+2	+2.85	3	-4	-11.18	+4	+10.30	3	-3	-1.36
Siemens Suite		44	11	3.57	15	-18	-5.84	15	4.34	17	-15	-3.51
grep	grep1	4	+3	+0.64	1	0	0	+2	+13.97	1	-1	-4.00
	grep3	7	+4	+2.13	0	-3	-0.10	+4	+0.24	0	-3	-0.10
	grep4	1	0	0	0	-1	-0.12	0	0	0	-1	-0.12
grep		12	+7	+1.49	1	-4	-0.10	+6	+4.82	1	-5	-0.89
gzip	gzip1	13	+9	+13.58	0	-4	-7.17	+10	+6.38	0	-3	-9.32
	gzip2	4	+4	+0.70	0	0	0	+1	+0.10	0	-3	-0.55
	gzip5	4	+3	+1.18	0	-1	-2.76	+2	+0.40	0	-2	-2.21
gzip		21	+16	+8.03	0	-5	-6.29	+13	4.98	0	-8	-4.25

For multiple-fault versions (Table 4-24), BEN outperforms Ochiai for all the three cases, Siemens Suite, *grep* and *gzip*, although the difference between the two approaches is very small for the Siemens programs. In Siemens programs, Tarantula is more effective than BEN; however, BEN is much more effective in the *grep* and *gzip* programs.

We investigated all the four versions of *totinfo* in which Tarantula outperforms BEN. In all cases the faulty statement localized by BEN is different from the one localized by Tarantula. The faulty statement, which is detected by Tarantula, is not even executed by the core member generated by BEN, thus it is not suspicious. The same situation happens for two out of three versions of the tcas program that Tarantula outperforms BEN (Table 4-24).

The effectiveness of BEN could be different in localizing different faulty statements. However, as we mentioned, BEN focuses on one inducing combination, which is likely due to one faulty statement. While there may be more than one inducing combination, BEN stops searching for inducing combinations, as soon as the first one is identified, in the first phase. The effectiveness of BEN to some extent depends on the faulty statement related to the identified inducing combination.

4.5.2.4 Threats to validity

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. One of the key steps in our experiments is modeling the input parameters, which may affect the correctness of the result. To reduce this threat, we have modeled the input parameters by using the program specifications and if they are not available, the error-free versions, without having any knowledge about the faults. All the models, except the *gzip* model, have been used in other studies [13][10]. In [10], the models are used to compare the effectiveness of combinatorial testing and random testing.

In addition, we automated the experimental procedure as much as possible, as an effort to remove human errors. In particular, all the steps are automated except counting the number of statements that should be inspected to find the faulty statement. Further, consistency of the results has been carefully checked to detect potential mistakes made in the experiments. For example, the higher the average of inducing probability, the more likely the core member fails. In the extreme case, if the inducing probability is 1, the core member must fail. To check the consistency of the results, we check the inducing probability whenever the core member did not fail. For instance, in one out of seven killed versions of the *schedule* program, the core member did not fail. We checked the inducing probability for this version, which is relatively small, 0.25.

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from the Siemens suite [7]; these programs are created by a third party and have been used in other studies [17][27][23]. However, the subject programs are programs of relatively small size with seeded faults. To mitigate this threat, the *grep* and *gzip* programs were added to the experiments, but more experiments on larger programs with real faults can further reduce this threat.

Each of the Siemens program has multiple versions, each of which has a single fault. However, programs in practice could have multiple faults. To mitigate this threat, we created several multiple-fault versions that combined randomly selected faults and conducted an experiment on these versions. More experiments on programs with real faults can further reduce this threat.

4.6 RELATED WORK

In this section, we first discuss existing work on identifying failure-inducing combination, i.e., the first phase of BEN. Then, we focus on fault localization work, which is related to the second phase of BEN.

4.6.1 Related work on identifying inducing combinations

Existing approaches to identifying inducing combinations can be classified into two groups. The first group takes as input a single failed test and tries to identify inducing combinations in the test.

Two techniques, called FIC and FIC_BS [37], try to identify all the inducing combinations contained in a failed test. These approaches take one failed test from a combinatorial test set, then generate and execute a small number of tests in a systematic manner to identify inducing combinations in the failed test. New tests are generated such that one value, v_i, of the failed test is changed to another possible value. When the newly generated test passes, v_i is part of inducing combination because its removal makes the test pass. FIC generates k tests; where k is the number of parameters, for each failure inducing combination.

FIC_BS is the binary search version of FIC. To generate a new test, FIC_BS changes the values of k/2 parameters of the failed test. If the newly generated test passes, FIC_BS searches for inducing combination in the changed values (k/2). The process continues until all inducing combinations are found. FIC and FIC_BS assume that no new inducing combinations are introduced when a value is changed to create a new test.

Li et al. [22] introduced two techniques for identifying inducing combinations called RI and SRI. These techniques use a method called delta debugging [36] in an iterative framework. The RI approach takes one failed test from the initial combinatorial test set, and adopts a similar approach to FIC_BS to generate a small number of tests. The SRI approach, which is an improved version of RI, takes one failed test, f, and the combinatorial test set. Then it tries to find a similar passed test to f from the combinatorial test set. SRI uses the fact that the inducing combination appeared in the failed test f, but not in the similar passed test. Therefore, it focuses on the parameters, which are different in the failed and passed tests. SRI could identify inducing combination by generating fewer tests than RI.

The second group of techniques for identifying inducing combinations takes a set of tests as well as their execution statuses.

The AIFL technique in [30][33] first identifies a set A of suspicious combinations as candidates for being inducing. Second, it generates a group of tests for each failed test using SOFOT strategy [26]. After executing the newly generated tests, combinations which appeared in the passed tests are removed from the suspicious set, A.

The InterAIFL technique is an iterative approach proposed by Wang et al. in [33]. It iteratively generates and refines suspicious set A until it becomes stable.

Let k be the number of parameters. For each test f, the SOFOT strategy generates k tests by changing the value of one parameter at a time. Each test is different from the original test f in one value; the value is selected randomly from the corresponding parameter’s domain.

BEN also, tries to identify inducing combinations in a combinatorial test set, instead of a single failed test. There are two advantages resulting from using the whole test set rather than a single test. First, a test set contains more information than a single test. Second, it would be possible to identify inducing combinations that appear in different tests.

BEN identifies suspicious combinations in the same way as AIFL and Inter-AIFL. However, BEN produces a ranking of suspicious combinations and focuses on the most suspicious combinations. Moreover, BEN significantly differs from AIFL and Inter-AIFL in the way of generating new tests. BEN generates tests for a top-ranked suspicious combinations based on the notions of suspiciousness combination and suspiciousness of the environment. While AIFL and Inter-AIFL generate tests for failed tests and select values randomly.

We mention that Yilmaz et al. proposed a machine learning approach to identify failure-inducing combinations [35]. The approach analyzes the combinatorial test set and tests statuses and builds a classification tree. The classification tree is used to predict inducing combinations. Shakya et al. in [28] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

4.6.2 Related work on fault localization

In Section 4.5, we already mention two fault localization approaches, Tarantula [17][18] and Ochiai [1]. Similar to BEN, Tarantula and Ochiai use statement coverage information to compute suspiciousness of each statement. Statement coverage is computed by multiple execution traces of failed and passed tests.

In Tarantula, the suspiciousness value of each statement is the ratio of failed tests that execute the statement divided by the ratio of failed tests that execute the statement plus the ratio of passed tests that execute the statement. However, Ochiai computes the suspiciousness value of each statement by dividing the number of failed tests that execute the statement by the square root of all failed tests multiply by all tests that execute the statement.

Then, Tarantula and Ochiai look for faulty statement in a non-increasing order of their suspiciousness values.

Three spectrum-based approaches, set union, set intersection and nearest neighbor, are proposed by Renieris and Reiss in [27]. These approaches assume that there are one failed run (the spectrum of a failed test) and a large number of passed runs (the spectra of passed tests).

Each of the three approaches has a different way to identify highly suspicious statements for being faulty, and these statements are then checked to find the actual faults. Let f be the program spectrum of a failing run and S be a set of program spectra of passed runs. The set union method computes $\mathrm{f} - \bigcup_S \mathrm{s}$, where $\bigcup_S \mathrm{s}$ is the union spectra of a set of passed runs. The statements in the spectrum of the failed run but not in the union spectra of the passed runs are highly suspicious. In the intersection method, the highly suspicious statements are in the intersection spectra of a set of passed runs but not in the spectrum of the failed run, $\bigcap_S \mathrm{s} - \mathrm{f}$.

In the nearest neighbor approach, one passed run whose spectrum is the most similar to the failed spectrum is selected from S. The statements in the difference set of these two spectra have the highest suspiciousness of being faulty.

If the fault is not found in the highly suspicious statements set, the program dependence graph is build. The nodes corresponding to the highly suspicious statements are marked as blamed nodes. Then, in both directions, backward and forward, a breadth-first search is performed from the blamed nodes. The statements corresponding to the nodes at a distance of one are also suspicious and must be checked. This process is repeated until the faulty statement is found.

Empirical evaluation in [17] shows that for the Siemens suite, Tarantula is more effective and efficient than the other methods, including set union, set intersection, and nearest neighbor. Lucia et al. in [23] reported the experiments that show Tarantula and Ochiai are comparable to each other for the Siemens programs. However, the work reported in [1] suggests that Ochiai outperforms Tarantula. The former work used statement coverage spectra while the latter used branch coverage spectra. Both works, i.e., [1] and [23], applied fault localization methods using the test pools provided for each program by the benchmark [7].

Our experimental results also show that Ochiai is slightly better than Tarantula. BEN used combinatorial test set and statement coverage spectra.

The fundamental difference between BEN and the above spectrum-based approaches is that BEN systematically generates a small group of tests, and then analyzes their spectra to produce a ranking of statements. The existing approaches do not deal with test generation. Instead, they assume the existence of a large number of test runs, which are generated randomly or using other techniques. In addition, they require every test execution to be traced. As a result, they cannot utilize the testing results if the test executions were not traced. In contrast, our approach is designed to work after normal testing is performed where test executions are not traced. Our approach only needs to trace the execution of a small number of tests that are generated in the second phase of our approach. As shown in Section 4.5, our approach can significantly reduce the number of tests needed to be instrumented for tracing but still produce results that are competitive to or better than Tarantula and Ochiai.

We mention that an approach, called LCEC [24], was reported that also leverages the result of combinatorial testing to localize the faulty statement. LCEC was published after our original work in [13][14]. LCEC selects a failed test from the initial combinatorial test set, and generates a group of passed tests by changing values of failed test involved in the inducing combination. The execution traces of failed and passed tests are analyzed to derive cause-effect chains of statements. A depth-first search is performed for all cause-effect chains to locate faulty statement. Then, if the faulty statement is not found, the user does breath-first search in the dynamic backward slice, which has been done in, associated with the incorrect output value. LCEC is applied to four small programs, maximum 220 lines of code, including tcas. The cost of applying LCEC is not reported in [24].

4.7 CONCLUSION

In this paper, we presented an approach called BEN to localizing faults that leverages the result of combinatorial testing. Our approach consists of two phases. The first phase identifies a failure-inducing combination, which is used in the second phase to localize the faulty statement in the source code.

In the first phase, BEN adopts an iterative framework that ranks suspicious combinations and generates new tests repeatedly until an inducing combination is identified. The novelty of this phase lies in the fact that we rank suspicious combinations and generate new tests based on the notions of suspiciousness of a combination and suspiciousness of its environment. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. New tests are generated for a user-specified number of top-ranked suspicious combinations such that the suspiciousness of the environment of a combination is minimized in each test. Our approach starts with searching for inducing combinations whose size is equal to the strength t of the initial test set. If it is not found, the approach expands its search to combinations whose size is greater than t.

The key idea of the second phase of BEN is that we systematically generate a group of tests from an inducing combination such that the spectra of these tests can be analyzed quickly to identify the faulty statement. This group of tests consists of a core member that is a failed test run and a number of derived members that are passed test runs but are very similar to the core member. The suspiciousness values of statements are computed by analyzing the spectra of the core member and the derived members.

We applied BEN to the Siemens suite and also the *grep* and *gzip* programs. Our experimental results show that our approach requires a very small number of tests to be generated while significantly reducing the number of statements to be inspected for fault localization. In particular, our approach achieves results that are competitive to or better than those of Tarantula [18] and Ochiai [1] while requiring significantly fewer tests to be instrumented.

We emphasize that our approach has an important advantage over existing spectrum-based approaches such as Tarantula and Ochiai. Existing spectrum-based approaches require every test execution to be traced. If a test set is already executed without being traced, the test set must be re-executed to collect traces before they can be used by approaches like Tarantula an Ochiai. In contrast, our approach only requires a small number of tests generated in the second phase of our approach to be traced. Our approach is designed to work after normal testing is performed where test executions do not need to be traced.

We plan to conduct more empirical studies to further evaluate the performance of our approach. In particular, our current approach assumes that a combinatorial test set is used to test a program. We plan to investigate how to adapt our approach to work with an arbitrary test set. This will further increase the applicability of our approach. That is, we will try to identify inducing combinations from an arbitrary test set and then use them to generate tests for fault localization. The challenge is to deal with the fact that unlike a combinatorial test set, an arbitrary test set does not guarantee that all t-way combinations are covered. This might reduce the effectiveness of our approach.

4.8 Acknowledgment

This work is partly supported by three grants (70NANB12H175, 70NANB10H168, and 70NANB15H199) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

4.9 REFERENCES

1. R. Abreu, P. Zoeteweij, and A. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization", In 12th Pacific Rim International Symposium on Dependable Computing, pp.39,46, Dec. 2006.
2. Advanced Combinatorial Testing System (ACTS), http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html, 2015.
3. BEN: a combinatorial tesing-based fault localization tool, http://barbie.uta.edu/~laleh/BEN.html, 2015.
4. M. N. Borazjany, Y. Linbin, Y. Lei, R. Kacker, and D. R. Kuhn, “T-way testing of ACTS: A Case Study”, In Proceedings of the IEEE fifth International Conference on Software Testing, Verification and Validation, pp.591-600, 2012.
5. D. Cohen, S. Dalal, M. Fredman, and G. Patton. “The AETG system: An approach to testing based on combinatorial design”, In Proceedings of the IEEE Transactions on Software Engineering, 23(7):437–444, 1997.
6. M. B. Cohen, P. B. Gibbons, W.B. Mugridge, C.J. Colbourn. “Constructing test suites for interaction testing”, In Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pages 38-48, 2003.
7. 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.
8. Empirical study on combinatorial testing, http://barbie.uta.edu/~laleh/research.html, 2015.
9. GCC online documentation, https://gcc.gnu.org/onlinedocs, 2015.

10. L. Ghandehari, M. N. Borazjany, Yu Lei, Raghu Kacker, Richard Kuhn, "Applying Combinatorial Testing to the Siemens Suite", In IEEE International Conference on Software Testing, Verification and Validation (ICSTW), 2013.

11. L. Ghandehari, J. Czerwonka, Y. Lei; S. Shafiee, R. Kacker, R. Kuhn, "An Empirical Comparison of Combinatorial and Random Testing," In Proceedings of the Software Testing, Verification and Validation Workshops (ICSTW), pp.68-77, 2014.

12. L. Ghandehari, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker, Richard Kuhn, "BEN: A Combinatorial Testing-Based Fault Localization Tool", In IEEE International Conference on Software Testing, Verification and Validation (ICSTW), Graz, Austria, April, 2015.

13. L. Ghandehari, Y. Lei, D. Kung, R. Kacker, R, Kuhn. Fault localization based on failure-inducing combinations. Proceeding of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 168-177, 2013.

14. L. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. "Identifying Failure-Inducing Combinations in a Combinatorial Test Set", Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), 370-379, 2012.

15. GNU Grep, http://www.gnu.org/software/grep/manual/grep.html, 2015.

16. GNU Gzip, http://www.gnu.org/software/gzip/manual/gzip.html, 2015.

17. J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique", In Proceeding IEEE/ACM Automated software engineering, 2005, 273-282.

18. J. Jones, M. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization", In Proceedings of International Conf. on Software Engineering, 2002, 467-477.

19. D. R. Kuhn and V. Okum. “Pseudo-Exhaustive Testing for Software”, In Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW '06). IEEE Computer Society, 2006, 153-158.

20. D.R. Kuhn, D.R. Wallace, A.M. Gallo. “Software fault interactions and implications for software testing”, In Proceedings of the IEEE Transaction on Software Engineering, 2004, 30: 418–421.

21. Y. Lei, R. Kacker, D. Kuhn, V. Okun, J. Lawrence, “IPOG/IPOD: Efficient test generation for multi-way software testing”, *Journal of Software Testing, Verification, and Reliability*, 18(3):125-148, Sept. 2008.

22. J. Li; C. Nie, and Y. Lei, "Improved Delta Debugging Based on Combinatorial Testing," In Proceedings of International Conference on Quality Software (QSIC), pp.102,105, 2012.

23. Lucia, D. Lo, L. Jiang, A. Budi, “Comprehensive evaluation of association measures for fault localization”, In Proceedings of the IEEE International Conference on Software Maintenance, 1-10, 2010.

24. C. Ma, Y. Zhang, J. Liu, and M. Zhao, "Locating Faulty Code Using Failure-Causing Input Combinations in Combinatorial Testing," In Proceedings of 4th World Congress on of Software Engineering (WCSE), pp.91,98, 2013.

25. C. Nie and H. Leung. “A survey of combinatorial testing”, ACM Computing Surveys (CSUR), 43(2):11: 1-11: 29, January 2011.

26. C. Nie, H. Leung, and B. Xu. “The minimal failure-causing schema of combinatorial testing”, ACM Transactions on Software Engineering and Methodology, Volume 20 Issue 4, September 2011.

27. M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries”, In Proceedings of the International Conference on Automated Software Engineering, 2003.

28. K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn, "Isolating Failure-Inducing Combinations in Combinatorial Testing Using Test Augmentation and Classification," In proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), pp.620-623, 2012.

29. P.J. Schroeder, P. Bolaki, V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," In Proceeding of the International Symposium on Empirical Software Engineering, pp.49-59, 2004.

30. L. Shi, C. Nie, B. Xu. “A software debugging method based on pairwise testing”, In Proceedings of the International Conference on Computational Science (ICCS2005), pages 1088-1091, 2005.

31. Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php, 2012.

32. D. R. Wallace, D. R. Kuhn, “Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data”, In Proceeding of the ACS/ IEEE International Conference on Computer Systems and Applications, pp. 301-311, 2001.

33. Z. Wang, B. Xu, L. Chen, and L. Xu. “Adaptive interaction fault location based on combinatorial testing”, In Proceedings of the 10th International Conference on Quality Software (QSIC 2010), pages 495–502, 2010.

34. E. Wong and V. Debroy, “A survey on software fault localization”, Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, Nov. 2009.

35. C. Yilmaz, M. B. Cohen, A. A. Porter. “Covering arrays for efficient fault characterization in complex configuration spaces”, In Proceedings of the IEEE Transaction on Software Engineering, 2006, 32(1): 20-34.

36. A. Zeller and R. Hildebrandt. “Simplifying and isolating failure-inducing input”, In Proceedings of the IEEE Transactions on Software Engineering, 2002, pages 183–200.

37. Z. Zhang, and J. Zhang. “Characterizing failure-causing parameter interactions by adaptive testing”, In Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA 2011), pages 331-341, 2011.

Chapter 5. BEN: A Combinatorial Testing-Based Fault Localization Tool

The paper is published in IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2015.

BEN: A Combinatorial Testing-Based Fault Localization Tool[*]

Laleh Sh. Ghandehari[1], Jaganmohan Chandrasekaran[1], Yu Lei[1], Raghu Kacker[2], D. Richard Kuhn[2]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA

{laleh.shikhgholamhosseing, jaganmohan.chandrasekaran}@mavs.uta.edu, ylei@cse.uta.edu

[2]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD, USA

{raghu.kacker, d.kuhn}@nist.gov

Abstract- We present a combinatorial testing-based fault localization tool called BEN. BEN takes as input three types of information, including the subject program, the source code, an input parameter model, and a combinatorial test set created based on the input parameter model. It is assumed that the combinatorial test set has already been executed, and thus the execution status of each test is known. The output of BEN is a ranking of statements in terms of their likelihood to be faulty. In the fault localization process, a small number of additional tests are generated by BEN and need to be executed by the user. In this paper, we present the major user scenarios and the high-level design of BEN. BEN is implemented in Java and provides a graphical user interface that provides friendly access to the tool.

Keywords- BEN, Fault Localization, Combinatorial Testing.

[*] Copyright © 2015 IEEE. Reprinted, with permission, from Laleh Sh. Ghandehari, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker and Richard Kuhn, BEN: A Combinatorial Testing-Based Fault Localization Tool, IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), April 2015.

5.1 INTRODUCTION

In this paper, we introduce a combinatorial testing-based fault localization tool called BEN. BEN takes as input three types of information about the subject program, the source code, an input parameter model, and a combinatorial test set created based on the input parameter model. It is assumed that the combinatorial test set has already been executed, and thus the execution status of each test is known. The output of BEN is a ranking of statements such that the higher rank a statement has, the more likely it is faulty.

The fault localization process conducted by BEN consists of two major phases. The first phase produces a ranking of combinations in terms of their likelihood to be failure-inducing. A combination is failure-inducing, or simply inducing, if all tests containing this combination fail [3, 5, 7, 12, 13]. In the second phase, BEN takes a top ranked inducing combination from which a failed test and a small number of passed tests are generated. The execution traces of these tests are analyzed to produce the final ranking of faulty statements.

BEN is written in Java and thus can be executed on different platforms such as Windows, Linux and MacOS. BEN provides both Graphical User Interface (GUI) and Command Line Interface. BEN is developed with support from NIST and the University of Texas at Arlington. BEN is publicly available [1].

Several approaches are reported on how to identify inducing combinations in a combinatorial test set [5, 8, 12, 14]. Ma et al. reported an approach that identifies faulty code based on failure-inducing combinations [6]. While they adopt a similar two-phase framework, they use very different techniques to identify inducing combinations and faulty statements. To the best of our knowledge, their work is the only other work that performs code-based fault localization based on combinatorial testing. However, they did not provide a public tool that implements their approach.

The remainder of this paper is organized as follows. Section 5.2 describes the main idea of the fault localization approach implemented by BEN. Section 5.3 discusses how to use BEN through a use case. Section 5.4 describes the design of BEN in terms of major data structures and modules. Section 5.5 provides concluding remarks and our plan for future work.

5.2 APPROACH

In this section we provide a high-level discussion about the fault localization approach implemented by BEN, in terms of its two major phases, i.e., *inducing combination identification*, and *faulty statement identification*. Refer to our earlier work [5, 4] for more details.

5.2.1 Inducing combination identification

This phase adopts an iterative framework. It begins by analyzing the initial combinatorial test set to identify the set of all suspicious combinations. A suspicious combination with respect to a test set F is a combination that only appears in the failed tests of F. Suspicious combinations are candidates of inducing combinations. Suspicious combinations are ranked based on their likelihood to be inducing. Next, a set of new tests is generated that the user may choose to execute. The results of these new tests are used to refine the ranking of suspicious combinations. This process continues until one or more stopping conditions are satisfied.

The ranking of suspicious combination is based on two key concepts, suspiciousness of combinations and suspiciousness of the environment of combination. Informally, the higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher the combination is ranked. The details of the ranking and test generation methods were explained in [5].

5.2.2 Faulty statement identification

In this phase, a small group of tests are generated using an inducing combination. The group has one failed test, which is referred to as a core member, containing the inducing combination. The group also has several passed tests which are referred to as derived members. Derived members are very similar to the core member but do not contain the inducing combination. The execution trace of the core member is compared to the execution trace of each derived member to produce a ranking of statements in terms of their likelihood of being faulty.

5.3 USE CASE

In this section, we describe how BEN works. We use the 26th faulty version of the *replace* program from the Siemens suite [2]. In order to apply combinatorial testing, we first modeled the input parameters for the *replace* program. The details of the model are discussed in [3]. Then, ACTS [11] is used to generate a 2-way test set consisting of 190 tests. All these 190 tests are executed, and 47 of them are failed. We show how to use BEN to locate faulty statements causing these failures.

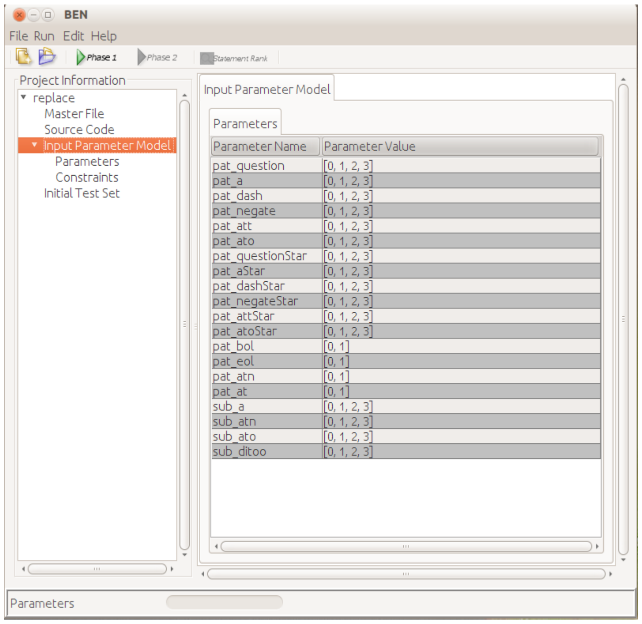
A new project, "*replace*", is created in BEN by providing two input files. The first file is the source code of the *replace* program. The second file is a configuration file consisting of the input parameter model, the 2-way test set, and the test results. Figure 5-1 shows part of the second input file.

Figure 5-2 shows the main window of BEN after creating the replace project. The main window has two parts. The left part provides an outline of the project where project components are organized into a tree structure. The right part provides the details of each component selected in the tree structure.

By pressing "Phase 1" button in the toolbar, BEN starts the first phase, i.e., inducing combination identification. The first phase may contain multiple iterations. When

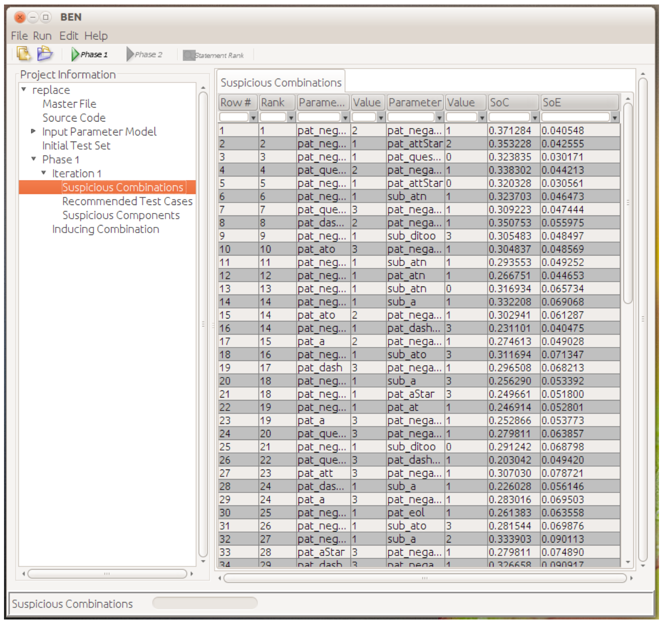
```
Parameters:
"pat_question:[0, 1, 2, 3]"
"pat_a:[0, 1, 2, 3]"
"pat_dash:[0, 1, 2, 3]"
"pat_negate:[0, 1, 2, 3]"
    .
    .
    .
Relations :
"[2,(pat_question, pat_a, pat_dash, pat_negate, . . .)]"

Constraints :
sub_a =1 => sub_atn != 1 && sub_ato != 1 && sub_ditoo != 1
sub_atn =1 => sub_a != 1 && sub_ato != 1 && sub_ditoo != 1
sub_ato =1 => sub_atn != 1 && sub_a != 1 && sub_ditoo != 1
    .
    .
    .
Tests
pat_question,pat_a,pat_dash,pat_negate, . . .
0,0,0,0,0,0,0,2,1,3,0,0,0,0,0,0,3,1,0,0
1,0,2,2,2,2,2,0,3,2,2,2,0,0,0,0,1,3,2,2
2,0,1,3,2,0,2,2,2,0,2,0,0,0,0,0,0,2,1,3
    .
    .
    .

Results
Test #,result,
0 , fail
1 , pass
2 , pass
    .
    .
    .
```
Figure 5-1. The example of input file

the first iteration completes, new nodes are added to the tree to show the results of the first iteration, including suspicious combinations, recommended test cases, suspicious components, and inducing combinations. The user can inspect each component by clicking on the desired node. Figure 5-3 shows the suspicious combinations and their ranks at the end of the first iteration.



Parameter Name	Parameter Value
pat_question	[0, 1, 2, 3]
pat_a	[0, 1, 2, 3]
pat_dash	[0, 1, 2, 3]
pat_negate	[0, 1, 2, 3]
pat_att	[0, 1, 2, 3]
pat_ato	[0, 1, 2, 3]
pat_questionStar	[0, 1, 2, 3]
pat_aStar	[0, 1, 2, 3]
pat_dashStar	[0, 1, 2, 3]
pat_negateStar	[0, 1, 2, 3]
pat_attStar	[0, 1, 2, 3]
pat_atoStar	[0, 1, 2, 3]
pat_bol	[0, 1]
pat_eol	[0, 1]
pat_atn	[0, 1]
pat_at	[0, 1]
sub_a	[0, 1, 2, 3]
sub_atn	[0, 1, 2, 3]
sub_ato	[0, 1, 2, 3]
sub_ditoo	[0, 1, 2, 3]

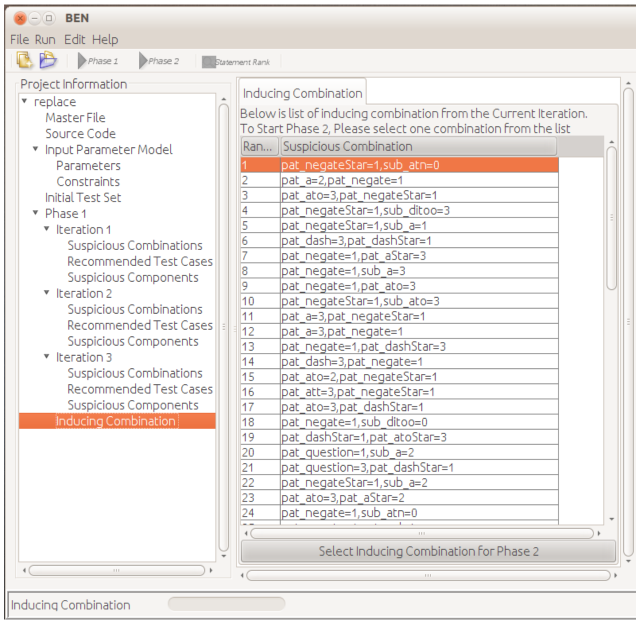
Figure 5-2. The main window of BEN after creating a project



Suspicious Combinations

Row #	Rank	Parame...	Value	Parameter	Value	SoC	SoE
1	1	pat_neg...	2	pat_nega...	1	0.371284	0.040548
2	2	pat_neg...	1	pat_attStar	2	0.353228	0.042555
3	3	pat_neg...	1	pat_ques...	0	0.323835	0.030171
4	4	pat_que...	2	pat_nega...	1	0.338302	0.044213
5	5	pat_neg...	1	pat_attStar	0	0.320328	0.030561
6	6	pat_neg...	1	sub_atn	1	0.323703	0.046473
7	7	pat_que...	3	pat_nega...	1	0.309223	0.047444
8	8	pat_das...	2	pat_nega...	1	0.350753	0.055975
9	9	pat_neg...	1	sub_ditoo	3	0.305483	0.048497
10	10	pat_ato	3	pat_nega...	1	0.304837	0.048569
11	11	pat_neg...	1	sub_atn	1	0.293553	0.049252
12	12	pat_neg...	1	pat_atn	1	0.266751	0.044653
13	13	pat_neg...	1	sub_atn	0	0.316934	0.065734
14	14	pat_neg...	1	sub_a	1	0.332208	0.069068
15	14	pat_ato	2	pat_nega...	1	0.302941	0.061287
16	14	pat_neg...	1	pat_dash...	3	0.231101	0.040475
17	15	pat_a	2	pat_nega...	1	0.274613	0.049028
18	16	pat_neg...	1	sub_ato	3	0.311694	0.071347
19	17	pat_dash	3	pat_nega...	1	0.296508	0.068213
20	18	pat_neg...	1	sub_a	3	0.256290	0.053392
21	18	pat_neg...	1	pat_aStar	3	0.249661	0.051800
22	19	pat_neg...	1	pat_at	1	0.246914	0.052801
23	19	pat_a	3	pat_nega...	1	0.252866	0.053773
24	20	pat_que...	3	pat_nega...	1	0.279811	0.063857
25	21	pat_neg...	1	sub_ditoo	0	0.291242	0.068798
26	22	pat_que...	3	pat_dash...	1	0.203042	0.049420
27	23	pat_att	3	pat_nega...	1	0.307030	0.078721
28	24	pat_das...	1	sub_a	1	0.226028	0.056146
29	24	pat_a	3	pat_nega...	1	0.283016	0.069503
30	25	pat_neg...	1	pat_eol	1	0.261383	0.063558
31	26	pat_neg...	1	sub_ato	3	0.281544	0.069876
32	27	pat_neg...	1	sub_a	2	0.333903	0.090113
33	28	pat_aStar	3	pat_nega...	1	0.279811	0.074890
34	29	pat_dash	3	pat_nega	1	0.326658	0.090917

Figure 5-3. Suspicious combinations after first iteration



Below is list of inducing combination from the Current Iteration.
To Start Phase 2, Please select one combination from the list

Ran...	Suspicious Combination
1	pat_negateStar=1,sub_atn=0
2	pat_a=2,pat_negate=1
3	pat_ato=3,pat_negateStar=1
4	pat_negateStar=1,sub_ditoo=3
5	pat_negateStar=1,sub_a=1
6	pat_dash=3,pat_dashStar=1
7	pat_negate=1,pat_aStar=3
8	pat_negate=1,sub_a=3
9	pat_negate=1,pat_ato=3
10	pat_negateStar=1,sub_ato=3
11	pat_a=3,pat_negateStar=1
12	pat_a=3,pat_negate=1
13	pat_negate=1,pat_dashStar=3
14	pat_dash=3,pat_negate=1
15	pat_ato=2,pat_negateStar=1
16	pat_att=3,pat_negateStar=1
17	pat_ato=3,pat_dashStar=1
18	pat_negate=1,sub_ditoo=0
19	pat_dashStar=1,pat_atoStar=3
20	pat_question=1,sub_a=2
21	pat_question=3,pat_dashStar=1
22	pat_negateStar=1,sub_a=2
23	pat_ato=3,pat_aStar=2
24	pat_negate=1,sub_atn=0

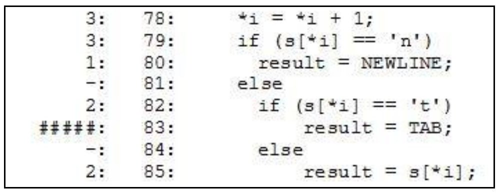
Figure 5-4. List of inducing combinations

At this point, the user can either choose to proceed to the next iteration of Phase 1 or begin Phase 2. In the first case, the user can press “Phase 1” button again. In the second case, the user should select an inducing combination and start Phase 2 by clicking on the “Phase 2” button. Figure 5-4 shows inducing combinations of the example.

When Phase 2 completes, new nodes are added to the tree, which shows the core member and three derived members. These tests should be executed and then the user should provide the execution traces for the core member and all the derived members. BEN adopts the Gcov [10] format for execution trace. The Gcov is a coverage tool and a standard utility with the GNU Compiler Collection (GCC) suite.

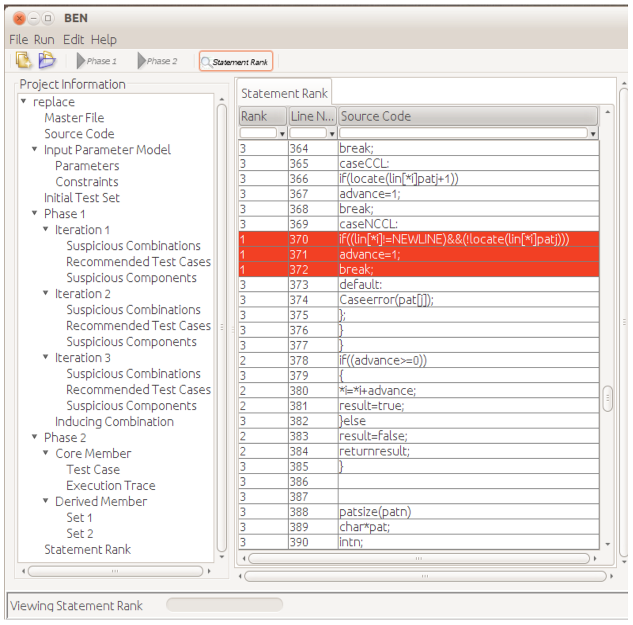
Note that other coverage tools could be used, but the output should be saved in the Gcov format. Figure 5-5 shows part of an execution trace of *replace* in the Gcov format. The details of the Gcov format are available in [10].

By clicking on the “Statements Rank” button, the statements are ranked and top ranked statements are highlighted, as shown in Figure 5-6. The demo of the example is available at [1]. BEN is applied on several programs, Siemens suite, grep and gzip [2], and the results show that it could locate faulty statement effectively and efficently. The result will be published in fututre.



```
        3:   78:      *i = *i + 1;
        3:   79:      if (s[*i] == 'n')
        1:   80:        result = NEWLINE;
        -:   81:      else
        2:   82:        if (s[*i] == 't')
    #####:   83:            result = TAB;
        -:   84:        else
        2:   85:            result = s[*i];
```

Figure 5-5. Part of an execution trace in Gcov format



Rank	Line N...	Source Code
3	364	break;
3	365	caseCCL:
3	366	if(locate(lin[*i]patj+1))
3	367	advance=1;
3	368	break;
3	369	caseNCCL:
1	370	if((lin[*i]!=NEWLINE)&&(!locate(lin[*i]patj)))
1	371	advance=1;
1	372	break;
3	373	default:
3	374	Caseerror(pat[j]);
3	375	};
3	376	}
3	377	}
2	378	if((advance>=0))
3	379	{
2	380	*i=*i+advance;
2	381	result=true;
3	382	}else
2	383	result=false;
2	384	returnresult;
3	385	}
3	386	
3	387	
3	388	patsize(patn)
3	389	char*pat;
3	390	intn;

Figure 5-6. Statements ranking

5.4 DESIGN

Figure 5-7 shows the architecture of BEN consisting of two layers. The logical layer contains the core functional components that manage the fault localization process. The data layer contains the core data structures that store the input, intermediate and final output.

5.4.1 Data Layer

IPM: A class that represents the input parameter model consisting of parameters and constraints. A parameter contains a name and a list of values it takes. BEN supports the same data types as supported by ACTS, i.e., Boolean, Integer and Enum [11]. A constraint expression is represented as a string.

Component: A class that represents a parameter value. Each parameter value may be associated with a suspiciousness value.

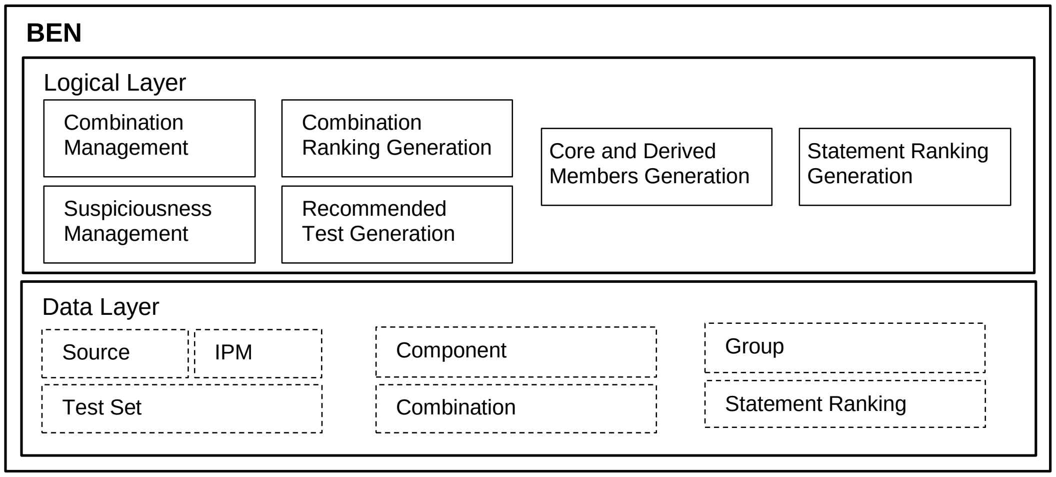

BEN
Logical Layer
Combination Management
Combination Ranking Generation
Core and Derived Members Generation
Statement Ranking Generation
Suspiciousness Management
Recommended Test Generation
Data Layer
Source
IPM
Component
Group
Test Set
Combination
Statement Ranking

Figure 5-7. Architecture diagram

Combination: A class that represents a combination of parameter values. Each combination may be associated with a suspiciousness value.

Test Set: A class that represents a test set, which includes an array of tests and also a list of execution results, one for each test. This class is used to represent both the initial test set and recommended test sets.

Group: A class that represents a group of tests, one of which is a core member and the others are derived members. An execution trace may be associated with each test in the group.

5.4.2 Logical Layer

Combination Management: A module is responsible to generate all possible combinations. This model is also responsible for checking validity of combination.

Suspiciousness Management: A module that compute different types of suspiciousness including suspiciousness of component, combination, and environment.

Combination Rank Generation: A module that computes the ranking of suspicious combinations, using suspiciousness values computed by Suspiciousness Management.

Recommended Test Generation: A module that generates recommended tests. Recommended tests are guaranteed to be new, i.e. they have not been executed before. The module also integrates an open source constraint solver, Choco [9] for constraint handling to ensure validity of tests.

Core and Derived Member Generation: A module that generates the core member and derived member based on an inducing combination. This module also uses the constraint solver, Choco, to make sure the core and derived members satisfy constraints.

Statement Ranking Generation: A module that analyzes the execution traces of the core member and derived members and produces the ranking of statements in terms of their likelihood of being faulty.

5.5 CONCLUSION

In this paper, we report a combinatorial testing-based fault localization tool, i.e., BEN. We present a use case to demonstrate how to use BEN and also the architectural design of BEN. Currently, BEN only implements our own approach to fault localization. We plan to define and make public an API that allows BEN to be used by other combinatorial testing-based approaches, e.g., Inter-AIFL [12] and FIC_BS [14].

5.6 Acknowledgment

This work is partly supported by a research grant (70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

Disclaimer: NIST does not endorse or recommend any commercial product referenced in this paper or imply that a referenced product is necessarily the best available for the purpose.

5.7 REFRENCES

1. BEN the combinatorial testing-based fault localization tool, http://barbie.uta.edu/~laleh/BEN.html
2. 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.
3. L.S. Ghandehari, M.N. Borazjany, Y. Lei, R. Kacker, R. Kuhn, Applying Combinatorial Testing to the Siemens Suite. Proceeding of IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 362-371, 2013.
4. L.S. Ghandehari, Y. Lei, D. Kung, R. Kacker, R, Kuhn. Fault localization based on failure-inducing combinations. Proceeding of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 168-177, 2013.
5. L.S. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. Identifying Failure-Inducing Combinations in a Combinatorial Test Set. Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), 370-379, 2012.
6. C. Ma, Y. Zhang; J. Liu; M. Zhao, Locating Faulty Code Using Failure-Causing Input Combinations in Combinatorial Testing, Proceedings of Fourth World Congress on Software Engineering (WCSE), 91-98, 2013.
7. C. Nie and H. Leung. The Minimal Failure-Causing Schema of Combinatorial Testing. ACM Transactions on Software Engineering and Methodology (TOSEM), 20(4):15 , 2011.

8. L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. Proceedings of the 5th International Conference on Computational Science (ICCS), 1088-1091, 2005.

9. The Choco Constraint Solver, http://www.emn.fr/z-info/choco-solver/index.html

10. The Test Coverage Tool, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

11. L. Yu, Y. Lei, R.N. Kacker, D.R. Kuhn. ACTS: A Combinatorial Test Generation Tool. Proceeding of IEEE International Conference on Software Testing, Verification and Validation (ICST), 370-375, 2013.

12. Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. Proceedings of International Conference on Quality Software (QSIC), 495–502, 2010.

13. Zeller and R. Hildebrandt. Simplifying and isolating failure inducing input. Proceedings of the IEEE Transactions on Software Engineering, 183-200, 2002.

14. Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA), 331-341, 2011.

Chapter 6. Applying Combinatorial Testing to the Siemens Suite

The paper is published in IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2015.

Applying Combinatorial Testing to the Siemens Suite[*]

Laleh Sh. Ghandehari[1], Mehra N. Borazjany[1], Yu Lei[1], Raghu N. Kacker[2], D. Richard Kuhn[2]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas 76019, USA

[2]Information Technology Laboratory National Institute of Standards and Technology, Gaithersburg, Maryland 20899, USA

Abstract- Combinatorial testing has attracted a lot of attention from both industry and academia. A number of reports suggest that combinatorial testing can be effective for practical applications. However, there are few systematic, controlled studies on the effectiveness of combinatorial testing. In particular, input parameter modeling is a key step in the combinatorial testing process. But most studies do not report the details of the modeling process. In this paper, we report an experiment that applies combinatorial testing to the Siemens suite. The Siemens suite has been used as a benchmark to evaluate the effectiveness of many testing techniques. Each program in the suite has a number of faulty versions. The effectiveness of combinatorial testing is measured in terms of the number of faulty versions that are detected. The experimental results show that combinatorial testing is effective in terms of detecting most of the faulty versions with a small number of tests. In addition, we report the details of our modeling process, which we hope to shed some lights on this critical, yet often ignored step, in the combinatorial testing process.

[*] Copyright © 2013 IEEE. Reprinted, with permission, from Laleh Sh. Ghandehari, Mehra N. Borazjany, Yu Lei, Raghu N. Kacker and D. Richard Kuhn, Applying Combinatorial Testing to the Siemens Suite, IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), March 2013.

Keywords- Combinatorial Testing, Input Modeling, Software Testing.

6.1 INTRODUCTION

Combinatorial testing has attracted a lot of attention from researchers. The key observation in combinatorial testing is that most software failures are caused by interactions of only a few input parameters. A t-way combinatorial test set is built to cover all the t-way interactions, where t is typically a small integer [11][6]. If test parameters and values are properly modeled, a t-way test set is able to expose all failures that involve no more than t parameters.

A number of empirical reports suggest that combinatorial testing can be effective for practical applications [2][3][7]. Most studies in these reports were designed to show that combinatorial testing could be applied to different types of applications. Thus, they were not controlled studies for evaluating the effectiveness of combinatorial testing. There are two notable exceptions. Kuhn et al. studied several fault databases and found that all the faults in these databases are caused by interaction of no more than six parameters [9][10].These studies did not perform actual combinatorial testing on the subject systems. Schroeder et al. compared the effectiveness of t-way testing to random testing in a controlled study [14]. They selected two software applications used in their laboratory as subject programs, and manually seeded a number of faults to measure fault detection effectiveness.

In this paper, we report an experiment that applies combinatorial testing to the Siemens suite [17]. The Siemens suite has been used as a benchmark to evaluate the effectiveness of many testing techniques [3][7][18]. Each program in the suite has a number of faulty versions. The effectiveness of combinatorial testing is measured in terms of the number of faulty versions that are detected. The results show that most of the faulty versions are detected by a small number of test cases. For example, all 32 faulty versions of replace program are detected by a 2-way test set containing only 192 tests. Furthermore, the results show that combinatorial testing is more effective than random testing.

We also report the details of our modeling process, which is a critical, yet often ignored step in the combinatorial testing process. Our approach consists of three main steps. First we create an abstract model for the system. This model consists of abstract parameters and values. On the one hand, abstraction reduces the modeling complexity that has to be managed at one time. On the other hand, abstraction helps to discover aspects that need to be tested. Second we generate a combinatorial test set based on the abstract model. Existing combinatorial test generation tools such as ACTS [1] can be used in this step. Third, we derive concrete tests from the abstract tests. These concrete tests are then used to perform the actual testing.

It is important to note that whereas the programs in the Siemens suite are relatively small, in terms of lines of code, and have a small number of input parameters, their input spaces are complex. For example, replace has 564 lines of code and 3 input parameters. However, its abstract model contains 20 abstract parameters and 36 constraints. The input parameters have different features and characteristics that must be considered for testing, e.g. one of the input parameters is a regular expression.

The remainder of this paper is organized as follows. In section 6.2, we describe our approach for applying combinatorial testing. Section 6.3 reports experimental results that demonstrate the effectiveness of our modeling. Section 6.4 discusses existing work on input space modeling. Section 6.5 provides concluding remarks.

6.2 APPROACH

In this section, we explain our approach to apply combinatorial testing. The approach consists of three major steps: (1) Create an abstract model, (2) Generate an abstract test set, and (3) Derive concrete tests. We use the *replace* program in the Siemens suite, to explain each task in detail.

6.2.1 Create abstract model

This step has two major tasks: (1) define abstract parameters and values, (2) define relations and constraints.

6.2.1.1 Define abstract parameters and values

First, we analyze the system specification and identify factors that may affect the behavior of the system. These factors are candidates for abstract parameters. The equivalence partitioning approach is used to define the values of each abstract parameter.

We use the *replace* program in the Siemens suite to show how we define abstract parameters and values based on its specification. The *replace* program has three inputs, *pattern*, *substitute* and *input text*. The program finds every match of the *pattern* in the *input text* and replaces it with the *substitute*.

The *pattern* is a restricted form of regular expression. Table 6-1 shows the metacharacters that can be used in *pattern*. Note that the @ character can have different meanings, depending on the next character. If a character other than *n* and *t* appears after @, the program ignores it. For example, @*e* matches *e*. But when @ appears at the end of the *pattern*, the program behave as if it is a simple character and matches with @. For example, *e@* matches *e@*.

The *substitute* is a string that allows only three metacharacters to be used. These include two metacharacters, @*t* and @*n*, as shown in Table 6-1 and a metacharacter &, which represents the string that matches the *pattern*. For example, if the string that matches the *pattern* is *ab* and the substitute is *a&c*, all *ab* strings in the file are replaced with *aabc*.

Table 6-1. Pattern's metacharacter

Metacharacter	Description
?	Matches every character.
*	Matches the preceding pattern element zero or more times.
[-]	Matches a single character that is in the specified range. For example [a-c] matches “a”, “b” and “c”.
[^]	Matches every character except the ones inside brackets.
@t	Matches a tab.
@n	Matches the end of a line.
%	Matches the beginning of a line. (BOL)
$	Matches the end of a line. (EOL)

Table 6-2 shows the abstract model of the *replace* program for *pattern* and *substitute*. There are a total of 20 parameters in the model. The parameters with prefix *pat* are identified for pattern, and the parameters with prefix *sub* are identified for *substitute*. Note that these parameters are abstract as they are not the actual input parameters taken by the *replace* program.

The key modeling decision is twofold. First, each metacharacter is identified to be an abstract parameter. Our motivation is that the core logic of the *replace* program is dealing with these metacharacters. Thus, we consider each metacharacter to be an important factor that could affect the program behavior. Special attention is paid to metacharacters * and &. These two metacharacters can be combined with other meta or regular characters. An abstract parameter is identified for each possible combination. For example, *pat_question** represents the combination where a question mark appear before *.

Second, the values of each abstract parameter (i.e., metacharacter) are identified based two considerations. The first consideration is whether or not a parameter appears in the pattern (or substitute). Two values, *off* and *on*, can be used to represent the two cases.

Table 6-2. The Abstract Model of Replace

Parameters	Values
pat_character[1]	[off, begin, middle, end]
pat_question[2]	[off, begin, middle, end]
pat_range[3]	[off, begin, middle, end]
pat_negate[4]	[off, begin, middle, end]
pat_@t	[off, begin, middle, end]
pat_@character	[off, begin, middle, end]
pat_question*	[off, begin, middle, end]
pat_character*	[off, begin, middle, end]
pat_range*	[off, begin, middle, end]
pat_negate*	[off, begin, middle, end]
pat_@t*	[off, begin, middle, end]
pat_@character*	[off, begin, middle, end]
pat_BOL[5]	[off,on]
pat_EOL[6]	[off,on]
pat_@n	[off,on]
pat_@	[off,on]
sub_character	[off, begin, middle, end]
sub_@n	[off, begin, middle, end]
sub_@character	[off, begin, middle, end]
sub_&	[off, begin, middle, end]

[1]Regular character
[2]? metacharacter
[3][-] metacharacter
[4][^] metacharacter
[5]% metacharacter
[6]$ metacharacter

The second consideration is the following: If a parameter does appear in the pattern (or substitute), where does it appear? Thus, the *on* value identified earlier is further divided into three abstract values, *begin*, *middle*, and *end*. In Table II, all the parameters but four have four values, *off*, *begin*, *middle*, and *end*. The four exceptions, i.e., *pat_BOL*, *pat_EOL*, *pat_@n*, and *pat_@*, only have two values, *on* and *off*, because they can only appear in a particular position by nature. For example, BOL (i.e., %) by definition can only appear in the beginning of the pattern.

Now we discuss how to model the third input parameter, i.e., the *input text*, of the *replace* program. We consider that an *input text* consists of a sequence of lines. The key observation is that a line is relevant from the testing perspective only if it contains a match or mismatch of the *pattern*. Assume that the *pattern* consists of *k* elements. The *input text* is modeled such that it consists of k + 2 lines. The first line matches the *pattern*. The second line matches all the elements but the first in the *pattern*. The third line matches all the elements but the second in the *pattern*, and so on. The last line does not match any element in the *pattern*. Note that we do not consider cases where a mismatch is due to multiple, but not all, of the elements in the *pattern*. This is essentially a trade-off made between test effort and test coverage.

6.2.1.2 Define relations and constraints

Relations are used to create parameter groups that can be covered at different strengths. Furthermore, parameters in different groups are independent and thus their combinations do not have to be tested. In our experiments, we used the default relation where all the parameters are considered to be in the same group. In retrospect, the parameters for *pattern* could be put into one group and the parameters for *substitute* in a second group. This would allow us to reduce the number of tests.

Constraints are used to exclude combinations that are not valid from the domain semantics. For the *replace* program, a total of 36 constraints are specified. All these 36 constraints are concerned with the position values of different parameters. In particular, in each test, there shall be only one parameter that has the value *begin* or *end*.

6.2.2 Generate abstract tests

In this step, an abstract test set is generated using an existing combinatorial test generation tool [11]. We used the ACTS tool [1]. ACTS can generate a combinatorial test

a. Abstract test

Parameters	Values
pat_character	middle
pat_question	middle
pat_range	middle
pat_negate	middle
pat_@t	off
pat_@character	off
pat_question*	off
pat_character*	off
pat_range*	off
pat_negate*	off
pat_@t*	off
pat_@character*	off
pat_BOL	on
pat_EOL	off
pat_@n	on
pat_@	off
sub_character	begin
sub_@n	end
sub_@character	off
sub_&	middle

b. Concrete test

Parameters	Values
Pattern	%a?[a-e][^a]@n
Substitute	a&@n
Input file	1. abef 2. gabef 3. bef 4. aef 5. abf 6. abe 7. abefg 8. gbfag

Figure 6-1. An example of abstract test and its concrete test

set with strength 2 through 6. Note that these tests are abstract in that they cannot be directly executed. Instead, concrete tests must be derived first, which is discussed below.

6.2.3 Derive concrete tests

A scheme is needed to derive a concrete test from each abstract test. Conceptually, such a scheme consists of two parts. The first part is to map each abstract value to a concrete value. An abstract value is typically identified in a way such that it represents an equivalence group, i.e., a group of values that are equivalent to each other in terms of how they could affect the system behavior. Thus, it is sufficient to map an abstract value to any value in its equivalence group. For example, in the replace program, the abstract value, middle, represents all the positions those are neither at the beginning nor at the end. The specific position is often not important.

The second step is to map an abstract test to a concrete test. This part builds on the first step. In addition, it needs to map abstract parameters to concrete parameters. Recall that abstract parameters are identified to represent factors that could affect the system behavior. There typically does not exist a one-to-one mapping between abstract and concrete parameters. In fact, there are often more abstract parameters than concrete parameters. For example, for the replace program, there exist 20 abstract parameters, which need to be mapped to three concrete input parameters.

As an example, consider the abstract test in Figure 6-1(a) and the concrete test in Figure 6-1 (b) for the *replace* program. In this example, the value of *pat_BOL* is *on*, so "%" is put at the beginning of the pattern. Similar, "@n" is placed at the end of the pattern. Other parameters, whose values are *middle*, are placed in the middle of the pattern. For *pat_character*, *pat_range* and *pat_negate* a, [a-e] and [^a] are put in pattern. Similarly, the *substitute* is created based on the corresponding parameter values in the abstract test.

The last row of Figure 6-1 (b) shows different lines in the input file. The first line, *abef*, matches the pattern, since *a* matches with *a*, *b* matches with question mark, *e* matches with [*a-e*] element, and *f* matches with [^*a*]. Also, the first line matches *%* at the beginning and @*n* at the end.

Each line from line 2 to 7 matches all but one element in the *pattern*. For example the second line has the exact string *abef* which matches the pattern. However, since it is not at the beginning of the line (i.e., there is *g* at the beginning), the first element, *%*, in the pattern is not matched. The third line violates *a* in the pattern, and so on. The last line, i.e., line 8, does not match any element in the pattern.

Note that the scheme used to derive concrete tests from abstract tests is often specific to the subject application. However, such a scheme typically can be fully automated. This is the case for our experiments, where we wrote a program for each subject program to automate this process.

6.3 EXPERIMENT

We used the Siemens suite as our subject programs [17]. The Siemens suite contains 7 programs and each of these programs contains a number of faulty versions. The Siemens suite also provides an error-free version and a test pool for each program.

Table 6-3 represents properties of subject programs. The second column shows the number of lines of uncommented code. The third column shows the number of procedures. The forth column shows the number of faulty versions for each program.

Two programs, *printtokens* and *printtokens2*, have the same specification but different implementations. Since the input space model is independent from the source code, these programs share the same model. Similarly, two programs *schedule* and *schedule2* have the same specification and thus share the same model. Therefore, in this section, we present five input models for the Siemens suite programs. Note that the input model for *tcas* is given in [9] and is included here for completeness.

Table 6-3. Subject Programs

Program	LOC	Procedures	#Faulty Versions
print_tokens	726	20	7
print_tokens2	570	21	10
replace	564	21	32
schedule	412	18	9
schedule2	374	16	10
tcas	173	8	41
totinfo	565	16	23

In our experiments, we focus on interaction faults. As a result, our models are not designed for boundary testing or invalid testing. We believe most boundary and invalid faults are one-way faults, and they can be detected more efficiently using a different model where the focus is to identify special values of individual parameters. However, this belief needs to be validated by more experiments, which is beyond the scope of this paper.

Specifications of the programs are not provided by the benchmark. To understand what each program is supposed to do, we had to inspect the source code. (A search on the Internet did not find any such specification either.) To avoid potential bias in developing the model, only the source code of the error-free version was used. That is, we were not aware of the faults during the modeling process. Nonetheless, this is an internal threat to validity that needs to be considered.

We start with 2-way testing, and then move to 3-way testing, and so on, until (1) all faulty versions are detected; or (2) testing at the current strength does not detect any faulty versions that were not detected in testing at the previous strength. For example, 2-way testing did not detect 2 out of 9 faulty versions of the *schedule* program. So 3-way testing was performed on these 2 versions, which did not detect any of the two versions. At this point, we stopped testing and started to inspect the testing results.

6.3.1 Replace

We explained the modeling details of the *replace* program in the previous section. We applied 2-way testing to this program, which had a total of 192 tests. We detected all the 32 faulty versions of this program.

6.3.2 Schedule

Two programs, *schedule* and *schedule2*, take the following inputs: (1) three non-negative integers representing the number of processes in three different priority queues, *low*, *medium* and *high*; and (2) a list of commands that must be done on queues. The output of these two programs is a list of numbers indicating the order in which the processes exit (from the scheduling system).

For example, consider the first three input parameters which are 3, 2 and 1. Three processes are placed in low priority queue, two processes in medium priority queue, and one process is high priority queue. The id is assigned to the processes by their priority so the 0 is in the high priority queue, 1 and 2 are in medium priority queue and 3, 4 and 5 are in low priority queue.

There are seven commands (1) *new job*: this command has one attribute, *queue*, and adds a new process at the specified priority queue. (2) *upgrade_prio*: it has two attributes, *queue* and *ratio*. This command promotes a process form the specified priority queue to the next higher priority queue. The *ratio* attribute is used to determine which process to be promoted. (3) *block*: this command adds the current process to the blocked queue. (4) *unblock*: this command unblocks a process from the blocked queue. It has one attribute, *ratio*, which is used to determine which process must be unblocked. (5) *quantum_expire*: this command puts the current process at the end of its priority queue. (6) *finish*: this command exits the current process and prints its number. (7) *flush*: this command causes all processes from the priority queues to exit in their priority order.

Two commands, *upgrade_prio* and *unblock*, operate on the *n*-th process where $\mathrm{n} = (\mathrm{int})\ (\mathrm{r} + 1)$ and $\mathrm{r} = (\text{length of queue} * \text{ratio})$.

In our previous example, if a *flush* command (7) is executed, the output is 0 1 2 3 4 5. But, assume that before the *flush* command, a *new job* command (1 3) is executed, where 1 indicates the new job command and 3 indicates the high priority queue. This *new job* command adds a process to the high priority queue. The next available ID, which is 6, is assigned to the new process and the process is placed at the end of the high priority

Table 6-4. The Abstract Model of Schedule

Parameters	Values
new_process	[0, 1, >1]
new_proc_queue	[low, mid, high]
upgrade_prio	[0, 1, >1]
upgrade_queue	[low, mid]
upgrade_ratio	[0, 1, >1, {r}=0.1, {r}=0.4, {r}=0.5, {r}=0.6, {r}=0.9]
block	[0, 1, >1]
unblock	[0, 1, >1]
unblock_ratio	[0, 1, >1, {r}=0.1, {r}=0.4, {r}=0.5, {r}=0.6, {r}=0.9]
quantum_expire	[0, 1, >1]
finish	[0, 1, >1]
flush	[0, 1, >1]

queue, i.e. after process 0. Now, if we execute the flush command, the output will be 0 6 1 2 3 4 5.

Table 6-4 shows the input model of the two *schedule* programs. Commands and their attributes are modeled as parameters. Each command parameter has three values, 0, 1 and >1, where 0 means that this command does not appear, 1 means that this command appears once, and >1 means that this command appears more than once. The *priority* attribute of the *new job* command could be one of the three possible queues. But the attribute of *upgrade_prio* could be either low or mid. (Processes in the high priority queue cannot upgrade.)

Two commands *unblock* and *upgrade_prio* are affected by the length of the queues, they select a process based on queue's length and ratio. For these commands, first, we test if the ratio equals to 0, 1, or >1. Then we check that if the number after floating point in $\mathrm{r} = (\text{length of queue} * \text{ratio})$ is 1, 4, 5, 6 or 9. These numbers are selected to cover upper limit (9), lower limit (1) and middle of the range (5), and also two numbers (4 and 6) around the middle.

A C++ program was written to create the file that contains commands based on abstract tests. For the initial length of the queues, we randomly selected 60. We fixed >1 values to 2, i.e. if the value of a command is >1, the command appears twice in the file.

Performing 2-way testing detected 7 out of 9 versions of the *schedule* and 3 out of 10 versions of the *shedule2*. In total, 9 versions were not detected. Performing 3-way testing did not detect any more versions. We investigated all versions that were not detected, 8 out of 9 (version 9 of the *schedule* and 7 versions, 1, 4, 5, 6, 8, 9 and 10, of the *schedule2*) can be detected by invalid testing, which as mentioned is not the focus of our study.

For example, version 10 of the *schedule2* was detected by a test case which contains *new_process* or *upgrade_prio* commands with invalid value for the queue attribute (*new_proc_queue* or *upgrade_queue* parameter).

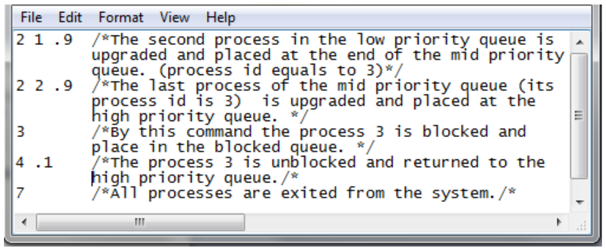
Version 8 of the *schedule* is the only version that was not detected and could not be detected by invalid testing. This version could be detected only when two *upgrade* commands, one *block* command, and one *unblock* command are executed consecutively on one process.

The following example will reveal the bug:

./schedule 2 2 0 <file.txt

There are 4 processes, 0 to 3, two of which, 0 and 1, are in the *mid* priority queue, and the other two, 2 and 3, are in the low priority queue. The high priority queue is empty. Figure 6-2 shows the file that contains 5 commands. The comments explain the state of the system after each command is executed.

In the *schedule* program, each process keeps the id of the queue to which it belongs. The faulty code in the version 8 does not change the queue id of the process after



```
2 1 .9   /*The second process in the low priority queue is
         upgraded and placed at the end of the mid priority
         queue. (process id equals to 3)*/
2 2 .9   /*The last process of the mid priority queue (its
         process id is 3)  is upgraded and placed at the
         high priority queue.  */
3        /*By this command the process 3 is blocked and
         place in the blocked queue.  */
4 .1     /*The process 3 is unblocked and returned to the
         high priority queue./*
7        /*All processes are exited from the system./*
```

Figure 6-2. File example to detect v8 of schedule

the *upgrade* command (lines #1 and #2). Thus when the process is unblocked (line #4), it is assigned to the wrong queue.

We did not detect this version, because our approach, at this point, does not generate test sequences. Combinatorial test sequence generation is a subject that we plan to study in the future.

6.3.3 Tcas

This program was previously modeled by Kuhn et al. in [9][10], based on the specification in [12]. The *tcas* program is an aircraft collision avoidance system, and it takes 12 numbers as input and generates as output one number, which can be 0, 1 and 2.

Table 6-5 shows the input model of the tcas program. Some input parameters, e.g., *high_confidence*, *two_of_three_reports_valid*, and c*limb_inhibit*, are boolean values, 0 and 1. Some input parameters, like *alt_layer_value*, are of enum type and have a set of specific values. For the other parameters, the values are identified by analyzing the code and by equivalence partitioning. Note that the input space of this program is not complex, and thus an abstract model is not needed.

According to [9] all 41 faulty versions of *tcas* are detected by the model. The maximum strength to detect all versions is six; we also got the same results.

As discussed in Section 6.3.6, all faulty versions of the tcas program were detected by 6-way testing. However, the degree of fault is actually more than 6 in all faulty versions.

Table 6-5. The abstract model of *tcas*

Parameters	Values
cur_vertical_sep	[299,300, 601]
high_confidence	[0, 1]
two_of_three_reports_valid	[0, 1]
own_tracked_alt	[1, 2]
own_tracked_alt_rate	[600, 601]
other_tracked_alt	[1, 2]
alt_layer_value	[0,1, 2, 3]
up_separation	[0, 399, 400, 499, 500, 639, 640, 739, 740, 840]
down_separation	[0, 399, 400, 499, 500, 639, 640, 739, 740, 840]
other_rac	[0, 1, 2]
other_capability	[1, 2]
climb_inhibit	[0, 1]

Thus, these faulty versions were actually detected by higher strength combinations that happen to appear in a 6-way testing.

6.3.4 Totinfo

This program takes as input a file containing one or more tables. The program uses the notions of chi-square and degree of freedom to calculate whether the distribution of the numbers in these tables is logarithm-gamma distribution. The output is the total degree of freedom of rows and columns and chi-square.

We focused on the correctness of the syntax of input parameters instead of the mathematical aspect of the program. The reason is that the logic of the program is very complex and is difficult to understand due to a lack of specification.

We identified a total of 6 parameters related to the syntax input of the program. Parameter *# of tables* can be 0, 1 or more than one. The maximum number of members in a table is 1000. We set the maximum number of rows and columns to 500 and the minimum number of rows and columns to 1. Thus, parameters *# of rows* and *# of columns* have three values, 1, between 2 and 499, and 500.

Table 6-6. The abstract model of *totinfo*

Parameters	Values
#of tables	[0, 1, >1]
#of rows	[1, between 2 and 499, 500]
#of columns	[1, between 2 and 499, 500]
tbl_attr	[sufficient number positive[1], sufficient number negative[2], sufficient number mix[3], sufficient number equal 0[4], more than enough[5], less than enough[6]]
options	[normal, row & column in 2 lines, comment at the beginning, comment in the middle, comment at the end]
maxline	[1, Between 2and 254, 255, 256, 257]

[1]There are $\#of\ rows \times \#\ of\ columns$ positive numbers in the input file.
[2]There are $\#of\ rows \times \#\ of\ columns$ negative numbers in the input file.
[3]There are $\#of\ rows \times \#\ of\ columns$ positive and negative number in the input file.
[4]There are $\#of\ rows \times \#\ of\ columns$ zero in the input file.
[5]There are less than $\#of\ rows \times \#\ of\ columns$ numbers in the input file.
[6]There are more than $\#of\ rows \times \#\ of\ columns$ numbers in the input file.

Parameter *tbl_attr* is identified to define general attributes for tables' elements. One important attribute for the table elements is sign, they can be *positive*, *negative*, *zero*, or *mix.* The number of elements is another attribute we identified for *tbl_attr.* The number of the elements in a table defined by $\#of\ rows \times \#\ of\ columns$; we added *sufficient*, *more than* and *less than enough* values to check that whether the number of elements in the table is consistent with $\#of\ rows \times \#\ of\ columns$.

The *option* parameter models the position in which a comment appears. The *maxline* parameter defines the maximum number of lines in the input file.

A program was written to generate the input tables from the abstract tests. 2-way testing detected 5 out of 23 versions. 3-way testing detected 7 more versions, but 4-way testing did not detect any new version. So, totally 12 out of 23 versions were detected. We investigated the 11 versions which were not detected by the model. All of these versions have faults related to the mathematical aspects of the program, which is out of our testing scope.

6.3.5 Printtokens Model

The goal of the two programs, *printtokens* and *printtokens2*, is tokenizing the input file and determining the type of each token. Token could have one of these types: *identifier*, *special*, *keyword*, *number*, *comment*, *character constan*t or *string constant*.

Keyword type includes *and*, *or*, *if*, *xor*, and *lambda*. Special type includes *lparen*, *rparen*, *lsquare*, *rsquare*, *quote*, *bquote*, *comma* and *equalgreater*. Comment is started with semicolon and ended when a new line character is seen. String constant is confined in two double quotations. Character is a token started with #.

To model the system, we divided it into seven subsystems: keyword, special, identifier, number, comment, character and string. By this classification each token type was tested independently from the others. We assumed that the program analyzes each token independent from previous and next token, i.e. the type of the previous or next token does not affect on the analyzing the current token.

Each subsystem has 3 parameters, value, position and number of lines. Keyword model is shown in Table 6-7, as an example. The *kyw_value* parameter covers all possible values for keyword (corresponding token type in general). An important property for each token type is position, depends on different position of token type the program may behave differently. So for each token type the position property with three values, *begin, middle* and *end*, is added to the model. The last parameter, *# of lines*, checks the behavior of the system when the input file has a single line or multiple lines.

The possible values for some token types, such as keyword and special are explicitly defined in the program specification. But for the others such as identifier, the features and characteristic of its values are described in the specification. For each token type, identifier, number, comment, character and string, we designed an abstract model to define their values. Then after the possible values were defined in the next level they have

Table 6-7. The abstract model of Keyword

Parameters	**Values**
kyw_value	[and, or, xor, if, lambda]
position	[begin, middle, end]
# of lines	[1, >1]

Table 6-8. The abstract model of Identifier Values

Parameters	Values
lowercase	[off, on]
uppercase	[off, on]
number	[off, on]
keyword	[off, on]
whitespace	[Space, tab]

the same model as keyword. We explain the model of values for three subsystems identifier, number and comment in more details.

Identifier has different feature such as having uppercase, lowercase, keyword or numbers, a model is designed to cover all features of identifier values (Table 6-8).These features are parameters with two values *off* and *on*, to show weather an identifier contains the parameter or not. The *whitespace* parameter determines whether an identifier separate from next token by *space* or *tab*. Note that we add a constraint to prevent having null identifier. For 2-way test generation, we generate 2-way test set for identifier values model first. The number of tests is 7. Then, we put these seven tests as values in the *value* parameter of the identifier model, and generate 2-way test set for identifier.

For the number model, the characteristics of the number are the number of digit and having zero at the beginning of it. So its model has 2 parameters, Table 6-9. Note that sign and decimal point do not support by the *printtokes* programs.

Table 6-9. The abstract model of Number Values

Parameters	Values
#of digits	[1, >1]
begins with zero	[off, on]

Table 6-10. The abstract model of Comment Values

Parameters	Values
identifier	[off, on]
keyword	[off, on]
character	[off, on]
string	[off, on]
special	[off, on]
number	[off, on]
comment	[off, on]
whitespace	[Space, tab]

The comment model is shown in Table 6-10. We check the behavior of the system when each token type appears as a comment. Also, *whitespace* parameter determines if a comment separate from next token by *space* or *tab*, what would be the behavior of the system. The models of sting and character values are the same as comment.

The 2-way testing detected 2 out of seven versions of the *printtokens* and nine versions out of 10 versions of the *printtokens2*. Note that 2-way test set has only 141 tests.

The programs were tested by 3-way testing, but no new version was detected. So we stopped testing and investigated versions which were not detected. Five versions out of six can be detected by invalid testing. For example, in versions 6 of the *printtokens*, the failure happen when the number of tokens in the input file exceeds the defined value. The second version of the *printtokens* is not detected by invalid testing. The fault in this version is adding code. The adding code is reached when there is a i token in the input file.

Table 6-11. Fault classification of detected versions

Program	#faulty versions with degree of fault							
	1	*2*	*3*	*4*	*5*	*6*	*Beyond 6*	*sum*
print_tokens	0	0	2	0	0	0	0	2
print_tokens2	0	6	3	0	0	0	0	9
replace	10	7	2	0	0	0	13	32
schedule	0	5	1	1	0	0	0	7
schedule2	0	3	0	0	0	0	0	3
tcas	0	0	0	0	0	0	41	41
totinfo	0	0	2	1	6	3	0	12

6.3.6 Discussion

After testing programs using the combinatorial technique, we investigated the faults detected by our model to ensure that the fault is caused by the interaction between input parameters. In order to do that we introduce the notion of degree of fault or fault strength which is defined to be the minimum number of parameters that must be involved to trigger the fault.

As a t-way test set contains all t-way combinations, it is guaranteed to detect a faulty version if the strength of the fault does not exceed t. But it is also possible that a t-way test set detects a version whose degree of fault is higher than t. This is because the test set may contain the inducing combination (in which more than t parameters are involved) by chance.

In Table 6-11, we classified the degree of fault for all detected versions. For example, in the *schedule* program, the model detected a total of 7 versions. The fault strength in five of these versions is 2. In the two remaining versions, one of them is 3 and another one is 4.

To define the degree of fault, we used the concept of inducing combination. An inducing combination is a combination of parameter values such that all test cases containing this combination fail. The length of the minimum inducing combination shows the degree of fault.

We used a tool called BEN [4] to find minimum inducing combinations. BEN takes a t-way test set as input and generates a ranking of t-way combinations based on their likelihood to be inducing combinations. BEN has been shown very effective in identifying inducing combinations [4]. However, BEN is heuristic by nature and thus does not guarantee to always find minimum inducing combinations. This should be taken into account when reading the results in Table 6-11. We are not aware of any method that can precisely determine the degree of a fault.

For example seven versions of *schedule* are detected by 2-way test sets. BEN finds an inducing combination for five of them, so the degree of fault is 2 for these versions. For the two other versions BEN did not find an inducing combination, we used a 3-way test set. BEN finds an inducing combination for one of them. We then used a 4-way test set for the last version, which found an inducing combination.

Since there is a probability that the fault is not due to any parameter interaction, we need to check whether only one parameter is involved in the fault. BEN has a feature to derive inducing combinations with smaller size than t. We used this feature on 2-way test sets, to derive one-way inducing combination. In ten versions of the *replace* the degree of fault was 1. Table 6-11 shows that most faults are interaction faults.

In 13 versions of *replace* and 41 versions of *tcas*, BEN cannot identify inducing combinations in the 6-way test sets, so the degree of fault is more than 6 for these versions. Note that in the *replace* all 13 versions and in the *tcas* 9 of these versions are detected by 2-way testing. A 2-way test set is not guaranteed to detect these versions, since it is not guaranteed to cover all combinations for $t > 2$, and the versions are detected accidentally.

Table 6-12. Fault Classification based on Test Strength

program	Test strength	# of detected versions with the same or lower strength	# of detected versions with higher strength	total	Total not detected
print_tokens	2	0	2	2	5
print_tokens2	2	6	3	9	1
replace	2	17	15	32	0
schedule	2	5	2	7	2
schedule2	2	3	0	3	7
tcas	2	0	9	41	0
	3	0	13		
	4	0	14		
	5	0	4		
	6	0	1		
totinfo	2	1	4	12	11
	3	1	6		

- If a t-way test detects a version, the version does not show in the result of (t+1)-way test.
- All 1-way and 2-way faulty versions of replace are detected in 2-way test set.

We show the strength of fault for detected versions in respect to the test strength in Table 6-12. The second column shows the test strength at which the faulty versions were detected. The third one shows the number of faulty versions that were detected by the test set, and the combinatorial test set guarantees to detect them, since their fault strength is equal or less than the test strength. The forth column shows the number of detected versions with higher fault strength than test strength, which are detected by chance.

For example, by applying 2-way testing to all faulty versions of the *replace* program, we detected not only 17 versions whose degree of fault is 1 or 2, but also 13 versions whose degree of fault is higher than 6.

Another point to note is that, in each step we excluded detected versions in the next step. For example, in the *totinfo* program 5 versions were detected by 2-way testing. One of these 5 versions has the same degree of fault as the test strength, i.e., 2, and the other four versions have the degree of fault higher than the test strength. For the next step we excluded all five versions from testing and we applied 3-way testing only on versions which were not detected.

6.3.7 Comparison

In this section, we show the effectiveness of combinatorial testing by comparing it with random testing. We generated a random test suite corresponds to each combinatorial test set which was used in the previous section. The random test suite and its corresponding combinatorial test set have the same number of tests. For example, the 2-way combinatorial test set for *printtokens* program has 141 tests; thus 141 tests are generated for random testing.

For random test generation, we used the models which were described. Since the subject programs have complex input spaces, we cannot apply random testing without any abstraction. For instance, the first input parameter in the *replace* program is a regular expression; generating valid random regular expressions is impractical.

Our random test generation approach is as follows. For programs whose models do not have any constraint, *schedule*, *schedule2*, *tcas* and *totinfo*, a random value is selected for each parameter in a test. For *printtokens,* we generate the same number of tests as a 2-way test set for each subsystem. If the value parameter comes from the model, such as *identifier*, first we randomly generate a test for value, and then for the subsystem.

If a model has constraints, random selected values may create invalid tests. We avoided invalid tests using the following algorithm. In the *replace* program, constraints are related to the position of elements. There are 4 parameters related to substitution. At most one of them can be *begin* and also at most one can be *end*. Note that it is possible for a test case to not include *begin* or *end* .

To generate random values for substitution related parameters (*sub_character*, *sub_@n*, *sub_@character*, *sub_&*), we define which parameter should appear at the beginning and which one at the end, randomly. A number between 0 and 4 (number of parameters, *sub_character, sub_@n, sub_@character* and *sub_&,* plus 1) are selected randomly. This number is used to select the parameter whose value should be *begin* and appearing at the beginning. If 0 is selected, the first parameter, *sub_character* is set to *begin*, and so on. If 4 is selected, none of the parameters would have *begin* value. Similarly, we select the parameter that should appear at the end. For other parameters, *off* or *middle* is selected randomly. The same approach is used for parameters which are involved in the pattern.

Table 6-13 compares the results of combinatorial and random testing. The second column shows the number of tests in the test sets, third and forth columns are shown the strength and the number of detected versions in combinatorial test set. The last column shows the number of detected versions in random test sets. According to the table, the

Table 6-13. Compare random testing and combinatorial testing

Program	#tests	Combinatorial		Random
		Strength	*#detected version*	*#detected version*
print_tokens	141	2-way	2	1
print_tokens2	141	2-way	9	9
replace	192	2-way	32	17
schedule	64	2-way	7	7
schedule2	64	2-way	3	3
tcas	100	2-way	9	7
	400	3-way	13	14
	1363	4-way	14	6
	4222	5-way	4	12
	10843	6-way	1	2
totinfo	30	2-way	5	2
	156	3-way	7	5

result of random testing is different in different programs. In the two *schedule* programs, *schedule* and *schedule2*, combinatorial testing and random testing have the same results, 7 versions in the *schedule* and 3 versions in the *schedule2* were detected. But in the *replace* program, random testing detected 17 versions compared to 32 versions in combinatorial testing.

In the *tcas* program, combinatorial test set and random test set detected all 41 faulty versions. But combinatorial test can detect more versions by using fewer tests. Combinatorial test sets, 2-way, 3-way and 4-way, detected 36 versions, but random test set with the same number of tests detected 27 versions.

6.4 RELATED WORK

First, we review existing work on input parameter modeling for combinatorial testing. Grindal and Offutt [5] presented a structured method for input parameter modeling. Their method provides guidelines for defining parameters, values, constraints and relations. We followed this method, whereas applicable, in our experiments.

Several common patterns were reported for combinatorial models [15][16]. These patterns include optional values, multi-selection, ranges and boundaries, order and padding, redundant interactions, and auxiliary aggregates or commonality. We used similar ideas for optional values, order and padding, and multiplicity patterns in our experiments. For example, the optional values pattern occurred in the *replace* program. We added the *off* value for each optional parameter.

Segall et al. suggested two constructs, called counters and properties, to model high-level constraints [16]. Some abstract parameters, e.g., the position parameter, identified in our experiments can be considered as properties of a concrete parameter. However, these parameters are not used to facilitate constraint specification in our experiments.

Second, we review existing work on empirical studies on combinatorial testing. We focus on these controlled studies. Dalal et al. [3] reported four relatively large applications that are modeled for combinatorial testing. They reported the number of failed tests and the number of different types of failures that were detected. They showed that combinatorial testing was more effective than traditional testing methods. The difference between their approach and our work is that they did not identify abstract parameters and values. In addition, their subject programs contain real faults, instead of seeded faults [3].

Kuhn et al. studied several fault databases and found that all the faults in these databases are caused by interaction of no more than six parameters [9][10]. This study did not perform actual combinatorial testing on the subject systems.

Schroeder et al. compared combinatorial testing to random testing in a controlled study [14]. They selected two software applications used in their laboratory and used faults that are manually seeded by a graduate student. In contrast, the Siemens suite used in our experiments is a third-party benchmark that has been used to evaluate many testing techniques [18]. We also used faults that come with the Siemens suite.

In [8], Kuhn et al. applied combinatorial testing to a multicomputer network simulator. They compared combinatorial testing to random testing in terms of the number of deadlocks that can be detected by both approaches. The modeling process was not explained in [8].

In [12][13], combinatorial testing was compared to several prioritization techniques and random testing. The experiments were done on two programs *flex* and *make* from SIR [17] repository. The results showed there was no significant difference between combinatorial testing and random testing. The details about the programs models were, however, not, reported in the paper.

6.5 CONCLUSION

In this paper, we presented a three-step approach to apply combinatorial testing. First we create an abstract model for the system. Then, based on this model, a combinatorial abstract test set is generated. The last step derives a set of concrete tests from these abstract tests. We reported our experiments in which we modeled the seven programs in the Siemens suite and applied combinatorial testing to these programs. The details of the abstract model and the results of applying combinatorial testing are presented in the paper. The results show that combinatorial testing can detect most faulty versions of the Siemens programs, and is more effective than random testing.

To better understand the effectiveness of combinatorial testing, we distinguished faults guaranteed to be detected by t-way testing from faults detected incidentally. A fault is detected incidentally by a t-way test set if the degree t' of the fault is higher than t, but the t-way test set happens to contain a t'-way combination that can trigger this fault. In our experiments, we observed that t-way testing often detected some faults incidentally, i.e., the degrees of these faults were higher than t. In particular, for the *tcas* program, all the faults were detected incidentally. This suggests that a t-way test set can be potentially more effective if it covers more higher-strength combinations, in addition to all the t-way combinations.

In the future, we plan to conduct more empirical studies on larger and more complex programs. We believe this research will provide guidance for practitioners to apply combinatorial testing in practice.

6.6 Acknowledgment

This work is supported by two grants (70NANB9H9178 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: NIST does not endorse or recommend any commercial product neither referenced in this paper nor imply that the referenced product is necessarily the best.

6.7 REFERENCES

1. Advanced Combinatorial Testing System (ACTS), 2010. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.
2. M. N. Borazjany, Y. Linbin, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial Testing of ACTS: A Case Study. In Proc. of the 5th IEEE International Conference on Software Testing, Verifcation and Validation, ICST, pages 591-600, Montreal, Canada, 2012.
3. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In Proceedings of the 21st international conference on Software engineering , pages 285-294, New York, USA, 1999.
4. L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. 2012. Identifying Failure-Inducing Combinations in a Combinatorial Test Set. In Proceedings of International Conference on Software Testing, Verification and Validation, IEEE Computer Society, Washington, DC, USA, pages 370-379, 2012.
5. M. Grindal , J. Offutt, Input parameter modeling for combination strategies, Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, pages 255-260, Innsbruck, Austria,2007.
6. M. Grindal, J. Offutt, and S. F. Andler. 2005. Combination Testing Strategies: A Survey. Journal of Software Testing, Verification and Reliability vol. 15, no. 3, pp. 167-199, 2005.

7. R. Krishnan, S. Murali Krishna, and P. Siva Nandhan. Combinatorial testing: learnings from our experience. ACM SIGSOFT Software Engineering Notes, v.32 n.3, May 2007.

8. D. R. Kuhn, R. Kacker, Y. Lei. Combinatorial and Random Testing Effectiveness for a Grid Computer Simulator. presented at the Mod Sim World, Virginia, USA, 2009.

9. D. R. Kuhn and V. Okum. 2006. Pseudo-Exhaustive Testing for Software. 30th NASA/IEEE Software Engineering Workshop, pages 153-158, April 2006.

10. D. R. Kuhn, D. Wallace, and A. Gallo, Software Fault Interactions and Implications for Software Testing, IEEE Transactions on Software Engineering, 30(6): 418-421, 2004.

11. C. Nie and H. Leung. 2011. A survey of combinatorial testing. ACM Computing Surveys (CSUR), v.43 n.2, pages 1-29, January 2011

12. V. Okun, Specification Mutation for Test Generation and Analysis, PhD Dissertation, University of Maryland, 2004

13. X. Qu, M. Cohen, and K. Woolf, Combinatorial interaction regression testing: A study of test case generation and prioritization. In Proceedings of the IEEE International Conferance on Software Maintenance (ICSM). IEEE Computer Society, 413–418, 2007.

14. P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. Proceedings of International Symposium on Empirical Software Engineering, pages 49-59, August 19-20, 2004.

15. Segall, R.Tzoref-Brill, and A. Zlotnick. 2012. Common Patterns in Combinatorial Models. In Proc. of the 5th IEEE International Conference on Software Testing, Verifcation and Validation, ICST, pages 624-629, Montreal, Canada, 2012.

16. Segall, R. Tzoref-Brill, and A. Zlotnick. Simplified Modeling of Combinatorial Test Spaces. In Proc. of the 5th IEEE International Conference on Software Testing, Verifcation and Validation, ICST, pages 573-579, Montreal, Canada, 2012.

17. Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php, 2012.

18. E. Wong and V. Debroy, A survey on software fault localization, Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, Nov. 2009.

Chapter 7. An Empirical Comparison of Combinatorial and Random Testing

The paper was published in IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2014.

An Empirical Comparison of Combinatorial and Random Testing[*]

Laleh Sh. Ghandehari[1], Jacek Czerwonka[2], Yu Lei[1], Soheil Shafiee[1], Raghu N. Kacker[3], D. Richard Kuhn[3]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas 76019, USA

[2]Microsoft Research, Redmond, Washington 98052, USA

[3]Information Technology Laboratory National Institute of Standards and Technology, Gaithersburg, Maryland 20899, USA

Abstract- Some conflicting results have been reported on the comparison between t-way combinatorial testing and random testing. In this paper, we report a new study that applies t-way and random testing to the Siemens suite. In particular, we investigate the stability of the two techniques. We measure both code coverage and fault detection effectiveness. Each program in the Siemens suite has a number of faulty versions. In addition, mutation faults are used to better evaluate fault detection effectiveness in terms of both number and diversity of faults. The experimental results show that in most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better, but with a very small margin. Overall, the differences between the two techniques are not as significant as one would

[*] Copyright © 2014 IEEE. Reprinted, with permission, from Laleh Sh. Ghandehari, Jacek Czerwonka, Yu Lei, Soheil Shafiee, Raghu Kacker and Richard Kuhn, An Empirical Comparison of Combinatorial and Random Testing, IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), April 2014.

have probably expected. We discuss the practical implications of the results. We believe that more studies are needed to better understand the comparison of the two techniques.

Keywords- Combinatorial Testing, Random Testing, Software Testing.

7.1 INTRODUCTION

Software failures are often caused by interactions of a few input parameters. A technique called t-way combinatorial testing, or t-way testing, employs a test set that covers all t-way interactions, i.e. interactions that involve no more than t parameters. If parameters and values are modeled correctly, a t-way test set guarantees to expose all failures that involve no more than t parameters. In practical applications, t is typically a small integer that is no more than six [19].

Many empirical studies show that t-way testing can be very effective in fault detection while significantly reducing the number of tests. However, a question that is often asked by the research community is about the comparative effectiveness of t-way testing. That is, how does t-way testing compare to other testing techniques? In particular, how does t-way testing compare to random testing?

Some conflicting results have been reported in the literature. The studies such as [6][14][17][18] find that t-way testing is more effective than random testing. However, other studies such as [4][5][20][21] suggest that there is no significant difference between t-way testing and random testing. This lack of consensus suggests a need for more studies to better understand the effectiveness of these two techniques.

In this paper, we report a new study that responds to the above need. In particular, we investigate the stability of the two testing techniques. For a given test strength t, multiple test sets can be generated to satisfy t-way coverage. Similarly, multiple random test sets of the same size can be generated. The notion of stability refers to the degree to which the effectiveness of such multiple test sets varies. In practice, testers normally execute only one test set that is essentially an arbitrary selection among multiple possible test sets. The more stable a testing technique, the more confidence one has about the effectiveness of the test set that is actually executed. Our work is inspired by Czerwonka's earlier work that has investigated the stability of t-way testing in terms of code coverage [9]. In this paper we compare the stability of t-way testing to that of random testing and also measure it in terms of both code coverage and fault detection.

In our study, we use the Siemens suite as our subject programs. The Siemens suite has been used a benchmark to evaluate the effectiveness of many testing techniques. The suite consists of seven programs, each of which has a number of faulty versions. Our earlier work modeled the input space of these programs [15]. In this current study, for a given test strength t, a total of 100 t-way test sets are generated for each program. For each t-way test set, a random test set of the same size is also generated. Both t-way and random test sets are generated using the same input models in [15].

The effectiveness of an individual test set is measured in terms of code coverage and fault detection. Code coverage data are collected by running test sets on the error-free version of each program. For fault detection, we run test sets on the error-free version and the faulty versions of each program. A fault is detected if the faulty version produces a different output than the error-free version. A mutation test tool called Milu [16] is used to generate additional faulty versions for three programs in the Siemens suite. Mutation faults increase the number and diversity of the faults used in our experiments and thus helps to better evaluate fault detection effectiveness.

The results of our study suggest that in most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better but with a very small margin. Overall, the differences between the two are not as significant as one would have probably expected. This can be partially explained by the fact that most random test sets have a high percentage of t-way coverage. That is, while a random test set does not cover all the t-way combinations, it covers most of them. A small number of combinations being missing does not always make a difference on code coverage and fault detection results.

It is important to make several notes about the results of our study. First, we used the same input model for t-way and random testing. While t-way test generation is computationally more expensive than random test generation, both procedures are automated. Thus the advantage of random testing in terms of reducing test generation cost is not as significant in practice as one would probably have perceived. Second, in our experiments, the size of a random test set is set to be the same as its corresponding t-way test. However, when we apply random testing in practice, we need to decide when to stop, i.e., how many tests are sufficient. This can be a difficult decision. In this respect, t-way testing has an advantage in that it has a well-defined stopping point, i.e., achieving full t-way coverage. Finally, we must acknowledge that our study is limited in terms of both the number and sizes of the subject programs, and the number and types of faults. More studies are needed to obtain a better understanding.

The remainder of this paper is organized as follows. In section 7.2, we describe our experimental design. Section 7.3 reports experimental results. Section 7.4 provides some general discussion about the experimental results. Section 7.5 describes threats to validity. Section 7.6 gives an overview of work that is related to ours. Section 7.7 provides concluding remark.

7.2 EXPERIMENTAL DESIGN

This section describes the design of our experiments, including the subject programs, the evaluation metrics, and the test generation procedure used by our experiments.

7.2.1 Subject Programs

Our experiments use the Siemens suite from the Software Infrastructure Repository [12]. This suite contains 7 programs. Two programs, *printtokens* and *printtokens2*, have the same specification but different implementations. They tokenize a text file and determine the type of each token. The *replace* program takes three inputs, *pattern*, *substitute* and *input text*, and it *replace*s every match of *pattern* in *input text* with *substitute*. Two programs, *schedule* and *schedule2*, provide two different implementations of a scheduling scheme that determines the execution order of a set of processes based on their priorities. The *tcas* program is an aircraft collision avoidance system. The *totinfo* program takes as input a file containing one or more tables, and computes the total degree of freedom and chi-square of rows and columns.

In the Siemens suite, each program has an error-free version and several faulty versions. There also exists a test set for each program. These test sets are not used in our experiments. Table 7-1 shows some characteristics of the subject programs. The second column shows the number of lines of (uncommented) code. The third column shows the number of functions. The fourth column shows the number of faulty versions. The fifth column shows the input models used for test generation. The input models are shown in an exponential format. For example, *totinfo* has six parameters, where three, two and one of them have a domain size of 3, 5 and 6, respectively. The model of this program is shown in an exponential format by $(3^3 \times 5^2 \times 6^1)$. The last column shows the number of constraints in the input model. The details of the models are explained in [15].

In addition to the faulty versions that come with the Siemens suite, a mutation testing tool called Milu [16] is used to generate additional faulty versions. This helps to better evaluate fault detection effectiveness both in terms of number and diversity of faults. The number of mutants generated by Milu is typically large, and running hundreds of test sets over them is very time consuming. In our experiments, we select three programs, *replace*, *schedule* and *totinfo*, and for each of the three programs, we select a few functions, for mutant generation.

We refer to faults in the faulty versions provided by the Siemens suite as Siemens faults, and faults that are generated by mutation as mutation faults.

Table 7-1. Characteristics of subject programs

Programs	LOC	#of functions	#of faulty versions	Model	Number of constraints
printtokens	472	18	7	$(2^2) \times (2^4) \times (5) \times (8) \times (2 \times 7)^3$ [a]	4[b]
				$(4^7 \times 2^2)$	14
printtokens2	399	19	10	$(2^2) \times (2^4) \times (5) \times (8) \times (2 \times 7)^3$ [a]	4[b]
				$(4^7 \times 2^2)$	14
replace	512	21	32	$(2^4 \times 4^{16})$	36
schedule	292	18	9	$(2^1 \times 3^8 \times 8^2)$	0
schedule2	301	16	10	$(2^1 \times 3^8 \times 8^2)$	0
tcas	141	9	41	$(2^7 \times 3^2 \times 4^1 \times 10^2)$	0
totinfo	440	7	23	$(3^3 \times 5^2 \times 6^1)$	0

a. The model of the *replace* program has two levels; sub level consists of 7 sub models and the top model with 9 parameters. Three out of 7 sub models share the same model, two parameters with 2 and 7 values.

b. The second sub model with (2^4) input model, has 4 constraints and the other does not have any constraints.

Table 7-2 shows some characteristics of generated mutants. The second column shows the number of functions selected for each program. Note that *schedule* is smaller than the other two programs, the mutants are generated for the entire program. The third column indicates the number of mutants generated.

We do not select *printtokens* and *printtokens2* for mutant generation because of the hierarchical nature of their input models. We do not select *schedule2* since it has the same model as *schedule*. Also *tcas* is not selected because it has complex decision logic and its mutants are likely to represent faults with strength of more than 6.

7.2.2 Evaluation Metrics

We measure the effectiveness of an individual test set in two dimensions, i.e., code coverage and fault detection.

For code coverage, line and branch coverage collected for each test set run with the error-free version of each program. A tool called *gcov* is used to gather coverage data. The tool is executed with the "branch-probabilities" option, and the "line executed" output is taken for line coverage and the "taken at least once" output is used for branch coverage.

For fault detection, we check how many faults can be detected by a test set. A fault is detected if the output of a faulty version is different from the output of the error-free version by one or more tests in a test set.

For code coverage and fault detection data collected from a group of test sets, we compute minimum, first quartile (Q1), median, third quartile (Q3), maximum, spread and

Table 7-2. Characteristics of generated mutants

Programs	Number of functions used for mutants generation	Number of mutants
replace	4	143
schedule	18	94
totinfo	2	151

relative standard deviation. The first five measures summarize the effectiveness of the test sets as a group, whereas the latter two summarize how stable the results are across different test sets in the group.

7.2.3 Test Generation

For each subject program, we generate 100 t-way test sets for each strength t, where t is from 2 to 5. There are a total of 400 t-way test sets for each program. We use PICT [10] to generate t-way test sets. PICT uses a greedy, random algorithm for t-way test generation and allows the user to specify a seed. In order to obtain different test sets, a different seed is given each time a test set is generated. Test sets are compared to ensure that no two test sets are exactly the same. In our experiments no redundant test sets are detected. Note that ACTS was not used because it uses a deterministic algorithm which does not give us multiple test sets [1].

For *replace*, we did not generate 5-way test sets as they are very large, and take too much time to execute. On average, there are 12604.22 tests in a 5-way test set for *replace* and it takes 3.22 seconds to execute each test (against all the 32 faulty versions in the Siemens suite). Thus it takes about 11.27 hours to execute each test set. The time needed to execute 100 test sets is prohibitive and thus we did not conduct 5-way testing for *replace* in our experiments. Note that our experiments are conducted on a PC that has a Pentium (R) 4 (2.40 GHZ) processor and 2 GB memory and that runs Ubuntu 12.04 LTS (32bit).

For each t-way test set, we generate a random test set of the same size. The same input model used by t-way test generation is used for random test generation. If the input model of a program does not have any constraint, a random test is generated by simply giving each parameter a random value of its domain. Otherwise, additional care needs to be taken to ensure that all the constraints are satisfied. More details about random test generation with the presence of constraints can be found in [15].

7.3 EXPERIMENTAL RESULTS

In this section, we first present the test generation results, i.e., some important properties and statistics of the test sets generated in our experiments. Then we present the test execution results in terms of code coverage and fault detection that are achieved by these test sets.

7.3.1 Test generation result

Table 7-3 shows some statistics about the sizes of the generated test sets including minimum, maximum, average and relative standard deviation. Note that

Table 7-3. Test sets' size

Program	Strength	Min	Max	Average	RelStdDev
printtokens	2	42	47	44.46	2.72
	3	113	127	119.6	2.17
	4	307	330	319.97	1.64
	5	763	791	776.38	0.80
replace	2	200	220	210.86	2.18
	3	904	955	928.66	1.10
	4	3730	3805	3773.07	0.41
schedule	2	64	64	64	0
	3	244	259	251.22	1.45
	4	1060	1088	1075.30	0.57
	5	3788	3806	3812.26	0.26
tcas	2	100	100	100	0
	3	400	409	403.38	0.47
	4	1401	1447	1423.28	0.65
	5	4240	4321	4277.85	0.36
totinfo	2	31	35	32.41	3.10
	3	150	158	153.26	0.92
	4	532	560	544.5	1.05
	5	1554	1613	1586.35	0.72

printtokens and *printtokens2* use the same input model and thus have the same test sets, and so do *schedule* and *schedule2*. Also note that *printtokens* and *printtokens2* have a hierarchical input model. Due to limited space, we only show statistics for the test sets generated from the top model.

Table 7-4 shows the statistics of the t-way coverage achieved by the random test sets. The t-way coverage of a test set is computed using the ACTS tool with a special option on the command line interface [1]. For most cases, more than 80% (on average) of t-way coverage is achieved by a random test set. The exceptions are for *printtokens* with t = 2 and 3, where the average t-way coverage is more than 70% but lower than 80%. ACTS was not able to compute the t-way coverage for *replace* when t = 4. The reason is that

Table 7-4. Combinatorial coverage of random sets

Program	Strength	Min	Max	Average	RelStdDev
printtokens	2	52.94	82.5	72.03	7.88
	3	54.29	88.5	76.95	11.45
	4	61.10	94.2	86.31	9.24
	5	73.68	95.3	91.76	4.13
replace	2	89.38	96.0	94.85	0.95
	3	89.27	96.4	94.46	1.73
schedule	2	91.15	96.5	93.64	1.06
	3	92.51	94.0	93.43	0.39
	4	94.85	95.6	95.30	0.17
	5	95.66	96.1	95.89	0.08
tcas	2	92.23	96.1	94.25	0.74
	3	93.51	95.1	94.30	0.34
	4	95.15	96.0	95.52	0.17
	5	96.05	96.4	96.26	0.08
totinfo	2	75.78	88.6	82.64	2.96
	3	83.18	88.8	86.20	1.31
	4	83.47	87.1	85.05	0.79
	5	81.92	83.7	82.96	0.46

replace has a relative large and complex input model while the option for computing t-way coverage in ACTS is mainly experimental and is thus not optimized.

7.3.2 Test execution result

The test execution results are presented in three parts, including code coverage results, Siemens fault detection results, and mutation fault detection results.

Code Coverage: Code coverage is collected by running each test set on the error - free version of each subject program. Table 7-5 shows the maximum line and branch coverage achieved by these test sets. Maximum coverage indicates to certain degree the quality of the input model. For *printtokens* and *printtokens2*, the maximum line and branch coverage are shown for the top model and all the sub-models. The maximum line and branch coverage achieved by t-way and random test sets are the same. This is consistent with the fact that both types of test set use the same input model.

Tables 7-6, 7-7, 7-8 and 7-9 show the comparison of some code coverage statistics between t-way and random testing, for four programs, *printtokens2*, *replace*, *tcas*

Table 7-5. Maximum line and branch coverage results

Programs	Max of line coverage	Max of branch coverage
printtokens[a]	46.15, 46.67, 45.13, 43.08, 74.36, 35.38, 47.69	35.78, 36.7, 38.53, 40.37, 57.8, 27.52, 35.78
	69.74	55.05
printtokens2[a]	58.5, 58.5, 57, 71, 73.5, 56.5, 74.5	45.68, 46.91, 46.3, 58.02, 67.9, 40.74, 70.99
	80.5	76.54
replace	88.93	80.56
schedule	94.74	80.30
schedule2	94.57	75
tcas	89.23	90.91[b]
totinfo	92.68	84.09

a. For *printtokens* and *printtokens2* the maximum line and branch coverage achieved by sub models are shown in the first row, in order of number, identifier, keyword, special, character, comment, string sub models. The coverage achieved by top model is shown in the second row.
b. In this program maximum branch coverage is greater than maximum line coverage, the reason is that || and && operators (in an if statement) introduce new branches, in gcov.

and *totinfo*. In these tables, the numbers show the differences between t-way and random testing results. A positive (negative) number indicates that t-way testing performs better (worse) than random testing. Negative numbers are also highlighted.

Due to space limitation we do not show the results for programs where t-way and random testing produced exactly the same statistics. That is, the tables for these programs only consist of zeros. These programs include *printtokens*, *schedule* and *schedule2*. They are made available in [2]. Also, for *printtokens* and *printtokens2*, we show the results of their top model only.

For the *replace* program, t-way and random testing produce the same results for line coverage. However, when t = 3, random testing has a slightly smaller relative standard deviation for branch coverage than t-way testing (Table 7-7).

For the *tcas* program, random testing performed better than t-way testing when t = 2 and 3, whereas t-way testing performed better when t>3 (Table 7-8).

For the *printtokens2* and *totinfo* programs, t-way testing outperforms random testing in many cases. For example, in the *totinfo* program, the minimum line and branch coverage of t-way testing are greater, sometimes significantly greater, than random testing for t = 3, 4, and 5 (Table 7-9). When t = 3, in the *totinfo* program random testing has a smaller standard deviation than t-way testing for both line and branch coverage. However t-way testing has higher *min*, *Q1*, *median* and *Q3* than random testing.

We note that for t-way test sets, *spread* and *standard deviation* are non-increasing as t increases. This indicates that as t increases, code coverage becomes more stable for t-way test sets. This is, however, not true for random test sets. For example, for *totinfo*, the spreads of both line and branch coverage when t = 3 are greater than when t =2. This information is not shown in Table 7-9, which only show the differences between the two methods. The reader is referred to [2] for the specific values of these statistics.

Table 7-6. T-way vs. random coverage result of top model of *printtokens2*

Metric	Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
Line Coverage	2	1.5	1.5	1.5	1.5	0	1.5	0.49
	3	1.5	1.5	1.5	0	0	1.5	0.67
	4	1.5	0	0	0	0	1.5	0.45
	5	0	0	0	0	0	0	0
Branch Coverage	2	2.47	2.47	1.23	1.23	0	2.47	0.99
	3	2.47	1.23	1.23	0	0	2.47	1.04
	4	2.47	0	0	0	0	2.47	0.62
	5	0	0	0	0	0	0	0

Table 7-7. T-way vs random coverage results of *replace*

Metric	Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
Line Coverage	2	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0
Branch Coverage	2	0	0	0	0	0	0	0.01
	3	0	0	0	0	0	0	-0.15
	4	0	0	0	0	0	0	0

Table 7-8. T-way vs random coverage results of *tcas*

Metric	Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
Line Coverage	2	-30.77	0	0	0	0	-30.77	-3.6
	3	0	0	0	0	0	0	-0.01
	4	0	0	0	0	0	0	0.15
	5	0	0	0	0	0	0	0
Branch Coverage	2	-19.7	0	0	-1.52	-6.06	-13.64	-0.62
	3	0	0	0	0	0	0	-0.6
	4	3.03	0	0	0	0	3.03	0.71
	5	0	0	0	0	0	0	0

Siemens Faults: Each program has a number of faulty versions in the Siemens suite in SIR [12]. Table 7-10 show the maximum number of faults that are detected by the t-way and random test sets generated in our experiments.

Table 7-9. T-way vs random coverage results of *totinfo*

Metric	Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
Line Coverage	2	0	1.63	3.25	0	0	0	5.8
	3	11.38	1.63	1.62	0.61	0	11.38	-1.62
	4	1.62	13.82	0	0	0	1.62	1.47
	5	13.14	0	0	0	0	13.82	1.49
Branch Coverage	2	0	1.13	3.97	1.13	0	0	4.4
	3	9.09	1.17	2.27	1.7	0	9.09	-1.42
	4	2.27	9.09	0	0	0	2.27	1.09
	5	9.09	0	0	0	0	9.09	1.08

Tables 7-10, 7-11 and 7-12 show the results for three programs, i.e., *replace*, *tcas* and *totinfo* respectively. The statistics for the other programs are not shown as they are exactly the same between t-way and random testing. Again, positive (negative) numbers indicate cases where t-way testing performed better (or worse) than random testing.

For *replace*, random testing has better *Q1*, *Median*, and *RelStdDev* when t = 2, and better *RelStdDev* when t = 3 (Table 7-11). For *tcas*, random testing performs better when t = 2 and 3, whereas t-way testing performs better when t = 4 and 5 (Table 7-12). For *totinfo*, when t = 2, random testing has a smaller *Spread* but it has a higher *Median* and *Maximum*. When t = 3, random testing has a smaller *RelStdDev*, but all the other measures are the same. When t = 4, t-way testing clearly outperforms random testing, and when t = 5 both reach the maximum results.

For both t-way and random test sets, *spread* and *RelStdDev* are non-increasing as t increases. This suggests that the fault detection results become more stable as t increases [2]. For tcas, the fault detection results do not become stable as t increases. The reason is probably because the degree of all the faults in tcas is more than 5 [15].

Table 7-10. Maximum number of Siemens faults detected

Programs	Total	Max number of faults detected
printtokens	7	2
printtokens2	10	7
replace	32	32
schedule	9	7
schedule2	10	3
tcas	41	41
totinfo	23	12

Table 7-11. T-way vs random Siemens faults detection of *replace*

Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	0	-2	-14	0	0	0	-6.88
3	0	0	0	0	0	0	-2.49
4	0	0	0	0	0	0	0

Table 7-12. T-way vs. random Siemens fault detection of *tcas*

Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	1	-1	-1	0	-3	-4	-3.59
3	-1	-0.75	-1.5	-2	-3	2	-0.35
4	4	1	0	-1	1	3	0.08
5	1	0	0	0	0	1	0.17

Table 7-13. Siemens faults detection of *totinfo*

Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	0	0	2	0	2	-2	2.51
3	0	0	0	0	0	0	-1.97
4	1	0	0	0	0	1	3.02
5	0	0	0	0	0	0	0

Mutation Faults: Only the first 30 (out of 100) test sets are executed on each mutant. This is because running all test sets for each mutant is prohibitively time consuming. For example, it takes 13.19 hours to execute (and evaluate) a 4-way test set on all the 128 mutants of the *replace* program.

Table 7-14 shows the maximum number of mutants detected by t-way and random testing. For the *replace* program all 143 mutants are detected. For *schedule* and *totinfo*, 22 and 27 mutants could not be detected, respectively.

Table 7-15 and Table 7-16 show some statistics of mutation fault detection for *replace* and *totinfo*, respectively. For *schedule*, t-way and random testing have the same results and are thus not shown [2].

For *replace*, t-way testing performed as good as or better than random testing when t = 2 and 3, whereas random testing performed better when t = 4. More discussion on the latter case is discussed later. For *totinfo*, when t = 2, t-way testing have better results in all measures except for *Q3* and *Max*. When t = 3, random testing seems to perform better as it has better results in *Min*, *Q1*, Spread, and *RelStdDev*. However, when t = 4, t-way testing clearly outperforms, and also it reaches the maximum point where the maximum number of faults are detected by all test sets. When t = 5, both t-way and random testing reach the maximum point.

Table 7-14. Maximum number of mutation faults detected

Programs	Total	Max number of faults detected
replace	143	143
schedule	94	72
totinfo	151	124

Table 7-15. T-way vs. random mutation faults detection of *replace*

Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	0	0	0	0	0	0	0
3	0	1	0.5	0	0	0	0.78
4	-26	0	0	0	0	-26	-3.29

Table 7-16. T-way vs random mutation faults detection of *totinfo*

Strength	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	0	0	1	-33.5	-3	3	3.6
3	-5	-1	0	0	0	-5	-9.76
4	7	0	0	0	0	7	1.44
5	0	0	0	0	0	0	0

For *replace*, when t = 4, the minimum number of faults detected by a t-way test is 26 less than that by a random test set. We randomly selected 4 out of these 26 mutants and analyzed their degrees of faults.

Our investigation showed that all these faults are more than 9-way, i.e., they involve more than 9 parameters. Whereas the probability is not high, we conjecture that the reason why there exists a t-way test set that detects none of these 26 mutants is because this test set does contain any combinations that trigger these higher-degree faults. In contrast, it happens to be that all the random tests happen to contain at least one triggering combination for each of these 26 mutants.

7.4 DISCUSSION

In most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better but with a very small margin. Overall, the differences between the two are not as significant as one would have probably expected. As shown in Table 7-4, random test sets provided on average a very high level of combinatorial coverage, almost always in excess of 80% and frequently over 95%. This result is consistent with the formal analysis in [23]. All test sets have some degree of t-way coverage, regardless of how they are generated. The fact that the randomly generated tests had a very high level of t-way coverage can explain why there is little difference between the two techniques.

Although there was little difference between combinatorial and random tests at a particular interaction strength t, fault detection increased rapidly with increasing t. For practical testing, the results suggest that higher levels of combinatorial coverage significantly improve fault detection, regardless of whether the combinatorial coverage is produced by t-way or random test generation.

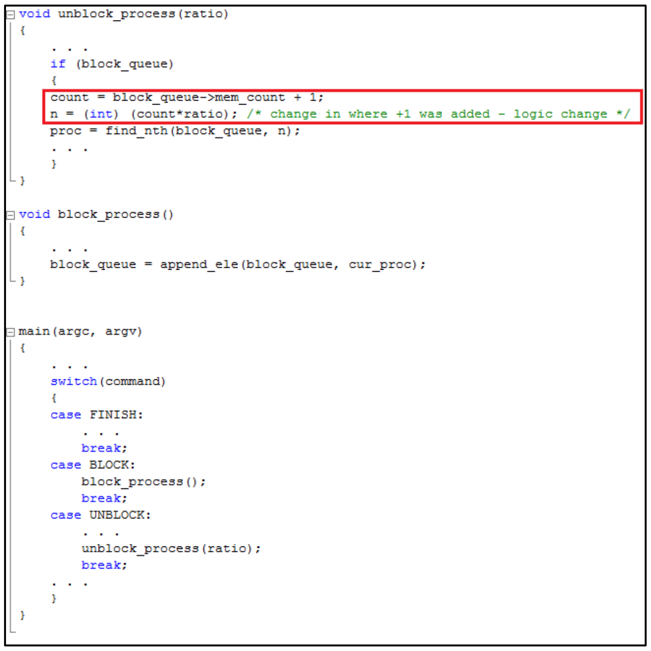
A t-way test set covers all t-way combinations and thus guarantees to detect all t-way faults. Moreover, a t-way test set also covers many combinations whose size is greater than t. Thus, a t-way test set may also detect faults of higher strength, but without guarantee. This phenomenon has been observed in our experiments. For example, all the faults that come with *totinfo* in the Siemens suite have a degree of at least three. However, one of the 2-way test sets generated in our experiments was able to detect 11 of these faults. Another example is the *tcas* program, for which all of the Siemens faults are more than 5-way. Five 5-way test sets generated in our experiments detected all these faults.

Another example is the second faulty version of the *schedule* program from the SIR benchmark. The fault is detected by all 100 2-way test set, while it is a 3-way fault. The fault is shown in Figure 7-1. The Schedule program takes the following inputs: (1) three non-negative integers representing the number of processes in three different priority queues; and (2) a list of commands that must be executed on the queues. The output of this program is a list of numbers indicating the order in which the processes exit (from the scheduling system).

There are seven commands: *new job*, *upgrade_prio*, *block*, *unblock*, *quantum_expire*, *finish* and *flush*. The faulty statement is in the *unblock_process* function where a process that should be unblocked is selected from the blocked queue. In the error - free version the unblock command operates on the n-th process where $\mathrm{n} = (\mathrm{int})\ (\mathrm{count} * \mathrm{ratio}\ +\ 1)$. In the faulty version n is computed by the two marked statements in Figure 7-1.

To trigger this fault, the following two conditions need to be satisfied: (1) the *unblock_process* must be called; and (2) the integer value, n, computed by the faulty statements should be different from the one computed by the error-free statement.

The values of two parameters, *ratio* and *size* of the blocked queue must be selected such that the second condition is satisfied. The blocked queue is initially empty



```
void unblock_process(ratio)
{
    . . .
    if (block_queue)
    {
    count = block_queue->mem_count + 1;
    n = (int) (count*ratio); /* change in where +1 was added - logic change */
    proc = find_nth(block_queue, n);
    . . .
    }
}

void block_process()
{
    . . .
    block_queue = append_ele(block_queue, cur_proc);
}


main(argc, argv)
{
    . . .
    switch(command)
    {
    case FINISH:
        . . .
        break;
    case BLOCK:
        block_process();
        break;
    case UNBLOCK:
        . . .
        unblock_process(ratio);
        break;
    . . .
    }
}
```

Figure 7-1. Second Siemens Fault of Schedule Program

and the block command should be called to add process to the blocked queue and therefore change its size.

Thus the following combination of the three parameters detects the fault: (1) number of times the block command is called (which determines the size of the blocked queue), (2) the value of ratio, and (3) calling the unblock command at least once. Based on the abstract model of the *schedule* program [15], there exist a total of six 3-way inducing combinations that triggers this fault. These inducing combinations are shown in Table 7-17. Each of the 100 2-way test sets contains at least one of these six inducing combinations and thus detects this fault.

7.5 THREADS TO VALIDITY

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. We have tried to automate the experimental procedure as much as possible, as an effort to remove human errors. In particular, we build a tool that automatically compares the results of the error-free version and a faulty version to evaluate each test run. Further, the consistency of the results are checked manually to determine whether the tool works correctly or not.

Table 7-17. 3-way inducing combinations of the second faulty version of the *schedule*

3-way inducing combination
$\text{Block} \leftarrow \text{"1"}, \text{Unblock} \leftarrow \text{"1"}, \text{Unblock\_ratio} \leftarrow \text{"\{r\} = 0.5"}$
$\text{Block} \leftarrow \text{"1"}, \text{Unblock} \leftarrow \text{"1"}, \text{Unblock\_ratio} \leftarrow \text{"\{r\} = 0.1"}$
$\text{Block} \leftarrow \text{"1"}, \text{Unblock} \leftarrow \text{"1"}, \text{Unblock\_ratio} \leftarrow \text{"\{r\} = 0.4"}$
$\text{Block} \leftarrow \text{"1"}, \text{Unblock} \leftarrow \text{"1"}, \text{Unblock\_ratio} \leftarrow \text{"\{r\} = 0.9"}$
$\text{Block} \leftarrow \text{"1"}, \text{Unblock} \leftarrow \text{"1"}, \text{Unblock\_ratio} \leftarrow \text{"\{r\} = 0.5"}$
$\text{Block} \leftarrow \text{"1"}, \text{Unblock} \leftarrow \text{"1"}, \text{Unblock\_ratio} \leftarrow \text{"\{r\} = 0.1"}$

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from the Siemens suite [12]; these programs are created by a third party, but the subject programs are programs of relatively small size and with a small number of seeded faults. To mitigate this threat, the mutation faults are added to the experiments. But more experiments on larger programs with real faults can further reduce this threat.

7.6 RELATED WORK

A number of studies have been reported that evaluate the effectiveness of t-way testing. In this section, we focus on related work that compares the effectiveness of t-way and random testing.

Schroeder et al. in [20] conducted an experiment to compare the fault detection effectiveness of combinatorial and random test sets. The subject programs are two real-life programs in C++, including the Data Management Analysis System (DMAS) and the Loan Arranger System (LAS), only one functionality of each program is tested. Their results show that there is no significant difference in t-way and random testing in terms of fault detection.

DMAS and LAS have 8.7 and 6.2 KLOC, and their input models are represented as $(2^{16} \times 5 \times 8)$ and $(2^{7} \times 3^{10} \times 4^{2})$, respectively. For each program, and for each strength t, where t is from 1 to 4, 10 t-way test sets are generated using a tool called TVG [20]. For each t-way test set, a random test set of the same size is generated. Mutants are created manually to generate faulty versions. Mutants that are killed by all the 1-way sets are removed. A total of 88 mutants for DMAS and a total of 82 mutants for LAS are used in their experiments.

In [20], a random test is generated by randomly selecting a test from all possible tests. This is different from our approach in which a random test is generated by giving each parameter a random value in its domain. This difference may slightly affect the combinatorial test coverage achieved by a random test set. Note that the approach in [20] assumes that all possible tests are first generated, which may not be practical for large input models.

Ellimis et al. [13] [14] report an experiment that tests 10 different functions of a system called Wallace that controls a large industrial engine. A mutation tool is used to generate faulty versions. For each function, three test sets are generated, one t-way test set, one pure random test set, and one manually generated test set. Pure random tests are generated without using an input model.

Their results show that 2-way test sets are not as effective as manually generated tests in term of fault detection. But a t-way test set of a higher strength could be as effective as a manually generated test set. Their results also show that random test sets may often provide good results. For example, for 5 out of 10 programs, random and t-way test sets provide the same results, and in one case random test sets even produce better results than t-way test sets.

Several studies are reported that compare t-way testing and random testing for testing logical expressions [5][17] [21]. The logical expressions are either taken from a program such as TCAS II or generated randomly. Mutants are generated to create faulty versions. The results consistently show that t-way testing is always more effective, and sometimes significantly more, than random testing

Kuhn et al. [18] report a study that applies t-way testing and random testing to detect deadlocks in a network simulator called Simured. The input model for the simulator is $(2^3 \times 3 \times 4^9 \times 5)$. T-way test sets are generated by ACTS with t = 2 to 4. For each t-way test set, eight random test sets of the same size are generated corresponding to each combinatorial set. Their experiments show that (1) random testing has better results than

2-way testing; (2) no significant difference exists between random and 3-way testing; (3) 4-way testing is more effective than random testing.

Bell and Vouk applied 2-way testing and random testing to a network-centric software [6]. They found that 2-way testing is more effective in fault detection. In particular, when there is at least one parameter with more than 10 values, random testing does not detect about 75% of faults that are detected by 2-way testing

Bryce et al. compared the coverage of combinatorial and random testing on a system called Flight Guidance System (FGS) [8]. The FGS system has 40 input parameters, each of which has 2 values. Four t-way test sets with t = 2 to 5, as well four random test sets of the same size, are generated. Their results show that t-way testing is more effective than random testing for the FGS system.

A formal analysis in [3] shows that a random test set of the same size as a t-way test set, could trigger at least one t-way fault with a probability of greater than 0.63. This is consistent with our results in Table 7-4. The analysis in [3] also shows random testing becomes more effective as the number of parameters increases and converges toward being equally effective as combinatorial testing. The analysis assumes no constraints that exist between parameters.

Finally we note that Czerwonka reported a study [9], that applies t-way testing to four utility programs in Windows 7, including attrib.exe, fc.exe, find.exe and findstr.exe. The focus of the study is to investigate the stability of t-way testing in terms of line and branch coverage. The results show that t-way test sets provide stable coverage when t = 2. This study, however, does not make a comparison with random testing.

7.7 CONCLUSION

In this paper, we report a study that compares the effectiveness of t-way testing to that of random testing in terms of both code coverage and fault detection. In particular, we investigated the stability of the two techniques. Our results show that in most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better, but with a very small margin. Overall, the differences between the two are not as significant as one would have probably expected. A possible explanation is that most random test sets seem to achieve a high level of t-way coverage. More studies are needed to better understand the effectiveness of the two testing techniques.

We plan to conduct more empirical studies to further evaluate the effectiveness and stability of combinatorial testing. We plan to use programs that are larger and/or more complex than the Siemens programs. We also plan to conduct studies where the degree of fault can be better controlled. This will help us to better study the relationship between the combinatorial coverage of a test set and the faults the test set is able to detect.

7.8 Acknowledgment

We thank Yue Jia for providing the Linux version of mutation tool. This work is supported by two grants (70NANB12H175 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

7.9 REFRENCES

1. Advanced T-way testing System (ACTS), http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html, 2013.
2. Applying Combinatorial Testing, http://barbie.uta.edu/~laleh/BEN/ben.html, 2014.

3. A. Arcuri, L. Briand, "Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing," IEEE Transactions on Software Engineering 38(5):1088-1099, 2012.

4. J. Bach, P. J. Schroeder, “Pairwise testing: A best practice that isn’t”, In Proceeding of the 22nd Annual Pacific Northwest Software Quality Conference, pp. 180-196, 2004.

5. W. A. Ballance, S. Vilkomir, W. Jenkins, “Effectiveness of Pair-Wise Testing for Software with Boolean Inputs”, In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp.580-586, 2012.

6. K. Z. Bell, M. A. Vouk, "On effectiveness of pairwise methodology for testing network-centric software". In Proceeding of the ITI 3rd International Conference on Information and Communications Technology, pp.221-235, 2005.

7. M. N. Borazjany, Y. Linbin, Y. Lei, R. Kacker, and D. R. Kuhn, “T-way testing of ACTS: A Case Study”, In Proceedings of the IEEE fifth International Conference on Software Testing, Verification and Validation, pp.591-600, 2012.

8. R.C. Bryce, A. Rajan, M.P.E. Heimdahl, "Interaction Testing in Model-Based Development: Effect on Model-Coverage," In Proceeding of the 13th Asia Pacific Software Engineering Conference, pp.259-268, 2006.

9. J. Czerwonka, “On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Deigns”, In Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 257-266, 2013.

10. J. Czerwonka, “Pairwise testing in real world. Practical extensions to test case generators”, In Proceedings of 24th Pacific Northwest Software Quality Conference, pp. 419-430, 2006.

11. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In Proceedings of the 21st international conference on Software engineering, pp. 285-294, 1999.

12. 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.

13. M. Ellims, D. Ince, and M. Petre, "AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation," Technical Report 2007/08, Dept. Computer Science, Open University, Milton Keynes, June 2007.

14. M. Ellims , D. Ince, M. Petre, "The effectiveness of t-way test data generation", In Proceedings of the 27th international conference on Computer Safety, Reliability, and Security, pp. 16-29, 2008.

15. L. S. Ghandehari, M. N. Bourazjany, Y. Lei, R.N. Kacker and D.R. Kuhn, "Applying T-way testing to the Siemens Suite", In Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 362-371, 2013.

16. Y. Jia and M. Harman. “Milu: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language”, TAIC PART '08. Testing: Academic & Industrial Conference, pp. 94-98, 2008.

17. N. Kobayashi, T. Tsuchiya, T. Kikuno, "Applicability of non-specification-based approaches to logic testing for software", In Proceeding of the International Conference on Dependable Systems and Networks, pp. 337-346, 2001.

18. D.R. Kuhn, R. Kacker, Y.Lei. “Random vs. Combinatorial Methods for Discrete Event Simulation of a Grid Computer Network”, In Proceedings of ModSim World, pp. 83-88, 2009.

19. D.R. Kuhn, D.R. Wallace, and A.M. Gallo. “Software Fault Interactions and Implications for Software Testing”, IEEE Transaction on Software Engineering 30(6):418-421, 2004.

20. P.J. Schroeder, P. Bolaki, V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," In Proceeding of the International Symposium on Empirical Software Engineering, pp.49-59, 2004.

21. S. Vilkomir, O. Starov and R. Bhambroo, “Evaluation of t-wise approach for testing logical expression in Software”, In Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp.249-256, 2013.

22. D. R. Wallace, D. R. Kuhn, “Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data”, In Proceeding of the ACS/ IEEE International Conference on Computer Systems and Applications, pp. 301-311, 2001.

Chapter 8. Conclusion

In this dissertation, we present a fault localization approach based on combinatorial testing. Our approach, i.e., BEN, consists of two phases, i.e., failure-inducing combination identification and faulty statement localization.

The novelty of our approach is twofold. In the first phase, we introduced two notions of suspiciousness, suspiciousness of a combination and suspiciousness of its environment. BEN uses these notions to rank suspicious combinations. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked.

In the second phase, we generate a small group of tests that include a failing test and several passing tests that are very similar to the failing test. The spectra of these tests are analyzed to rank statements in terms of their likelihood to be faulty. This approach is inspired by the notion of nearest neighbor, i.e., the faulty statement is likely to appear in the execution trace of a failed test but not in the execution trace of a passed test which is similar to the failed test as possible.

It is important to emphasize that BEN differs from existing spectrum-based approaches such as Tarantula and Ochiai in that existing approaches require the spectra of all test executions be recorded. If a test set is already executed without being traced, the test set must be re-executed to collect traces before they can be used by these approaches. In contrast, BEN only requires the spectra of a small number of tests generated in the second phase and can be applied after a regular testing process is performed where test executions are not traced.

We conducted our experiments in which BEN is applied to the Siemens suite and two real-life programs, i.e., the grep and gzip programs. Our experimental results show that BEN is very effective, i.e., significantly reduces the number of statements to be inspected for fault localization, and efficient, i.e., a very small number of tests needs to be generated and traced. Moreover, the comparison of BEN to two spectrum-based approaches, Tarantula and Ochiai, show that BEN achieves the results that are competitive or even better than Tarantula 18and Ochiai while requiring significantly fewer tests to be instrumented.

This dissertation also includes two empirical studies that were conducted as part of our effort to evaluate the effectiveness of BEN. These studies provide additional data that demonstrate the effectiveness of combinatorial testing. In the first study, we applied combinatorial testing on the Siemens programs. Our experimental results show that combinatorial testing is very effective in that it detects most faulty versions of these programs. In the second study, we compared combinatorial testing and random testing in terms of code coverage and fault detection. In particular, we investigated the stability of the two techniques. Our results suggest that in most cases combinatorial testing performs better or as good as random testing.

We plan to conduct more empirical studies to further evaluate the performance of our approach on different fault types. In particular, we plan to investigate how BEN works on security faults such as Buffer Overflow and Cross Site Scripting vulnerabilities. Security faults seem to demonstrate some different properties than functional faults. It is expected that BEN needs to be adapted for localizing security faults. We also plan to investigate how to adapt our approach to work with an arbitrary test set. The challenge is to deal with the fact that unlike a combinatorial test set, an arbitrary test set does not guarantee that all t-way combinations are covered. This could potentially reduce the effectiveness of our approach.