Comparing the Fault Detection Effectiveness of N-way and Random Test Suites

Patrick J. Schroeder, Pankaj Bolaki, Vijayram Gopu
Department of Elec. Eng. and Comp. Sci.
University of Wisconsin - Milwaukee
Milwaukee, WI. 53211
{pats,pbolaki,vvgopu}@uwm.edu

Abstract

Software testing plays a critical role in the timely delivery of high-quality software systems. Despite the important role that testing plays, little is known about the fault detection effectiveness of many testing techniques. In this paper we investigate "n-way" test suites created using a common greedy algorithm for use in combinatorial testing. A controlled study is designed and executed to compare the fault detection effectiveness of n-way and random test suites. Combinatorial testing is conducted on target systems that have been injected with software faults. The results are that there is no significant difference in the fault detection effectiveness of n-way and random test suites for the applications studied. Analysis of the random test suite finds that they are very similar to n-way test suites from the perspective of the number of test data combinations covered. This result concurs with other hypothetical results that indicate little difference between n-way and random test suites. While we do not expect this result to apply in all combinatorial testing situations, we believe the result will lead to the design of better combinatorial test suites.

Keywords: Combinatorial testing, n-way testing, fault injection, controlled experiment, empirical study.

1. Introduction

Software testing is a critical, but expensive, part of the process of creating and delivering high-quality software systems. Testing often accounts for 40-80% of the project budget [1]. As software becomes ubiquitous the consequences of bad software become ever more serious. Despite its importance and cost, little is known about the fault detection effectiveness of many of testing techniques.

In many software testing situations, testers must consider how to test combinations of a large number of system input variables. For example, many systems require a user to enter data into several different variables, or fields, in an online form. When the form is completely filled in, the user submits it to the system for processing. To test such a form, test data values are selected for each of the form's input variables; however, the tester must also be concerned with testing different combinations of the test data values. This is because it is possible that some software failures may occur only when a specific combination of test data values is entered into the system. Testing situations such as this are said to require *combinatorial* testing, because they require testing several input variables in combination.

Combinatorial testing is difficult for testers because of the large number of possible test cases (a result of the "combinatorial explosion" of selected test data values for the system's input variables). Running all possible combinatorial test cases is generally not possible due to the large amount of time and resources required; however, selecting too few combinatorial test cases may increase the risk of passing a damaging software fault onto the end-users of the product. It is important that testers find an effective strategy for selecting combinatorial test cases executable within a project schedule and resource constraints.

A very popular approach to combinatorial testing is referred to as *n-way* testing [2], where *n*, the level of interaction among the system's input variables, may be set to 2, 3, 4, or higher. In n-way testing, a set of test cases is generated to cover a subset of the possible combinations of the system's input variables, rather than trying to cover all possible combinations. The n-way testing technique has been the topic of over 40 conference and journal articles, and is presented in almost all recent books on software testing techniques (e.g., [3-7]). A result of this exposure is that the topic is now part of many commercial and academic software testing courses, including a testing course taught by one of the authors. In general, n-way testing is widely believed to be a systematic, effective

Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)
0-7695-2165-7/04 $20.00 © 2004 **IEEE**

testing technique. This view is often supported by a recent study by Wallace and Kuhn [8] that analyzed 15 years of failure data from recalled medical devices. They conclude that 98% of the failures could have been detected in testing if all pairs of parameters had been tested (i.e. 2-way testing).

Despite the popularity of n-way testing, there are some indicators that n-way testing may not be effective in every case. In testing Remote Agent Experiment (RAX) software used to control NASA spacecraft, Smith, et al. [9] present their experience with n-way testing. Their analysis indicates that 2-way testing detected 88% of the faults classified as correctness and convergence faults, but only 50% of the interface and engine faults. Smith, et al. do not disclose how the addition engine and interface faults were found, highlighting another problem with n-way testing: no controlled studies exist that compare the fault detection effectiveness of n-way testing to any other combinatorial testing technique.

Given that no controlled study of the fault detection effectiveness of n-way testing techniques exists, and that existing experience reports are contradictory, the objective of this work is to conduct a controlled study that compares the fault detection effectiveness of the n-way and random testing techniques. Fault detection effectiveness (FDE) is the percentage of faults detected by a testing technique when executed on system with known fault content. If we are to continue to study, practice, and teach n-way testing techniques, we would like some indication that they perform better than simple random selection.

In this paper we present the results of our controlled study. In section 2 we discuss combinatorial testing and review current literature. In section 3 we describe the goal and preparation of the experiment. In section 4 we discuss experimental planning. In section 5 we describe the experimental procedure. In section 6 analyze the results of the experiment, and in sections 7 we state our conclusions and future work.

2. Combinatorial Testing

2.1 Combinatorial testing techniques

Combinatorial testing is considered whenever testing system operations, or functionality, require multiple data input values. These data input values may come from a variety of sources, including: the users of the system, the file system or database, or from across a network. Combinatorial testing at the system level is approached using black-box techniques (black-box techniques do not use information from the internals of the software in creating test cases). When testing at the system level, we focus on ensuring that the software system meets its original high-level requirements; lower-levels of testing such as the unit-level and integration-level focus on ensuring that individual software components of the system function properly and communicate effectively.

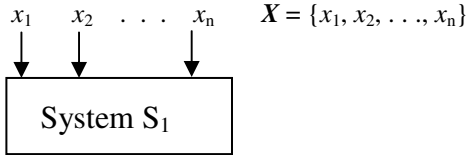

Figure 1. System S1 with set of input variables X

For an example of a combinatorial testing problem, consider system S_1 in Figure 1. We use $\boldsymbol{X}$ to indicate the set of all input variables identified for use in testing S_1. To test a system such as S_1, a set of test cases $T = \{t_1, t_2, t_3, \ldots t_n\}$, also called a *test suite*, would be created. To create T, test data values for each element of X are selected. One approach to selecting test data values is to use black-box test data selection techniques such as equivalence partitioning and boundary value analysis [4, 10]. After applying test data selection criteria to each element of $\boldsymbol{X}$, we will have a set of test data values for each of the input variables: $D(x_1)$, $D(x_2)$, …, $D(x_n)$. Because S_1 has multiple input variables, the tester must now consider testing combinations of the selected test data values. T_{cart}, the set of all possible combinations of the test data values, is created by taking the Cartesian product of the sets of selected test data values for the system's input variables:

$$T_{cart} = \{ D(x_1) \times D(x_2) \times D(x_3) \ldots D(x_n) \}$$

The result is a set of ordered n-tuples (d_1, d_2, …, d_n) where each d_i is an element of set $D(x_i)$. The size of T_{cart}, and the corresponding number of test cases, is normally quite large ($|T_{cart}| = |D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_n)|$). For example, if 3 test data values are selected for a system with 10 input variables, the number of possible test cases would be $3^{10} = 59{,}049$. This number of test cases may be too large to execute, or the testing priority for this operation may not warrant such a large number of test cases, so a smaller test suite must be created.

To employ the *n-way* testing technique, a set of test cases is selected from T_{cart} that cover, or contain, all combinations of the selected test data values for all possible subsets of the system's input variables of size n. The most common setting of n is 2. This is referred to as 2-way, or pairwise, testing. We refer to T_{pair} as the pairwise test suite. That is:

$T_{pair} \subset T_{cart}$ such that, for all pairs of inputs $(x_i, x_j) \in X$ and for every $(i, j) \in \{ D(x_i) \times D(x_j) \}$ there exists a ordered n - tuple $(d_1, d_2, \ldots, d_n) \in T_{pair}$ such that $d_i = i$ and $d_j = j$.

For example, consider a system with 3 input variables X = {A, B, C} where D(A) = {1, 2}, D(B) = {Q, R}, and D(C) = {5, 6}. The size of T_{cart} = 2×2×2 = 8 test cases. T_{pair} has a size of 4 test cases and is shown in Table 1.

Table 1: T_{pair} for inputs A, B, and C

Test ID	Input A	Input B	Input C
TC_1	1	Q	5
TC_2	1	R	6
TC_3	2	Q	6
TC_4	2	R	5

N-way test suites are used in a variety of ways in software testing. They are often used in setting the configuration of the testing environment when a systems must be tested on multiple hardware and operating system combinations (e.g., [5, 11]). They have also been used in evaluating the performance of software systems [12]. In our work, we use n-way testing in system-level functional testing conducted with the intent of exposing faults in the software.

Our study also uses random combinatorial test suites. A random combinatorial test suite, T_{rand} = rand(T_{cart},n), where rand is a function that uses a pseudo-random number generator to select n test cases from T_{cart} without replacement. The size of T_{rand} may range for 1 to $|T_{cart}|$; however, in our study we generate random test suites that match the size of the n-way test suite we are evaluating. For example, in all evaluations of pairwise test suites, $|T_{pair}| = |T_{rand}|$. This is done to ensure that the FDE of the test suite is attributed to the technique used to create it, rather than the size of the test suite.

2.2 Review of previous empirical work

As pairwise testing has grown in popularity, the interest in understanding its fault detection effectiveness has also grown. In this section we review some of the experiments and case studies conducted using the n-way testing technique.

Early work by Mandl [13] described the use of orthogonal Latin and Greco Latin Squares in the validation of an Ada compiler. He concludes that orthogonal Latin squares are a very efficient compromise between testing effort and information obtained. He also points to random selection as being "much less effective in detecting malfunctions and yields much less information when it does in fact detect something," although no data is presented to support this intuition. Another early case study in pairwise testing is presented in Brownlie, et al. [14]. This study used the Robust Testing strategy, which includes the use of the OATS (Orthogonal Array Testing System), in the system level testing of AT&T's PMX/StarMAIL product. The results indicate a 22 percent advantage in fault detection effectiveness for the Robust Testing technique over *conventional* testing. While clearly a successful testing campaign, the fault detection effectiveness of the conventional testing technique is "estimated" using information from the change control database, rather than directly measured.

A great deal has been written about achieving coverage using pairwise testing [15-22]. "Coverage" in the majority of the studies refers to code coverage, or the measurement of the number of lines, branches, decisions or paths of code executed by a particular test suite. High code coverage has been correlated with high fault detection in some studies [23]. In general, the conclusion of these studies is that testing *efficiency* (i.e., amount time and resources required to conduct testing) is improved because the much smaller pairwise test suite achieves the same level of coverage as larger combinatorial test suites.

Many industrial case studies have been conducted using the Automatic Efficient Test Generator (ATEG). The AETG is a commercial testing tool that can generate n-way test suites. Cohen, et al. present three studies that apply pairwise testing to a Bellcore inventory database systems [17, 20, 24]. Software faults found during these studies lead the authors to conclude that the AETG coverage seems to be better than the *previous* test coverage. Dalal, et al. [25] presents two case studies using the ATEG on Bellcore's Integrated Services Control Point (ISCP). Similar studies by Dalal, et al. are presented in [19]. In all of these studies the pairwise testing technique appears to be effective in finding software faults. Some studies compare the fault detection of the pairwise testing with a less effective concurrently executed testing process.

The results of these case studies are difficult to generalize. In most of the studies, pairwise testing performs better than *conventional*, *concurrent*, or *traditional* testing practices; however, these practices are never described in enough detail to understand the significance of these results. Additionally, little is known about the characteristics of software systems used in the studies (e.g., how much combinatorial processing do the systems perform?). Since the previously mentioned RAX study [9] found that pairwise testing detected only 50% of engine and interface faults in the system, it appears that n-way testing may not be universally effective across all types of systems.

Research has also been conducted on the relationship between random testing and n-way testing. Dalal and Cohen [18] analyzed pairwise and random test data sets

generated for several hypothetical cases. They found a high degree of similarity in the test data sets with respect to the number of pairs of inputs covered. By definition, we know that a pairwise test data set covers 100% of the combinations of the selected test data values for every *pair* of input variables is created. Dalal and Cohen analyzed same-sized random test data sets and found them to cover a very high percentage of input pairs; as high as 90% in many cases. Based on these hypothetical cases, one may not expect the FDE of pairwise and same-sized random test data sets to vary a great deal.

Our review of the literature found that no controlled experiments had been conducted on the n-way technique, leading us to believe that conducting a controlled experiment would be insightful. The experiment will be conducted in an academic environment and so will clearly not be the final words on the topic, however, we believe the experiment can contribute to the discussion of the FDE of n-way test suites.

3. Experimental Goal and Preparation

The goals of the experiment are stated using the Goal/Question/Metric (GQM) paradigm [26] as follows:

Analyze ***n-way*** *and* ***random combinatorial testing techniques*** *for the purpose of* ***comparison*** *with respect to their* ***fault detection effectiveness*** *from the point of view of a* ***researcher*** *in the context of an* ***academic laboratory***.

Since no controlled experiments using n-way testing exist, we choose to begin the discussion with a comparison of n-way and random test suites. Given the high degree of popularity of the n-way technique with researchers, practitioners, and teachers of software testing, it is important to confirm that n-way testing technique performs better than simple random testing. The following sections explain the steps taken to prepare for the experiment.

3.1 Program selection

In order to conduct this experiment, two industrial software systems were selected. Since no widely accepted software classification scheme with representative systems exists, we used systems currently available to us in the laboratory. The systems we used are well known to the authors as they were the initial developers, have used the systems to conduct other research, and have used the systems as projects in teaching software engineering courses.

A requirement of the selected systems is that they require a significant combinatorial testing effort. Not all systems fit the combinatorial model of testing. The combinatorial model of testing is used for system operations that require multiple input variables, follow an input-process-output pattern, and create outputs largely determined by the value of the inputs, rather than on the state of software system. When the state of the system plays a significant role in the behavior of the system, as in real-time process control systems for example, a state-based model of testing is adapted [27, 28].

The first system is the Data Management Analysis System (DMAS). The DMAS is used by analytical chemists to process data files collected from experiments conducted using liquid and gas chromatography. The application was originally written in FORTRAN by a team of programmers, including one of the authors, while working in the pharmaceutical industry in the late 1980s. In the 1990s, the program was converted to C++ for use in an academic environment in research and software engineering course projects. While the DMAS provides a wide variety of functionality, in this experimental study, we tested only the *product sample concentration report*. The report requires the tester to consider the combination of 18 different input variables. The entire DMAS consists of 8.7 KNCSL (thousand (K) Non-Comment Source Lines). A full description of the DMAS is reported by Schroeder, et al [29].

The second system selected for the experiment is the Loan Arranger System (LAS). LAS is a software application that supports the mortgage-back securities business. The requirements for the system are specified in Pfleeger's software engineering textbook [30]. LAS is used as a semester long software engineering project at the University of Wisconsin-Milwaukee. It was developed in C++ in a UNIX environment. It consists of 6.2 KNCSL. For the LAS application, we tested only the system's *administrative mode* that is used to update and maintain the system's data repository. In testing this functionality, testers must consider the combinations of 19 different input variables.

3.2 Fault seeding

Many experiments have been carried out using fault seeding to evaluate testing techniques in terms of their FDE [31, 32]. In this experiment, fault seeding is used to create a test bed of faulty systems by injecting one fault into each copy of the software. Many faulty versions of the software are created as is done in *mutation testing* [32-35]. To create each "mutant" (i.e., a copy of the original program with one fault injected into it) we chose to inject faults taken from a subset of a model of "common software errors" documented by Kaner, et al [4]. Table 2 provides information regarding the faults injected into LAS and DMAS for the experiment.

In Table 2, the *Fault Classification* column describes how a fault is injected, or created, in the software; the

rightmost columns of the table give the number of times each fault type was injected into LAS and DMAS.

A graduate student was employed to inject the faults into the systems. The graduate student knew little of combinatorial testing or the n-way testing technique. The student was given a section of code and told to inject faults into code according to the fault classification description. For example, a *Change Operators or Conditions* fault may be created by changing operators in conditional statements, such as changing an *equal* sign (==) to a *not equal* sign (!=). The graduate student injected faults into the code until she believed she had, in her opinion, adequately covered the code for the given fault classes. Due to the nature of the code, some types of faults are more prevalent than others; no effort was made to inject the same number of each type of fault.

Table 2: Fault classification for LAS and DMAS

Fault Acronym	Fault Classification	# Faults	
		LAS	DMAS
	IF statements		
COC	- Change operators or conditions	34	40
ARC	- Add/Remove conditions	15	6
CSI	- Change statement(s) inside IF statements	12	29
ILE	- Information loss errors	16	16
	Loops		
IIF	- Incorrect initial or final value	20	10
CLE	- Change in loop exit condition	7	5
	Arrays		
AVL	- Assign incorrect values or Assign to incorrect locations	8	6
	Statements		
ARS	- Add/Remove statement(s)	20	29
IAS	- Incorrect var. initialization or incorrect assignment	24	73
ISP	- Incorrect statement placement	12	32
PAE	- Precision/Approx. errors	16	7
APE	- Argument passing errors	5	6
IRV	- Incorrect return values	0	4
	Total number of mutants created	189	263
	Number of 1-way detected faults removed	107	175
	Number of mutants used in the study	82	88

In this study, we did not attempt to inject complex faults. Since the testing we are conducting is system-level testing, one could argue that we should inject complex system-level faults. Complex system-level faults, as experienced by one of the authors in maintaining industrial systems, can result from the interaction of multiple errors in different parts of the system. Unfortunately, there is no accepted model of complex system-level faults. Because this is the first attempt at a controlled evaluation of n-way testing, we chose the simpler, well established fault model.

When injecting faults into software using a simple model it is easy to inject faults that cause severe system failures, which are easily detected in testing. Faults that are detected by *every* test case can give us no insight into the FDE of n-way and random test suites. In order to eliminate these obvious faults, 10 different all values, or 1-way, test suites were generated. A 1-way test suite covers all selected test data values for all system input variables (i.e., the suite covers all test data values only, without considering any combinations of these values). For LAS the size of the 1-way test suite was 4 test cases; for DMAS the size of the 1-way test suite was 5 test cases. Faults detected by all 10 1-way suites were eliminated from the experiment. At the bottom of Table 2, the total number of mutants created, the number of faults found and removed by the 1-way test suites, and the total number of mutants used in the experiment is given.

3.3 Test environment and tools

In order to conduct the experiment, several tools are required. To produce n-way and random test suites we used a tool of our own design called the Test Vector Generator (TVG). The tool was developed in our laboratory and had been used in previous research. The TVG generates n-way test suites using our implementation of a well known *greedy* algorithm created by Cohen, et al [17]. The TVG also generates random combinatorial test suites of any size. The random test suites are generated using the `java.util.random` package to select randomly from the set of all possible combinatorial test cases, T_{cart}, without replacement.

To execute the test suites, a data-driven test automation platform was assembled. This platform allows automated execution of the test cases on all mutated versions of the system. The platform also evaluates the results of each test case and determines a pass/fail status by comparing its result to the result produced by the original copy of the system.

3.4 Test suite preparation

The first step in test suite preparation is the selection of test data values for each of the system's input variables. The same set of selected test data values is used in the creation of n-way and random test suites. There is no definitive source on the n-way testing process and the technique used to select test data values for use in n-way testing is left to the discretion of the tester. We chose to use equivalence partitioning and boundary value analysis to select the test data values [1, 5]. The technique is commonly used by testers in selecting test data values for

an input variable. It is a black-box technique that is subjective, in that different testers using the technique on the same testing problem will often produce different sets of test data values.

The author with the greatest domain experience with the applications selected the test data values of the experiment. The test data values selected reflect *positive* test cases only. Positive test cases apply valid data to the system; negative test cases apply invalid data to the system and expect an error message in response [5]. In any testing campaign, the testing of error messages is an important step; however, it is generally conducted separately from combinatorial testing because testing combinations of invalid test data is ineffective in many situations.

The selection of invalid test data values is especially problematic in n-way test suite creation. N-way test suites are constructed expecting that each test case will cover some number of n-tuple data combinations. If invalid test data values are used in test suite construction, execution of the invalid input may result in "short circuiting," or early termination of the process, leaving some test data values unprocessed. The result is that the coverage of n-tuples is reduced and the n-way test suite no longer covers 100% of the n-tuples. To avoid this problem, we selected only valid test data values, resulting in fewer test data values per variable than one might expect. This is particularly noticeable in DMAS test data because in addition to omitting invalid test data values, the system has a number of variables that have binary data values (e.g., CalcType = *area* or *height*, OutlierTest = *on* or *off*, Bracketing = *yes* or *no*). Table 3 lists the number of test data values selected per input variable for LAS and DMAS.

Table 3: Number of selected test data values

System	Input Variables	Values/Input Variable
LAS	19	3,3,4,2,3,2,3,2,2,2,2,2,3,3,3,3,3,3,4
DMAS	18	2,2,2,2,5,2,8,2,2,2,2,2,2,2,2,2,2,2

Once test data values have been selected, n-way and random test suites are created. For a given testing technique and selection of test data variables, there are many possible test suites of the same size. For example, there are many different random test suites of size 10. Similarly, there are many n-way test suites for any setting of *n*. To account for this, 10 test suites were generated for each n-way setting (i.e., n = 2, 3, 4) for a total of 30 n-way test suites. Due to the non-deterministic algorithm used by the TVG to create the n-way test suites, the suites are guaranteed to be different from each other. Also due to TVG's non-deterministic algorithm, the size of the n-way test suites varies. After generating the 10 n-way test suites, random test suites that match the size of each n-way test suite were generated.

For the set of test data values selected, the size of T_{cart} for DMAS is 2,621,440 test cases and for LAS is 120,932,352 test cases. The range of sizes of test suites generated by TVG for n-way testing techniques is presented in Table 4.

Table 4: N-way test suite sizes for DMAS and LAS

Testing Technique	Range of sizes for LAS	Range of sizes for DMAS
2-way (10 suites)	22-25	40-42
3-way (10 suites)	92-100	120-123
4-way (10 suites)	364-370	375-384

3.4 Test suite analysis

Here we present an analysis similar to the work previously describe by Dalal and Mallows [18] in which n-tuple coverage of random test suites is measured for several hypothetical systems. Recall that their results indicate a high percentage of n-tuple coverage by random test suites, as high as 90% in some cases.

Table 5 presents the average coverage of test data combinations achieved in the 10 random test suites for each n-way setting. For example, we know that the LAS 2-way test suites cover 100% of the pairs of test data combinations; 2Rand, the same-sized randomly selected test suites for LAS, on average, cover 94.5% of the test data combinations.

Table 5: Coverage of n-tuples by random test suites

System	2Rand	3Rand	4Rand
LAS	94.5%	97.66%	99.07%
DMAS	99.6%	99.99%	99.99%

The coverage values in Table 5 are higher than those reported by Dalal and Mallows. A contributing factor is small number of test data values per variable in the DMAS and LAS test data sets. We do not expect such high coverage in all cases. Based on this data, to the extent that faults are exposed by 2, 3, and 4 way combinations of test data values, we expect the FDE of the random and n-way test suite to be very similar. However, n-way coverage alone does not determine the FDE of a test suite. If it did, one might expect 100% n-way coverage to result in 100% fault detection. Given the RAX case, with an FDE as low as 50%, clearly this is not the case [9].

4. Experimental Planning

4.1 Variables

The experiment consists of independent variables, controlled variables, and dependent variables. The testing technique used, n-way or random, is the independent variable. The controlled variable is the size of the test suites, which is determined by the greedy algorithm used in n-way test suite creation. The dependent variable, the FDE of a test suite, is the percentage of total seeded faults detected. More formally, it can be defined as follows:

Let $F = \{f_i \mid 1 \le i \le n\}$ be a set of faults injected into a software system. For program P, an object of experimentation, for each $f_i \in F$, P_i' is created by seeding f_i into P. A set of test cases $S = \{s_i \mid 1 \le i \le n\}$ is said to detect f_i in P_i' if there exists a test case $s_i \in S$ such that the output of P_i' is different than the output of P when executed on s_i. *Fault detection effectiveness (FDE)* of S for a given P and F is:

$$\frac{\text{total } f_i \text{ detected by } S}{|F|} * 100$$

4.2 Hypothesis

The null hypothesis (NH) and alternative hypothesis (AH) can be stated in terms of μ_1 and μ_2, where μ_1 is the average fault detection effectiveness of the n-way technique and μ_2 is the average fault detection effectiveness for the equal sized random technique.

- H1: The FDE of 2-way testing technique is greater than the randomly generated test suites of the same size.
  NH: $\mu_1 <= \mu_2$ AH: $\mu_1 > \mu_2$
- H2: The FDE of 3-way testing technique is greater than the randomly generated test suites of the same size.
  NH: $\mu_1 <= \mu_2$ AH: $\mu_1 > \mu_2$
- H3: The FDE of 4-way testing technique is greater than the randomly generated test suites of the same size.
  NH: $\mu_1 <= \mu_2$ AH: $\mu_1 > \mu_2$

4.3 Design

A simple one factor two alternative experimental design is used to test the hypotheses. The dependent variable, FDE, is measured for each of the 10 test suites and is reported as the average of the 10 values. No humans are involved in the measurement of FDE, as the number of faults detected by each test suite comes from the automated test execution environment.

T-tests are used to verify the hypotheses. The commonly available statistical package (SPSS) is used to execute the t-tests. An independent samples t-test is conducted to evaluate the mean fault detection effectiveness of the n-way test suites and the same-sized random test suites, taking into consideration test for equality of variances (i.e., the F statistic).

4.4 Threats to validity

In this section, threats to internal and external validity of the study results are considered. Threats to internal validity include a possible bias in the faults injected, and in the selection of test data values. The selection of the faults to inject was based on the availability of a defined model with well defined software errors. Cleary, injecting other types of faults could result in a different outcome. We believe that bias in act of injecting faults to be low because the student injecting the faults knew little about combinatorial testing or about the software systems under test. The selection of test data values also threatens internal validity. The test data values any tester selects are based on many different factors, including: knowledge of the application domain, years of experience in testing, the perceived quality of the software under test, and the intended use of the data values (i.e., in many cases, selecting more data values, results in more testing work). Additional experimentation with other types of data selection, including random selection of values, is needed. A final threat to internal validity exists because the software systems used in the experiment were selected based on convenience. Additional systems on different platforms with different architectures must be investigated.

Threats to external validity affect the generality of the results. In the experiment, the TVG in used to generate n-way test suites using a documented *greedy* algorithm. These results cannot be generalized to other techniques for constructing n-way test suites, such as algebraic approaches and orthogonal arrays without additional work. The process for injecting faults into the software in the experiment does not match the process by which faults are injected into software on industrial development projects. Industrial projects often have multiple faults present in the software at any given time, covering a wide range of fault complexities. Finally, a general problem in software experimentation remains that there is no software classification scheme with enough detail to allow practitioners to understand if the systems used in the experiment are sufficiently similar to their own such that they should expect similar results.

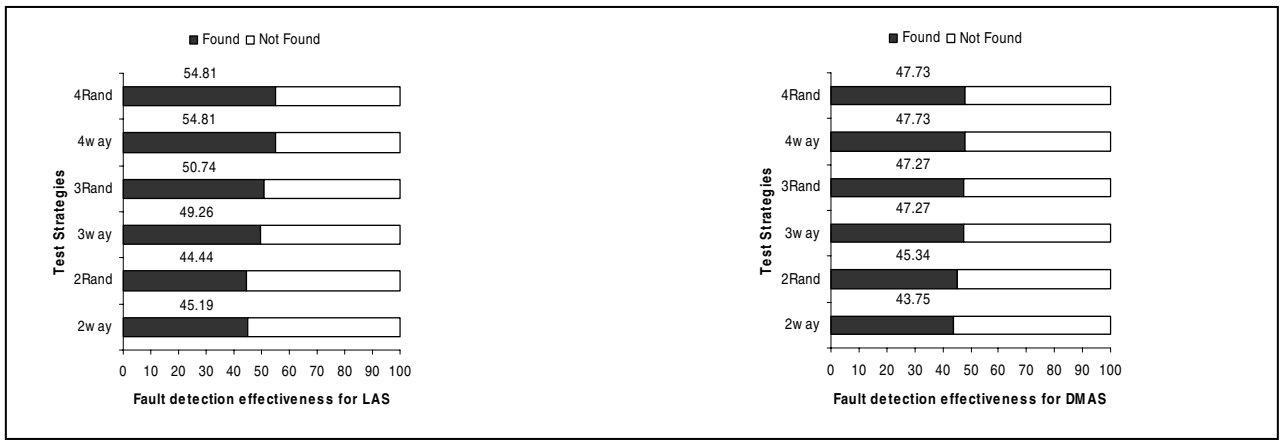

Figure 2. Average FDE of LAS and DMAS test suites

5. Experimental procedure

The section lists the procedure used to execute the experiment.

1. **Fault Injection**: A graduate student injects faults into the selected software system creating one mutant system for each fault injected.
2. **Test Data Value Selection**: A set of positive test data values is selected for each system input variable using equivalence partitioning and boundary value analysis (negative, error-producing test data values are not used in the study).
3. **Removal of Easily Detectable Faults**: Mutant systems that are likely detected by all test cases are removed from the study. This is accomplished by using 10 1-way test suites (size 4-5 test cases). Any mutant that is detected (i.e., at least one of the test cases fails) by all 10 test suites is removed form the study.
4. **Creation of N-Way Test Suites**: Using the test data values selected in step 2, the TVG tool generates 10 2-way test suites, 10 3-way test suites, and 10 4-way test suites.
5. **Creation of Same-Sized Random Test Suites**: The TVG tool uses the test data values selected in step 2 to create T_{cart}. From T_{cart}, TVG randomly selects a test suite that match the size of an n-way test suite generated in step 4. This is done for all n-way test suites.
6. **Creation of Oracle Results**: An "oracle" is a tool or technique for determining the correct, or expected, result for a test case [36]. In the experiment the original, un-mutated version of the system acts as the oracle. All test suites created in steps 4 and 5 are executed against the original system and the output is saved as the oracle results.
7. **Test Suite Execution and Evaluation**: All test suites created in steps 4 and 5 are executed against all mutant systems in the study. The output of the mutant system is compared to the oracle output created for the same test suite. If the mutant system output does not match the oracle output for any test case in the test suite, the test suite has detected the fault injected into the mutant. If the mutant system output matches the oracle output for every test case in the test suite, the test suite has not detected the fault.
8. **Calculation of Average FDE**: The FDE is calculated for each test suite in the study. The average FDE is calculated for the 10 test suites of each type.
9. **Statistical Analysis**: The t-test is used to compare the FDEs for each set of n-way test suites and same-sized random test suites.

6. Results

In this section, the results of the experiment are presented and average FDE for each technique is reported. Then the hypotheses are evaluated and the results are analyzed.

6.1 FDE of n-way and random test suites

The average FDE for n-way and random test suites is presented in Figure 2 . The y-axis represents the testing techniques. *2way* indicates the average FDE for 10 2-way test suites; *2Rand* indicates the average FDE for 10 random test suites that are the same size as the 2-way test suites. The results for 3-way and 4-way testing are similarly presented.

Figure 3 shows a box plot indicating the number of faults detected by the test suites for each testing technique. The upper and lower ends of the boxes represent the 75% quartile and 25% quartile of the number of faults detected.

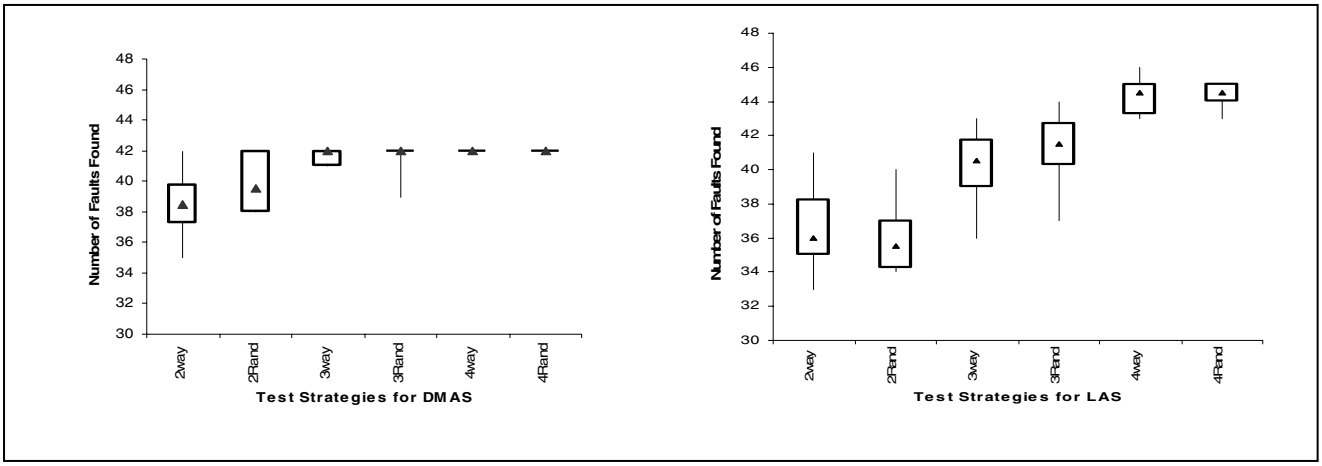

Figure 3. Box plot of fault found by test strategy for LAS and DMAS

The triangle represents the median (50% quartile) of the number of faults detected.

An independent samples t-test was conducted to evaluate the mean FDE of the n-way test suites and the same-sized random test suites. The results of t-test for LAS indicate that no significant difference was found in the FDE for n-way and random test suites at the 95% confidence level for H1, H2, and H3. The results of DMAS indicate that no significant difference was found in the FDE between n-way and random testing at the 95% confidence level for H1, H2, H3. In the case of 4way vs. 4Rand, the number of faults found by all 10 test suites for each technique was the same; therefore the t-test in this case cannot be evaluated because the standard deviation is zero.

6.2 Analysis

In this study we found no significant difference in the FDE of n-way and random combinatorial test suites. Given the similarity of the n-way and random test suites with respect to the n-tuples covered, the result is not unexpected.

In this study we injected what we considered to be "simple" faults, that is, we made no attempt to create complex system-level faults. Despite our simple approach, we found faults that require input combinations of size 4, 5, and above in order to be detected with certainty. Table 6 classifies faults based on the type of test suite that detected them.

A fault is classified as 2-way if it was detected by *all* 10 2-way test suites. Similarly, a fault classified as 3-way was detected by all 10 3-way test suites. Faults in the "> 4-way" category were found by some 4-way test suites, indicating that test data combinations of 5 or more values may be required to ensure they are detected. Faults in the "not found" category were not found for a variety of reasons: some faults are not found because they are in error-handling code that was not executed; some faults are 1-way and 2-way faults that were not found because we did not select test data values that could expose them; some faults are not found because they are triggered by higher-order data combinations that were not executed (e.g., 6-way faults).

Table 6. Fault classification for injected faults

Fault Type	LAS	DMAS
2-way	30	29
3-way	4	12
4-way	7	1
> 4-way	7	3
Not Found	34	43

Our ability to inject faults (or a developer's ability to commit an error) that requires a test suite of 4-way or above to be detected is dependent on the characteristics of the software. That is, to get a 4-way fault there must be outputs of the system that are determined by the data and control dependencies of 4 or more system inputs. Therefore, the FDE of a combinatorial test suite is dependent on:

1) The test data values selected for the system's input variables.
2) The combinations of test data values that are executed.
3) The characteristics of the software under test, including the degree to which input variables combine to create system outputs.

The n-way testing process addresses item 2, only. This makes interpretation of data from n-way experiments and case studies difficult. A study result that indicates that pairwise testing is effective may have been performed on a system for which all outputs are influenced by only 2 inputs.

If we incorporate the combinatorial characteristics of the software into our discussion, we can describe the conditions under which the FDE of n-way testing is likely to be higher than that of random testing. One condition would be that the random test suites cover fewer input combinations than the n-way test suites. As reported by Dalal and Mallows [18], one situation in which this occurs is in the creation of an n-way test suite with a small number of input variables (4 inputs). The second condition is that the software produces only outputs that are influenced by a small number of inputs, *or* that only faults detectable by a small number of input combinations are injected into the system. In our experiment, neither of these conditions was met.

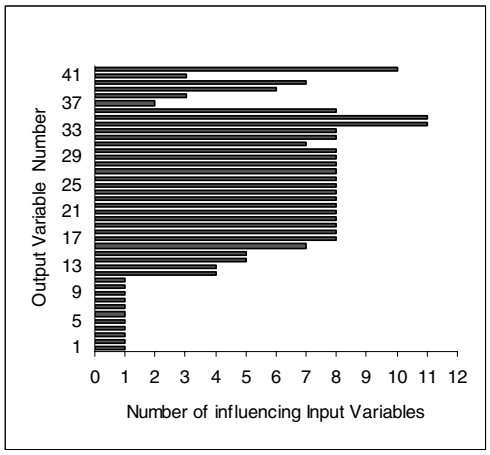

Figure 4. Number of inputs influencing an output for DMAS

Figure 4 presents the combinatorics of inputs and outputs for the DMAS system. The table is created by analyzing the system's input-output (IO) relationships. The IO relationships specify which system inputs influence the computation of each system output. Figure 4 displays the number of inputs that influence each output for DMAS. We gathered this information using IO analysis [29]; however, the information must also be determined by testers in calculating expected results for their test cases (i.e., to correctly determine an expected output of a test case one must know which inputs are used in creating the output).

In the systems we studied, the outputs are not uniformly influenced by 2, 3, or 4 inputs. The number of inputs that influence an output for DMAS ranged from 1-11; the number of inputs that influenced and output for LAS ranged form 1-13. In creating these systems, developers clearly have the opportunity to commit faults that require high-order combinatorial testing in order to be detected. More research is necessary, but practical advice to testers may be to analyze the combinatorics of the system under test before selecting a combinatorial testing strategy.

We agree with the testing heuristic that *most* faults are 1-way and 2-way faults, and our data supports this notion (especially if one includes the 1-way faults removed from the study). However, in complex industrial systems where outputs are influenced by many input variables, using pairwise testing to remove *most* of the faults may result in an FDE closer to the 50% experienced by Smith, et al. [9] rather than the 98% projected by Wallace and Kuhn [8].

7. Conclusion

Further experimentation in the form of controlled academic experiments and industrial case studies will be needed to confirm or refute the results of this experiment and further our understanding of the FDE of n-way and random testing.

N-way testing has become a popular combinatorial testing technique, yet little data exists on its effectiveness with respect to fault detection. In this paper we presented the results of a controlled study that compared the fault detection effectiveness of pairwise, 3-way and 4-way test suites to equally-sized random test suites. The results for the systems studied indicate no significant difference exists in the fault detection effectiveness of n-way and random testing. An analysis of the random test suites used in the study indicates that, from the perspective of the number of data combinations covered, the n-way and random test suites are nearly identical. This result confirms other hypothetical results which found coverage of input pairs in same-sized random and 2-way test suites to be similar in many cases [18]. Future work includes further experimentation with dissimilar n-way and random test suites, experimentation using different types of software systems, and work with alternative test data value selection techniques.

8. References

[1] C. Kaner, J. L. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed. New York: Van Nostrand Reinhold, 1993.

[2] E. Dustin, "Orthogonally Speaking," *STQE*, vol. 3, no. 5, pp. 46-51,September/October, 2001.

[3] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston, MA: Artech House Publishers, 2003.

[4] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context Driven Approach*. New York: John Wiley & Sons, Inc., 2002.

[5] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Boston: Artech House, 2002.

[6] S. Splaine and S. P. Jaskiel, *The Web Testing Handbook*. Orange Park, FL: STQE Publishing, 2001.

[7] J. D. McGregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*. Boston, MA: Addison-Wesley, 2001.
[8] D. R. Wallace and D. R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," *Int'l Jour. of Reliability, Quality and Safety Engineering*, vol. 8, no. 4, pp. 351-371, 2001.
[9] B. Smith, M. S. Feather, and N. Muscettola, "Challenges and Methods in Testing the Remote Agent Planner," in *Proc. 5th Int'l Conf. on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 2000, pp. 254-263.
[10] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.
[11] A. L. Williams and R. L. Probert, "A practical strategy for testing pairwise coverage at network interfaces," in *Proc. IEEE ISSRE: Int'l Symp. Softw. Reliability Eng*. White Plains, NY, 1996.
[12] T. Berling and P. Runeson, "Efficient Evaluation of Multifactor Dependent System Performance Using Fractional Factorial Design," *IEEE Trans. on Software Engineering*, vol. 29, no. 9, pp. 769-781, Sept. 2003.
[13] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Communication of the ACM*, vol. 28, no. 10, pp. 1054-1058, 1985.
[14] R. Brownlie, J. Prowse, and M. Phadke, "Robust Testing of AT&T PMX/StarMail Using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41-47, 1992.
[15] K. Burroughs, A. Jain, and R. L. Erickson, "Improved Quality of Protocol Testing Through Techniques of Experimental Design," in *Proc. Supercomm./IEEE International Conference on Communications*, 1994, pp. 745-752.
[16] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," in *Proceedings of the Nineteenth International Conference on Software Engineering*. Boston, MA, 1997, pp. 205-215.
[17] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437-444, 1997.
[18] S. Dalal and C. L. Mallows, "Factor-Covering Designs for Testing Software," *Technometrics*, vol. 50, no. 3, pp. 234-243, 1998.
[19] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," in *Proceedings of the International Conference on Software Engineering*. Los Angeles, CA, 1999, pp. 285-294.
[20] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software*, vol. 13, no. 5, pp. 83-88, 1996.
[21] L. J. White, "Regression Testing of GUI Event Interactions," in *Proceedings of the International Conference on Software Maintenance*. Washington, 1996, pp. 350-358.
[22] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya, "Automatic Test Generation using Checkpoint Encoding and Antirandom Testing," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*. Albuquerque, NM, 1997, pp. 84-95.
[23] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness," in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*. Monterey, CA, 1994, pp. 230-238.
[24] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton, "The Automatic Efficient Test Generator (AETG) System," in *Proceedings of the Fifth International Symposium on Software Reliability Engineering*: IEEE Computer Society Press, 1994, pp. 303-309.
[25] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott, "Model-Based Testing of a Highly Programmable System," in *Proceedings of the Nineth International Symposium on Software Reliability Engineering (ISSRE 98)*. Paderborn, Germany, 1998, pp. 174-178.
[26] V. R. Basili, G. Cladiera, and D. H. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, vol. 1, J. J. Marciniak, Ed. New York City: John Wiley and Sons, 1994, pp. 528-532.
[27] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test Selection Based on Finite-State Models," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, 1991.
[28] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 178-187, 1978.
[29] P. J. Schroeder, B. Korel, and P. Faherty, "Generating Expected Results for Automated Black-Box Testing," in *Proc. of the Int'l Conf. on Automated Software Engineering (ASE2002)*. Edinburgh, UK, 2002, pp. 139-48.
[30] S. L. Pfleeger, *Software Engineering: Theory and Practice*, 2 ed. Upper Saddle River, NJ: Prentice-Hall, Inc., 2001.
[31] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Trans. on Software Engineering*, vol. 28, no. 2, pp. 159-182, 2002.
[32] J. Offutt and J. Hayes, "A Semantic Model of Program Faults," in *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA 96)*. San Diego, CA: ACM Press, 1996, pp. 195-200.
[33] R. A. DeMillo, "Testing Adequacy and Program Mutation," presented at Eleventh International Conference on Software Engineering, Washington, DC, 1989.
[34] R. A. DeMillo, "Mutation Analysis as a Tool for Software Quality Assurance," presented at Fourth International Computer Software & Applications Conference, New York, NY, USA, 1980.
[35] A. J. Offut, "Investigations of the Software Testing Coupling Effect," *ACM Transactions of Software Engineering and Methodology*, vol. 1, no. 1, pp. 5-20, 1992.
[36] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.