An Evaluation of Combination Strategies for Test Case Selection

Mats Grindal*, Birgitta Lindström*, Jeff Offutt†, and Sten F. Andler*

2006-10-06

Abstract

This paper presents results from a comparative evaluation of five combination strategies. Combination strategies are test case selection methods that combine "interesting" values of the input parameters of a test subject to form test cases. This research comparatively evaluated five combination strategies; the All Combination strategy (AC), the Each Choice strategy (EC), the Base Choice strategy (BC), Orthogonal Arrays (OA) and the algorithm from the Automatic Efficient Test Generator (AETG). AC satisfies n-wise coverage, EC and BC satisfy 1-wise coverage, and OA and AETG satisfy pair-wise coverage. The All Combinations strategy was used as a "gold standard" strategy; it subsumes the others but is usually too expensive for practical use. The others were used in an experiment that used five programs seeded with 128 faults.

The combination strategies were evaluated with respect to the number of test cases, the number of faults found, failure size, and number of decisions covered.

The strategy that requires the least number of tests, Each Choice, found the smallest number of faults. Although the Base Choice strategy requires fewer test cases than Orthogonal Arrays and AETG, it found as many faults. Analysis also shows some properties of the combination strategies that appear significant. The two most important results are that the Each Choice strategy is unpredictable in terms of which faults will be revealed, possibly indicating that faults are found by chance, and that the Base Choice and the pair-wise combination strategies to some extent target different types of faults.

Keywords: Combination Strategies, Orthogonal Arrays, AETG, Test Case Selection, Testing Experiment

*School of Humanities and Informatics, University of Skövde, email: {mats.grindal,birgitta.lindstrom,sten.f.andler}@his.se

†Department of Information and Software Engineering, George Mason University, Fairfax, VA 22030, USA, part-time faculty researcher at the National Institute of Standards and Technology, email: offutt@ise.gmu.edu

1 Introduction

The input space of a test problem can be generally described by the parameters of the test subject [1]. Often, the number of parameters and the possible values of each parameter result in too many combinations to be useful. *Combination strategies* is a class of test case selection methods that use combinatorial strategies to select test sets that have reasonable size.

The combination strategy approach consists of two broad steps. In the first step, the tester analyzes each parameter in isolation to identify a small set of "interesting" values for each parameter. The term *interesting* may seem insufficiently precise and a little judgmental, but it is common in the literature. In order not to limit the use of combination strategies, this paper defines "interesting values" to be whatever values the tester decides to use. In the second step, the combination strategy is used to select a subset of all combinations of the interesting values based on some coverage criterion.

Some papers have discussed the effectiveness of combination strategies (Brownlie, Prowse & Phadke 1992, Cohen, Dalal, Fredman & Patton 1997, Kropp, Koopman & Siewiorek 1998). These papers indicate the usefulness of combination strategies, but many questions still remain. For instance, few papers compare the different combination strategies. Thus, it is difficult to decide which combination strategy to use. This paper presents results from an experimental comparative evaluation of five combination strategies.

Our previous paper surveyed over a dozen different combination strategies (Grindal, Offutt & Andler 2005). Five of these were chosen for this study. In this paper, a *criterion* is a rule for selecting combinations of values, and a *strategy* is a procedure for selecting values that satisfy a criterion.

The 1-wise coverage criterion requires each interesting value of each parameter to be represented at least once in the test suite. The Each Choice strategy satisfies 1-wise coverage directly by requiring that each interesting value be used in at least one test case. The Base Choice strategy satisfies 1-wise coverage by having the tester declare a "base" value for each parameter, a base test case that includes all base values, and then varies each parameter through the other values.

The pair-wise coverage criterion requires every possible pair of interesting values of any two parameters be included in the test suite. The Orthogonal Arrays and Automatic Efficient Test Generator strategies use different algorithms (described later) to satisfy pair-wise coverage.

The All Combinations strategy generates all possible combinations of interesting values of the input parameters. Thus, AC subsumes all the other strategies and requires the most test cases. It is considered to satisfy the n-wise criterion, where n is the number of parameters. AC is used as a "gold-standard" reference. It is usually too expensive for practical use, but provides a convenient reference for experimental comparison.

This paper evaluates and compares the combination strategies with respect to the number of test cases, number of faults found, failure size, and number of decisions covered in an experiment comprising five programs that were seeded with 128 faults.

Two joint combination strategies (BC+OA and BC+AETG) were created by taking the unions of the respective test suites and these are also evaluated on the same grounds as the others.

[1] In testing the object being tested is often called "program under test" or "system under test." However, since this paper reports the results of an experiment, it uses the term *test subject* instead.

That is, the independent variable in the experiment is the selection strategy, and it has seven values. There are four separate dependent variables, test cases, faults, decisions, and failure size.

The remainder of this paper presents our experimental procedure and results. Section 2 gives a more formal background to testing, test case selection methods, and how they are related to the combination strategies evaluated in this work. Section 3 describes each combination strategy that was investigated. Section 4 contains the details of the experiments conducted, section 5 describes the results of the experiment, and section 6 analyzes the results achieved. This analysis leads to the formulation of some recommendations about which combination strategies to use. In section 7 the work presented in this paper is contrasted with the work of others and future work is outlined in section 8.

2 Background

Testing and other fault revealing activities are crucial to the success of a software project (Basili & Selby 1987). A *fault* in the general sense is the adjudged or hypothesized cause of an error (Anderson, Avizienis, Carter, Costes, Christian, Koga, Kopetz, Lala, Laprie, Meyer, Randell & Robinson 1994). Further, an *error* is the part of the system state that is liable to lead to a subsequent failure (Anderson et al. 1994). Finally, a *failure* is a deviation of the delivered service from fulfilling the system function.

These definitions are similar to typical definitions in empirical software testing research. In particular, the number of faults found is a common way to evaluate test case selection methods (Briand & Pfahl 1999, Ntafos 1984, Offutt, Xiong & Liu 1999, Zweben & Heym 1992). These definitions are also consistent with the definitions used in the paper that first described the test subjects used in this experiment (Lott & Rombach 1996).

A *test case selection method* is a way to identify test cases according to a selection criterion. Most test case selection methods try to cover some aspect of the test subject, assuming that coverage will lead to fault detection. With different test case selection methods having different coverage criteria, an important question for both the practitioner and the researcher is: Given a specific test problem, which test case selection methods should be used? In the general test problem some properties of a software artefact should be tested to a sufficient level. Thus, the choice of test case selection method depends on the properties that should be tested, the associated coverage criteria of the test case selection method, and the types of faults that the test case selection method targets.

With focus on combination strategies, this research targets the question of finding a suitable test case selection method to use.

A common view among researchers is that experimentation is a good way to advance the common knowledge of software engineering (Lott & Rombach 1996, Harman, Hierons, Holocombe, Jones, Reid, Roper & Woodward 1999). For results to be useful it is important that the experiments are controlled and documented in such a way that the results can be reproduced (Basili, Shull & Lanubile 1999). Thus, a complete description of the work presented here can be found in a technical report (Grindal, Lindström, Offutt & Andler 2003).

	Parameter		
test case	P_1	P_2	P_3
1	1	1	1
2	2	2	2
3	3	1	2

Figure 1: EC test suite for the example test problem.

3 Combination Strategies

A prerequisite for all combination strategies is the creation of an input parameter model of the test subject. The input parameter model represents the parameters of the test subject with a set of interesting values for each parameter. The tester may select all or some values to be interesting, she may use a test case selection method such as Equivalence Partitioning (Myers 1979) or Boundary Value Analysis (Myers 1979), or she may use an operational profile (Musa 1993) if one is available. To avoid limiting the combination strategies, this paper defines "interesting" to be whatever values the tester decides to use.

The five combination strategies investigated in this study are Each-Choice (EC), Base Choice (BC), Orthogonal Arrays (OA), the strategy used by the tool Automatic Efficient Test Generator (AETG), and All-Combinations (AC). The main reason to choose these five combination strategies is to get diversity with respect to coverage levels. These five represent four different levels of coverage. OA and AETG both generate test suites with 100% pair-wise coverage, but their fundamentally different approaches to reach this goal motivates the inclusion of both.

AC requires every combination of values to be covered. Due to the number of test cases required for AC it has been excluded from the experimental part of this study and is only used as a reference in terms of the number of test cases for the different test subjects in the study.

To illustrate the strategies and their associated coverage levels, the next few subsections use a running example that has three parameters. Parameter P_1 has three interesting values, 1, 2 and 3, P_2 has two values, 1 and 2, and P_3 has two values, 1 and 2.

3.1 Each Choice (EC)

The *Each Choice (EC)* combination strategy requires each value of each parameter to be included in at least one test case (Ammann & Offutt 1994). This is also the definition of 1-wise coverage.

Figure 1 shows a test suite generated by the EC combination strategy. Test cases are identified by combining the next unused value of each parameter. In the third test case, there are no unused values for parameters P_2 and P_3, thus repeated values are used.

An important property of combination strategies is the number of test cases required to satisfy the associated coverage criterion. Let a test subject be represented by an input parameter model with N parameters $P_1, P_2, ..., P_N$, where parameter P_i has V_i values. Then, a test suite that satisfies 1-wise coverage must have at least $Max_{i=1}^{N} V_i$ test cases.

	Parameter		
test case	P_1	P_2	P_3
1 (base test case)	1	1	2
2	2	1	2
3	3	1	2
4	1	2	2
5	1	1	1

Figure 2: BC test suite for the example test problem.

3.2 Base Choice (BC)

The algorithm for the *Base Choice (BC)* combination strategy (Ammann & Offutt 1994) starts by identifying one base test case. The *base test case* may be determined by any criterion, including simplest, smallest, or first. A criterion suggested by Ammann and Offutt is the "most likely value" from an end-user point of view. This value could be determined by the tester or based on an operational profile if one exists.

From the base test case, new test cases are created by varying the interesting values of one parameter at a time, keeping the values of the other parameters fixed on the base test case. If the base test case is $[1, 1, 2]$, figure 2 shows the resulting test suite for the example test problem.

A test suite that satisfies base choice coverage will have at least $1 + \sum_{i=1}^{N}(V_i - 1)$ test cases, where N is the number of parameters and parameter P_i has V_i values in the input parameter model. Note that this test suite is larger than the corresponding EC test suite.

Base Choice includes each value of every parameter in at least one test case, so it satisfies 1-wise coverage. The semantic information of the base choice also affects the values chosen. Assume that the values of the different parameters can be classified as either normal or error values. *Normal* values are input values that will cause the test subject to perform some of its intended functions. *Error* values are values outside the scope of the normal working of the test subject. If the base test case contains only normal values, the test suite will, in addition to satisfying 1-wise coverage, also satisfy a criterion that we call single error coverage. *Single error coverage* requires that for each error value in the input parameter model, there is at least one test case that combines that value with a normal value from each of the other parameters. Single error coverage will be satisfied if the base choice is based on the most likely value, as mentioned above.

3.3 Orthogonal Arrays (OA)

The *Orthogonal Arrays (OA)* combination strategy is based on a mathematical concept with the same name. A Latin Square is an $N \times N$ square filled with symbols such that each symbol occurs exactly once in each row and each column. An Orthogonal Array combines two or more orthogonal Latin Squares. Orthogonal arrays have been used in the design of scientific experiments and was first applied to testing by Mandl (Mandl 1985). Williams and Probert (Williams & Probert 1996) described how the OA strategy can be used in testing. The test suite identified by the OA strategy

	Parameter		
test case	P_1	P_2	P_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1
5	3	1	2
6	3	2	1

Figure 3: OA test suite for the example test problem.

satisfies pair-wise (2-wise) coverage, which means that each pair of parameter values of any two parameters is included in at least one test case.

Based on the input parameter model, the tester identifies an orthogonal array that is large enough to handle the parameter with the most values. Each position in the orthogonal array then represents a test case.

For the example test problem, a single 3×3 Latin Square containing the symbols $1, 2$ and 3 is large enough. Let <*row*, *column*, *contents*> describe the resulting nine positions. To create test cases, the three parameters P_1, P_2 and P_3 and their values are mapped onto these positions in that order. If one parameter has more values than the others (as P_1 does in the example), the other parameters have one or more undefined values. Any defined value, 1 or 2 in this example, can be used without destroying the pair-wise coverage property. Sometimes a new test case is a duplicate of another test case in the test suite. In such cases the duplicate can be removed, reducing the size of the test suite. Figure 3 shows the final test suite for the example test problem identified by OA. In this example three test cases were removed because of duplication.

An advantage that OA has is that Orthogonal Arrays of different sizes can be precalculated. Thus, the tester only needs to find the right size orthogonal array, describe the mapping between parameters and indices, and possibly perform some test reduction. A drawback is that orthogonal arrays do not exist for all test problem sizes.

The number of test cases generated by the orthogonal arrays combination strategy without reducing duplicates is V_i^2, where $V_i = Max_{j=1}^{N} V_j$, N is the number of parameters, and parameter P_i has V_i values. However, if duplicates are removed, the number of test cases approaches the product of the number of values of the two largest parameters.

Williams and Probert (Williams & Probert 1996) give further details on how test cases are created from orthogonal arrays.

3.4 Automatic Efficient Test Generator (AETG)

The *Automatic Efficient Test Generator (AETG)* system was first described in 1994 by Cohen et al. (Cohen, Dalal, Kajla & Patton 1994). It contains a heuristic algorithm for generating a test suite that satisfies pair-wise coverage, the same criterion that OA satisfies. AETG and its algorithm is described in detail Cohen et al.'s 1997 paper (Cohen et al. 1997).

	Parameter		
test case	P_1	P_2	P_3
1	1	1	1
2	2	2	2
3	3	1	2
4	3	2	1
5	2	1	1
6	1	2	2

Figure 4: AETG test suite for the example test problem.

The AETG algorithm finds one new test case per iteration, attempting to find a test case that maximizes the increase in pair-wise coverage. Several test case candidates are identified and evaluated. The algorithm terminates when test cases have been found to satisfy 100% pair-wise coverage.

The number of test cases needed to reach pair-wise coverage is impossible to calculate in advance due to the heuristic nature of the algorithm. It is, among other things, related to the number of candidates generated for each test case. In general, the more candidates, the fewer test cases in the final test suite. However, Cohen et al. (Cohen, Dalal, Parelius & Patton 1996) report that using values higher than 50 will not dramatically decrease the number of test cases. Thus, in our experiment, we followed this advice and generated 50 candidates for each test case.

Figure 4 shows a test suite for the example test problem identified by the AETG algorithm. Although OA and AETG have selected different test cases for their test suites, both test suites satisfy pair-wise coverage.

The main advantage of AETG over OA is that AETG can handle test problems of any size. Another advantage is that a preselected test suite, identified by some other means, may be used as a starting point for AETG. A potential drawback is that it is not possible to calculate, in advance, an upper limit to the number of test cases.

3.5 All Combinations (AC)

All-combinations (AC) requires that every combination of values of each parameter be used in a test case (Ammann & Offutt 1994). This is also the definition of n-wise coverage (Grindal et al. 2005).

Given the example test problem, the final test suite will contain all 12 possible combinations of the parameter values. In the general case a test suite that satisfies n-wise coverage will have $\prod_{i=1}^{N} V_i$ test cases, where N is the number of parameters of the input parameter model and parameter P_i has V_i values.

Goal	Purpose	Compare
	Issue	five combination strategies with respect to
	Object	their effectiveness and efficiency
	Viewpoint	from the viewpoint of the practicing tester
	Context	in the context of a faulty program benchmark suite.
Question		How effective is a generated test suite?
Metric		a) Number of found faults b) Code coverage
Question		Do the combination strategies find different types of faults?
Metric		c) Types of found faults d) Failure size
Question		What is the cost of using a combination strategy?
Metric		e) Number of test cases

Table 1: The goal of this study and its refinement into metrics through questions.

4 Experimental Setting

Both Harman et al. (Harman et al. 1999) and Miller et al. (Miller, Roper, Wood & Brooks 1995), recommend that enough detail should be published about an experiment to allow replication. To support this goal, we followed the GQM (Goal, Questions, Metrics) method (Basili & Rombach 1988) to define the goals and their refinement into concrete metrics. The use of the GQM method is supported by Lott and Rombach (Lott & Rombach 1996) from which the test subjects used in our experiments have been borrowed. Table 1 shows the goal, questions, and metrics of this study.

The following sections describe and motivate the implementation details of this experiment.

4.1 Evaluation Metrics

Frankl et al. state that the goal of testing is either to measure or increase the reliability (Frankl, Hamlet, Littlewood & Stringini 1998). Testing to **measure** reliability is usually based on statistical methods and operational profiles, while testing to **increase** reliability relies on test case selection methods. These are assumed to generate test cases, which are good at revealing failures. The test techniques in this research were invented to reveal failures. At first glance, the number of failures revealed could be used to evaluate effectiveness. However, a test suite that reveals X failures that result from the same fault can be argued to be less effective than a test suite that reveals X failures that result from X different faults. Thus, the number of faults found is more useful.

This measure does not work if a program has no faults. Thus, code coverage is also used to assess the effectiveness of the generated test suites. Code coverage has been used in previous investigations of combination strategies, e.g., (Dunietz, Ehrlich, Szablak, Mallows & Iannino 1997, Piwowarski, Ohba & Caruso 1993, Cohen et al. 1996, Burr & Young 1998). In this experiment, branch coverage was measured on fault-free versions of each program.

The faults are classified into types to help analyze differences in the fault-finding abilities of the combination strategies. Two classification schemes are used. The first is based on the number of parameters involved in revealing the fault. The second is based on whether valid or invalid values are needed to reveal the fault.

During the analysis of the faults found by the different strategies, BC seemed to behave differently than the other strategies. The notion of failure size was introduced to understand this better. *Failure size* is defined as the percentage of test cases in a test suite that fail for a specific fault. Failure size was inspired by the notion of fault size. *Fault size* is the percentage of inputs that will trigger a fault, causing a failure (Bache 1997, Woodward & Al-Khanjari 2000, Offutt & Hayes 1996). In this experiment, the input parameter models are kept constant for all combination strategies, thus the fault sizes will be the same for all combination strategies.

In practice, the efficiency of a test selection method is related to the resources used, primarily time and money (Archibald 1992). Realistic resource consumption models of either of these resources are difficult to both create and validate. Variation in the cost of computers, variation in human ability, and the increasing performance of computers are just some of the factors that make it difficult. Nevertheless, some researchers have used "normalized utilized person-time" as a measure of the efficiency of a test case selection method (So, Cha, Shimeall & Kwon 2002).

This experiment uses a simplified model of resource consumption, in which the efficiency is approximated with the number of test cases in each test suite. The motivation for this is that the same input parameter models are used by all combination strategies. Further, it is assumed that differences in time to generate the test suites are small compared to the time it takes to define expected results and execute the test cases.

4.2 Test Subjects

The main requirements on the test subjects are: (1) specifications for deriving test cases must exist; (2) an implementation must exist; and (3) some known and documented faults must exist.

New test subjects can be created or already existing test subjects can be found. Creating new test subjects has the advantage of total control over the properties of the test subjects, e.g., size, type of application, types of faults, source code language, etc. Drawbacks are that it takes longer and creating programs in-house can introduce bias. A search on the Internet for existing test subjects gave only one hit; the experimental package containing a "Repeatable Software Experiment" including several different test objects by Kamsties and Lott (Lott n.d., Kamsties & Lott 1995b).

Using already existing test subjects also creates the possibility of cross-experiment comparisons (Basili et al. 1999). Thus, we used already existing test subjects.

The six programs were designed to be similar to Unix commands[2] and initially used in an experiment that compared defect revealing mechanisms (Lott & Rombach 1996). This experimental package was inspired by Basili and Selby (Basili & Selby 1987). The benchmark program suite has been used in two independent experiments by Kamsties and Lott (Kamsties & Lott 1995a, Kamsties & Lott 1995b), and later used in a replicated experiment by Wood et al. (Wood, Roper, Brooks & Miller 1997).

[2]The complete documentation of the Repeatable Software Experiment may be retrieved from URL: www.chrislott.org/work/exp/.

Test Subject	# Functions	Lines of Code	# Decisions (a,c,f,i,w)	# Global Vars	Nesting Level
count	1	42	$(0, 0, 0, 6, 2)$	0	4
tokens	5	117	$(2, 1, 4, 12, 3)$	4	3
series	1	76	$(0, 1, 1, 8, 0)$	5	2
nametbl	15	215	$(4, 0, 0, 17, 0)$	5	3
ntree	9	193	$(21, 0, 3, 10, 0)$	0	3

Table 2: Descriptive size metrics of the programs. The number of decision points are divided into assert ('a'), case ('c'), for ('f'), if ('i'), and while ('w') statements.

The programs are well documented, include specifications, and come with descriptions of existing faults. We were able to use five of the six programs. The first, **count**, implements the standard Unix command "wc." count takes zero or more files as input and returns the number of characters, words, and lines in the files. If no file is given as argument, count reads from standard input. Words are assumed to be separated by one or more white-spaces (space, tab, or line break).

The second program, **tokens**, reads from the standard input, counts all alphanumeric tokens, and prints their counts in increasing lexicographic order. Several flags can be used to control which tokens should be counted.

The third program, **series**, requires a start and an end argument and prints all real numbers between them in steps defined by an optional step size argument.

The fourth program, **nametbl**, reads commands from a file and runs one of several functions based on the command. Considered together, the functions implement a symbol table. For each symbol, the symbol table stores its name, the object type of the symbol, and the resource type of the symbol. The commands let the user insert a new symbol, enter the object type, enter the resource type, search for a symbol, and print the entire symbol table.

The fifth program, **ntree**, also reads commands from a file and runs run one of several functions based on the command. The functions implement a tree in which each node can have any number of child nodes. Each node in the tree contains a key and a content. The commands let the user add a root, add a child, search for a node, check if two nodes are siblings, and print the tree.

The sixth program, **cmdline**, did not contain enough details in the specification to fit the experimental set-up. Thus it was not used.

Table 2 contains some summary statistics of the five programs. All were implemented in the C programming language. We included `main()` in the number of functions. The counts of lines of code exclude any blank lines and lines that only contain comments. The number of decisions includes assert, case, for, if, and while statements. A case statement is counted as one decision. For global variables, each component of a struct is counted separately. Finally, the nesting level is the highest nesting level for any function in the program.

4.3 Test Subject Parameters and Values

One of the most important conceptual tasks in using combination strategies is creating the input parameter model. In their category partition method (Ostrand & Balcer 1988), Ostrand and Balcer discuss several approaches beyond the obvious idea of using the input parameters. Yin, Lebne-Dengel, and Malaiya (Yin, Lebne-Dengel & Malaiya 1997) suggest dividing the problem space into sub-domains that can be thought of as consisting of orthogonal dimensions that do not necessarily map one-to-one onto the actual input parameters of the implementation. Similarly, Cohen, Dalal, Parelius, and Patton (Cohen et al. 1996) suggest modeling the system's functionality instead of its interface.

In this experiment, input parameter models were defined based on functionality represented by "abstract parameters", such as the number of arguments, and how many of the same tokens are used. Equivalence partitioning (Myers 1979) was applied to these parameters and representative values were picked from each equivalence class. To support base choice testing, one equivalence class for each parameter was picked as the base choice class of that parameter. This made the corresponding value the base value for that parameter. The values selected as base choices for each parameter are all normal values.

Conflicts among parameters occur when some value of one parameter cannot be used with one or more values of another parameter. An example of a conflict is when a value for one parameter requires that all flags should be used, but values for another parameter states that a certain flag should be turned off. AETG is the only strategy in this experiment that has built-in function to handle conflicts. Ammann and Offutt (Ammann & Offutt 1994) suggested an outline for parameter conflict handling within BC that could also be used for EC. Williams and Probert (Williams & Probert 1996) suggested a way to handle parameter value conflicts within OA. However, Williams and Probert's method required the input parameter model to be changed. Since this would make comparisons between the combination strategies less reliable, handling conflicts would introduce a confounding variable into the experiment, thus all input parameter models were designed to be conflict free. If no complete conflict-free input parameter model could be designed, it was decided that some aspects of the test subject should be ignored to keep the input parameter model conflict-free, even at the cost of possibly loosing the ability to detect some faults. This was judged to be more fair for this experiment. Obviously, in actual testing, parameter value conflicts must be handled, and ongoing studies include a closer look at different ways of handling parameter conflicts.

Tables 3 through 7 show the input parameter models for each test subject. As mentioned earlier, these models were influenced by the equivalence partitioning technique. To aid the reader, the tables contain both a description of the equivalence classes and the actual values selected from each class. The test case sets generated by the combination strategies are quite extensive and can be found in an appendix in the technical report (Grindal et al. 2003).

4.4 Faults

The five programs in the study came with 33 known faults. The combination strategies exhibited only small differences in fault revealing for these faults, so an additional 118 faults were created and seeded by hand. Most of these faults were mutation-like, such as changing operators in

I	# files	min # words/row	min # chars/word	consecutive # WS	type of WS	# line feeds
0	**1**	0	1	1	**space**	0
1	$> 1(2)$	1	$> \mathbf{1}(\mathbf{4})$	$> \mathbf{1}(\mathbf{2})$	tab	1
2	-	$> \mathbf{1}(\mathbf{2})$	-	-	both	$> \mathbf{1}(\mathbf{2})$

Table 3: Input parameter model for **count**. Each column represents the equivalence classes of one parameter. Preselected values are in parentheses and base choices are shown in bold.

I	flags				No. of different tokens	No. of same tokens	numbers in tokens	upper and lower case
	-a	-i	-c	-m				
0	**No**	**No**	**0**	**No**	1	1	No	No
1	Yes	Yes	1	0	2	2	**Yes**	**Yes**
2	-	-	$> 1(4)$	1	$> \mathbf{2}(\mathbf{3})$	$> \mathbf{2}(\mathbf{5})$	-	-
3	-	-	-	$> 1(3)$	-	-	-	-

Table 4: Input parameter model for **tokens**. Each column represents the equivalence classes of one parameter. Preselected values are in parentheses and base choices are shown in bold.

I	start	end	step size
0	$< 0(-10)$	$< 0(-5)$	no
1	0	0	$< 0(-1)$
2	$> \mathbf{0}(\mathbf{15})$	$> \mathbf{0}(\mathbf{5})$	0
3	real (5.5)	real (7.5)	1
4	non-number (abc)	non-number (def)	$> \mathbf{1}(\mathbf{2})$
5	-	-	real (1.5)
6	-	-	non-number (ghi)

Table 5: Input parameter model for **series**. Each column represents the equivalence classes of one parameter. Preselected values are in parentheses and base choices are shown in bold.

I	INS	TOT	TRT	SCH
0	No instance	No instance	No instance	No instance
1	One instance	One instance	One instance	**One instance**
2	$>$ **one(4)instances**	$>$ **one(2)instances**	$>$ **one(3)instances**	>one (2) instances
3	Incorrect spelling	Too few args.	Too few args.	Too few args.
4	Too few args.	Too many args.	Too many args.	Too many args.
5	Too many args.	Incorrect obj. type	Incorrect obj. type	-
6	Same symbol twice	Unknown obj. type	Unknown obj. type	-

Table 6: Input parameter model for **nametbl**. Each column represents the equivalence classes of one parameter. Preselected values are in parentheses and base choices are shown in bold.

I	ROOT	CHILD	SEARCH	SIBS
0	No instance	No instance	No instance	No instance
1	**One instance**	One instance	**One instance**	**One instance**
2	Two instances	Two instances	Two instances	Two instances
3	Incorrect spelling	$>$ **two(5)instances**	Incorrect spelling	Incorrect spelling
4	Too few args.	Incorrect spelling	Too few args.	Too few args.
5	Too many args.	Too few args.	Too many args.	Too many args.
6	-	Too many args.	>one (2) hits	Not siblings
7	-	No father node	-	-
8	-	Two father nodes	-	-

Table 7: Input parameter model for **ntree**. Each column represents the equivalence classes of one parameter. Preselected values are in parentheses and base choices are shown in bold.

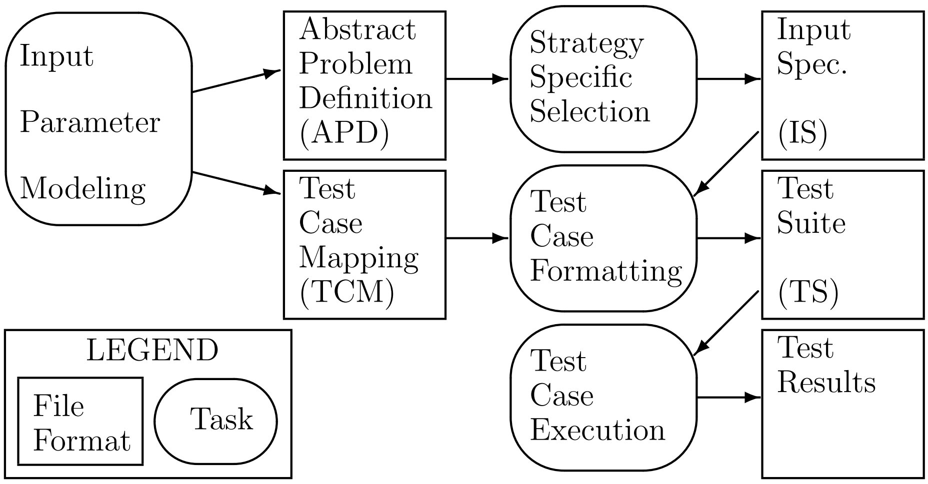

Figure 5: Test Case Generation and Execution

decisions, changing orders in enumerated types, and turning post-increment into pre-increment. Each function of every program contains at least one fault.

23 of the 151 faults were functionally equivalent to their original program. These were removed from the experiment, leaving a total of 128 faults.

Each fault was seeded in a separate copy of the program. This has the dual advantage of avoiding interactions among the faults and making it obvious which fault is found when a failure has occurred. This in turn made it easier to automate the experiment by using the Unix command "diff" to find the failures.

Next, simple instrumentation was added into the programs to measure code coverage. Faults were **not** put into these extra statements and care was taken to ensure that they did not change the programs' functionalities. A complete description of the faults used in this study is given in the technical report (Grindal et al. 2003).

4.5 Infrastructure for Test Case Generation and Execution

Figure 5 summarizes the experiment tasks and the intermediate representations of the information between each pair of tasks. Both test case generation and test case execution were automated as much as possible. Before the experiment started, the test subjects were instrumented for code coverage and seeded with faults.

The first task is input parameter modeling. The models are represented in two files for each subject: the *Abstract Problem Definition (APD)* and the *Test Case Mapping (TCM)* file. The APD file contains an abstract description of the test subject expressed in terms of the number of parameters, the number of values for each parameter, and the base choice value for each parameter. The TCM contains the mapping between the abstract representation of the test subject and actual values of physical parameters of the test subject. The same pair of APD and TCM files for each test subject were used for all combination strategies. The formats of these files are given in detail in the technical report (Grindal et al. 2003).

The second task is generation of abstract test cases. Based on the contents of the APD file, each combination strategy generates a separate abstract test suite called Input Specification (IS). It contains a list of test cases represented as tuples of parameter values. The test suites were generated automatically, except for OA, whose tests were created manually because it was deemed easier than implementing the OA algorithm.

The third task is translating abstract test cases to real inputs. The abstract test cases from the IS files were automatically converted into executable test cases by using the contents of the TCM file. For every parameter, the appropriate value in the TCM file was located by using the parameter name and value as index. The values identified by the different parameter values of a test case were appended to form the test case inputs. The actual test cases were stored in the Test Suite (TS) file.

The final task is to execute the test cases. A Perl script test case executor took the TS file as input and executed each test case in the file. The test case executor also created a log file in which the names of the different test cases are logged together with any response from the test subject. This log was compared with a reference log created by executing the test suite on a correct version of the program using the Unix "diff."

4.6 Threats to Validity

Lack of independence is a common threat to validity when conducting an experiment. Our approach was to introduce as much independence as possible throughout the experiment. We used an externally developed suite of test subjects. The additional faults were created by a person (third author) not previously involved in the experiment. To further avoid bias, an algorithm was followed for seeding faults as mutation-like modifications. Also, the five input parameters models were created without any knowledge of either the existing faults or the implementations. In fact all the test cases were generated prior to studying the faults and the implementations. These steps avoided all anticipated bias from lack of independence.

An obstacle in any software experiment is how representative the subjects are. In this experiment this issue applies to both programs and faults. A necessary condition in deciding if a sample is representative is knowledge about the complete population. Unfortunately, we do not understand the populations of either programs or faults. This is a general problem in software experimentation, not only for this study. This obviously limits the conclusions that can be drawn, therefore limiting external validity. Reasoning about the ability to detect different types of faults instead of just how many can help with this problem. Also, replicating experiments with different subjects is necessary.

Another aspect of representativity is whether a tester is likely to use the test strategies on this type of test object. As described in section 3, combination strategies are used to identify test cases by combining values of the test subject input parameters. This property makes combination strategies suitable for test problems that have discrete value inputs. However, the tester can always sample a continuous parameter, which is a fundamental aspect of equivalence partitioning, boundary value analysis, and many other test methods. Thus, we draw the conclusion that combination strategies are applicable to any test problem that can be expressed as a set of parameters with values that should be combined to form complete inputs, which is the case for all test subjects in this study.

The decision to avoid conflicts between values of different parameters in the input parameter model could also raise a threat to validity. This decision made it impossible to detect three faults. However, all test techniques used the same input parameter models, so this decision affected all techniques equally. That is, **none** of the test techniques could detect these three faults.

The process of making an input parameter model has much in common with equivalence partitioning in the sense that one value may be selected to represent a whole group of values. The underlying assumption is that all values in an equivalence class will detect a fault equally well. The accuracy of this assumption depends both on the faults we actually have and the experience of the tester. Different testers are likely to derive different classes. Hence, it is desirable to define the equivalence classes in such way that they are representative both with respect to faults and testers. We consider this particular form of representativity very hard to validate. Again, this affected all test techniques equally, so we believe that this only poses a minor threat to the validity of this experiment.

A more in-depth discussion of the validity issues in general may be found in our workshop paper (Lindström, Grindal & Offutt 2004).

5 Results

This section presents results from the experiment. First the number of test cases is presented, then the number and types of faults found, the decision coverage, and finally the failure sizes.

An initial observation from this experiment was that BC and the pair-wise combination strategies (OA and AETG) target different types of faults. Thus, a logical course of action is to investigate the results of combining BC with either OA or AETG. Results for (BC+OA) and (BC+AETG) are included in the data. These results have been derived from the individual results by taking the unions of the test suites of the included combination strategies, eliminating duplicates.

5.1 Number of Test Cases

Table 8 shows the number of test cases generated by the five basic strategies and the two combined strategies. In section 3 formulas were given for EC, BC, and AC to calculate the size of a test suite based on the contents of the input parameter model. The empirical values given in table 8 agree with the theoretical values from these formulas.

The increase in the number of test cases as the coverage criteria get more demanding is expected, and this increase occurs on all five test subjects. Also, OA and AETG satisfy the same coverage criterion and have about the same number of tests.

The test subjects are ordered by size (number of statements), and this order is preserved by all strategies except AC. This is because of differences in the number of parameters and parameter values in the input parameter models, as shown in Table 9. For EC the number of values of the largest parameter will dominate. For BC it is the sum of the parameter values minus the number of

	Combination Strategy						
Test Subject	**EC**	**BC**	**OA**	**AETG**	**BC+OA**	**BC+AETG**	**AC**
count	3	10	14	12	23	21	216
tokens	4	14	22	16	35	29	1728
series	7	15	35	35	48	48	175
nametbl	7	23	49	54	71	76	1715
ntree	9	26	71	64	93	89	2646
total	30	88	191	181	270	263	6480

Table 8: Number of test cases generated by the combination strategies.

Test Subject	**# Parameters**	**# Values for each Parameter**	**Total Values**
count	6	2, 3, 2, 2, 3, 3	15
tokens	8	2, 2, 3, 4, 3, 3, 2, 2	21
series	3	5, 5, 7	17
nametbl	4	7, 7, 7, 5	26
ntree	4	6, 9, 7, 7	29

Table 9: Sizes of test subjects.

parameters that gives the approximate number of test cases. For both OA and AETG the product of the number of values of the two largest parameters gives an approximate size of the test suite. Even if duplicates are removed for the two combined strategies, the order is still preserved when taking the unions of the included test suites. Finally, for AC, the number of test cases is the product of the number of values of each parameter. A test subject with many parameters with few values will require more test cases than a test subject with few parameters with many values even if the total number of values are the same. This can be seen in the case of **tokens**.

The large numbers of tests for AC effectively demonstrates why it is usually not considered practical.

5.2 Faults Found

Table 10 shows the number of faults found in each of the test subjects by the different combination strategies. The first two columns list the number of "known" faults and the number that are "detectable" based on the input parameter models.

There are three reasons why eight faults were never revealed. Three faults depend on parameter values that were removed to keep the input parameter models conflict-free. Our method of automation, feeding all commands from files and writing all output to files, made it impossible to feed commands from stdin and to discriminate between output to stdout and stderr. This made it impossible to detect two more faults. Finally, three faults were missed due to the specific selection of parameters and values in the input parameter models. These three faults were all implementa-

Test Subject	Faults		Combination Strategy					
	known	detectable	**EC**	**BC**	**OA**	**AETG**	**BC+OA**	**BC+AETG**
count	15	12	11	12	12	12	12	12
tokens	15	11	11	11	11	11	11	11
series	20	19	14	18	19	19	19	19
nametbl	49	49	46	49	49	49	49	49
ntree	29	29	25	29	26	26	29	29
total	128	120	107	119	117	117	120	120
% of detectable			89	99	98	98	100	100

Table 10: Number of faults revealed by the combination strategies.

Test Subject	*t*-factor				
	0	**1**	**2**	**3**	**4+**
count	2	7	3	0	0
tokens	3	4	4	0	0
series	0	3	4	12	0
nametbl	2	16	30	1	0
ntree	0	8	9	12	0
total	7	38	50	25	0

Table 11: Number of t-factor faults in each test subject with respect to the input parameter models used in this study.

tion specific. One example of this is a function in the series program to handle rounding errors in the conversion between real and integers. Reals within a 10^{-10} area of an integer are considered equal. To detect the fault in this function a real value within the rounding area would have been needed, but since this was not defined in the specification no such value was used.

Dalal and Mallows (Dalal & Mallows 1998) give a model for software faults in which faults are classified according to how many parameters (factors) need distinct values to cause the fault to result in a failure. A t-factor fault is triggered if the values of t parameters are required to trigger it. This model is used to analyze the faults in this study. Table 11 shows the number of t-factor faults for each test subject with respect to its input parameter model. A 0-factor fault is revealed by any combination of parameter values.

The number of t-factor faults for each t is a direct consequence of the contents of the input parameter model. A changed input parameter model may affect the t-factor classification of faults.

A few of the faults turned out to be 0-factor faults. These are faults that will always produce a failure with the input parameter models used in this experiment. Another interesting observation is that there are no 4+-factor faults and only 25 3-factor faults of the total 120 faults. The same observation, that is, that most faults are 2-factor or less and thus detectable with pair-wise strategies, has been made in a real industry setting. In a study of medical device software, Wallace and Kuhn report that 98% of the faults are 2-factor or less (Wallace & Kuhn 2001). These results may or may not generalize to other kinds of software and actual faults, so additional studies are desirable.

It may be surprising that EC reveals 89% of the detectable faults even though it had relatively few test cases. The 0-factor faults will always be revealed and by definition, EC also guarantees detection of all 1-factor. Taken together, these account for slightly more than one third of the detectable faults.

Many of the 2+-factor faults in this study share the property that many different combinations of values of the involved parameters result in failure detection. Only a small group of the 2+-factor faults require exactly one combination of values of the involved parameters to be revealed. Obviously, the faults revealed by many combinations have a higher chance of being detected than faults revealed by only one combination. Cohen et al. (Cohen et al. 1994) claims that the same situation, that is, that a large number of faults may be revealed by many different parameter combinations, is true for many real-world systems. This is probably why EC is so effective in this study. However, it should be stressed that even if a 2+-factor fault has a high detection rate due to many combinations revealing that fault, EC cannot be guaranteed to detect it.

It is not surprising that BC revealed more faults than EC; after all, it requires more tests. But it may be surprising that BC found a similar number of faults (even slightly more) than OA and AETG, both of which require more tests and more combinations of choices. Looking in detail at the faults is illuminating.

The only fault that BC missed was revealed by EC and both OA and AETG. This fault is located in the **series** program and is a 3-factor fault. Parameter one has five possible values, parameter two also has five possible values, and parameter three has seven possible values, giving a total of 175 possible combinations. Exactly six of these combinations will trigger this fault. Parameter one and two both need to have one specific value and parameter three can have any value except the value 1. None of the values of parameters one and two are base choices, which explains why BC not only missed this fault, but also could not reveal it. That is, revealing this fault required two non-base choice values.

OA and AETG both satisfy pair-wise coverage – every combination of values of two parameters is included in the test suites. In the case of the fault missed by BC, the specific combination of values of parameter one and two is included in one test case in each test suite. However, it is not enough for this combination to reveal the fault, so there is no guarantee that OA or AETG will reveal the fault. But since the third parameter may have several different values and still trigger the fault, the chance of OA and AETG selecting a fault revealing combination is relatively high. This is why EC is so effective, as explained earlier. The chance of EC revealing this fault is small (6 chances out of 175, 3.4%), and the fact that EC revealed it seems largely due to chance. On the other hand, pair-wise strategies have 6 chances out of 7 of revealing the fault (87%) so it is not surprising that OA and AETG found it.

The three faults that only BC revealed are all located in the **ntree** program. Its input parameter model contains four parameters with six, nine, seven, and seven values respectively.

The first fault requires two of the parameters to have one specific value each, one, which happened to be a base choice and a third parameter to have any normal value. Both OA and AETG had one test case each in their test suite with the specific combination of the two parameters,

	Combination Strategy			
Test Subject	**EC**	**BC**	**OA**	**AETG**
count	83	83	83	83
tokens	82	82	86	86
series	90	90	95	95
nametbl	100	100	100	100
ntree	83	88	88	88

Table 12: Percent decision coverage achieved for the correct versions of the test subjects.

but in both cases the third parameter happened to be invalid, so these two strategies did not reveal this fault. This is a good example of fault masking by an invalid value. BC revealed this fault because one of the parameters required exactly the base choice value and the third parameter required any normal value, including the base choice, which is satisfied by the base choice.

The second and third faults that were only revealed by BC are located in the same line of code, and both faults fail for the same combinations of parameter values. For these faults to be triggered, three parameters need to have one specific value each. This means that neither EC, OA, nor AETG can guarantee detection. However, two of the three required parameter values happened to be base choices, so BC was guaranteed to reveal this fault under this input parameter model.

OA and AETG revealed exactly the same faults. Thus we can expect combining either with BC to yield the same results. In this experiment both combinations revealed all detectable faults.

5.3 Decision Coverage

Table 12 shows the decision coverage achieved by each combination strategy on the correct versions of the test subjects. Very little difference was found. OA and AETG covered slightly more decision outcomes than EC did, which can be expected since they had more tests.

These data are also similar to data from previous studies. Cohen, Dalal, Parelius, and Patton (Cohen et al. 1996) obtained over 90% block coverage and Burr and Young (Burr & Young 1998) reached 93% block coverage in experiments using AETG.

It was a little surprising that EC covered as many decisions as it did. As was shown in table 2 the test subjects were all relatively small, in particular with respect to the number of decision points, which means that there is a fair chance of reaching relatively high code coverage even with few test cases.

The high decision coverage achieved for all test subjects by the EC test suites seems to indicate that for this experiment:

1. There is a close correspondence between the specifications and the implementations of the test subjects.
2. The selected parameters and parameter values used for testing the test subjects are good representatives of the total input space.

Strategy	Faults										
	count					tokens		series			
	1	5	6	7	8	1	4	1	2	3	4
EC	33	*67*	100	67	**33**	25	25	14	*0*	**14**	*14*
BC	*10*	**90**	100	**90**	*10*	*7*	7	*7*	7	*0*	**60**
OA	**43**	79	100	*57*	29	23	**27**	**17**	**14**	3	26
AETG	33	75	100	67	25	**31**	*6*	**17**	11	6	20

Table 13: Percent of test cases that failed for each fault for the different combination strategies applied to the test subjects **count**, **tokens**, and **series**. Bold indicates high and italic indicates low for each fault.

Strategy	Faults														
	nametbl								ntree						
	1	2	3	4	5	6	7	8	1	2	4	5	6	7	8
EC	14	71	29	29	71	**14**	*14*	57	*0*	33	22	**56**	**44**	44	67
BC	**48**	**91**	**74**	*26*	**78**	*4*	**74**	*30*	**4**	**77**	**58**	*27*	*19*	**77**	**92**
OA	4	71	27	**31**	59	**14**	*14*	**67**	*0*	32	21	38	31	42	75
AETG	*2*	*69*	*26*	30	*57*	13	15	43	*0*	*23*	*12*	31	27	*28*	*58*

Table 14: Percent of test cases that failed for a each fault for the different combination strategies applied to the test subjects **nametbl** and **ntree**. Bold indicates high and italic indicates low for each fault.

3. The actual test cases generated by EC are well scattered over the implementations.

5.4 Failure size

BC found more faults than OA and AETG, despite having fewer test cases. Failure size is studied to further explore the reasons behind this.

As was presented in section 4.1, the failure size of a fault is defined to be the percentage of test cases in a test suite that fails. Failure size shows a fundamental difference between BC and the other combination strategies.

Tables 13 and 14 contain the failure sizes of the 26 detectable faults in the original benchmark suite (not including the extra added faults, due to space constraints.) The numbers indicate the percent of test cases that triggered the fault. A 0 indicates that no test case triggered that fault, and a 100 indicates that all tests triggered the fault.

An initial observation of this data shows that BC is unusual. Specifically, BC has either the highest or the lowest failure size for 24 of the 26 faults. For many faults, for instance fault 1 in **count**, fault 4 in **series**, and fault 7 in **ntree**, the failure sizes of BC differ from the others by a great deal. To evaluate how different the failure sizes for BC are from the failure sizes for the other criteria, we computed the means of the failure sizes for each fault. Then, for each combination strategy we computed the variance of the distances between the actual failure sizes and the means for each fault. These results are displayed in the bar chart in figure 6. The resulting variances for each combination strategy were then compared using an F-test. With a hypothesis that the variance of distances between the actual failure sizes and the means for each fault for BC is different from the corresponding variances of the other three combination strategies, the results are that the hypothesis cannot be rejected with a significance level of 98%.

BC behaves differently because of the way it selects values. The other combination strategies choose values in an approximately uniform distribution, that is, all the values appear about the same number of times in the test suite. In a BC test suite, however, the base choice values appear more often because every test case is derived from the base test case.

Faults that are triggered by base choice values will result in BC having a higher failure size than the other combination strategies because the base choice values appear so frequently. The opposite is also true; when faults are triggered by non-base choice values, BC will have lower failure size than the other combination strategies.

6 Discussion and Conclusions

Some faults were not found by any combination strategy, yet all the strategies found most of the faults. At the same time, 100% decision coverage was only reached in one program. Thus we conclude that combination strategies should be combined with other methods such as code coverage-based methods. The following subsections discuss more detailed results.

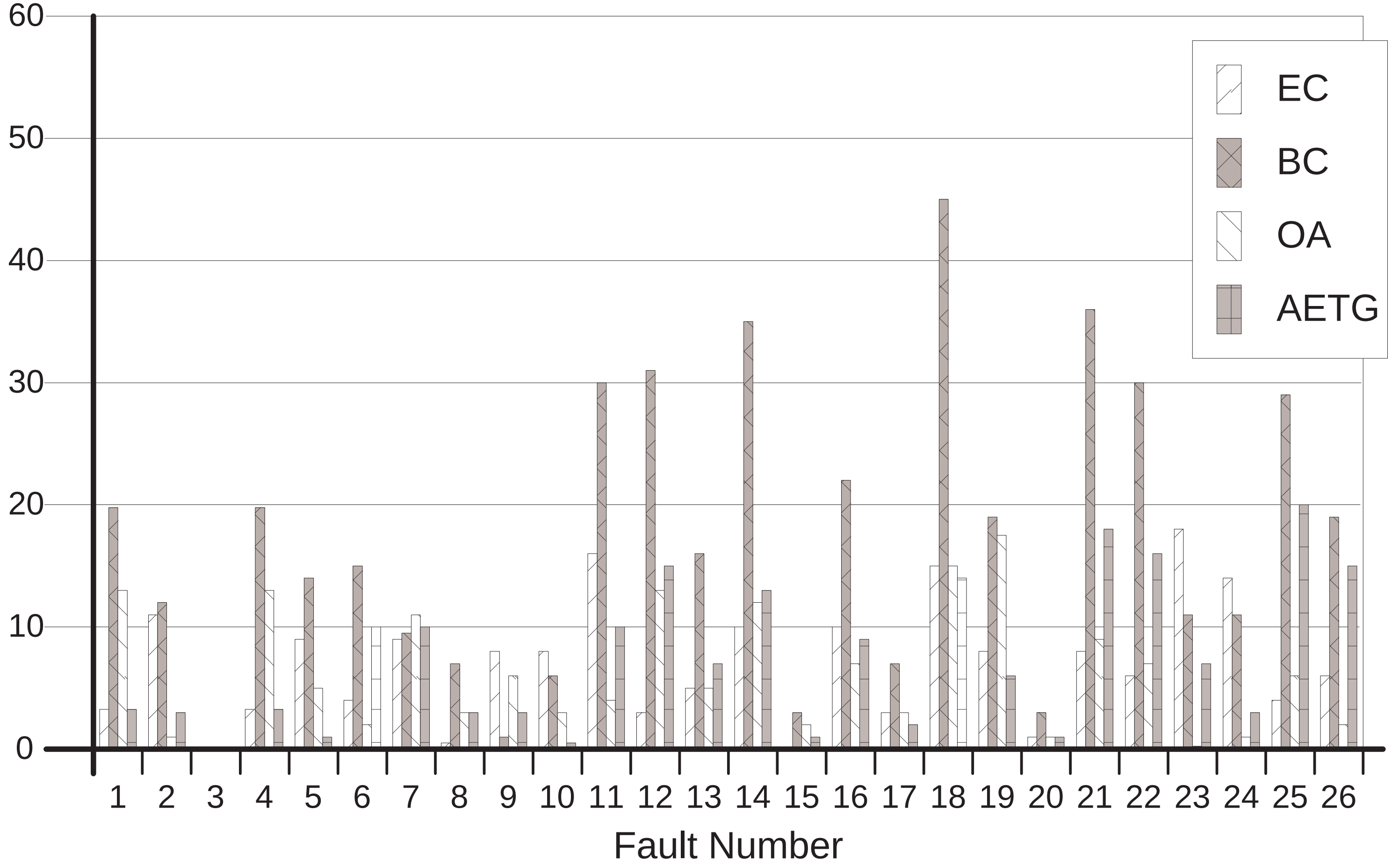

Figure 6: **Means of differences of the failure sizes**

6.1 Fault Finding Abilities of Combination Strategies

This subsection tries to make some general observations about the abilities of the combination strategies to find faults. These observations are based partly on the data, partly on experience applying the strategies, and partly on an analysis of the theory behind the combination strategies. The data showed that EC revealed many faults by coincidence. This leads to the conclusion that the result of applying EC is too unpredictable to be really useful for the tester.

In this experiment the combination strategies are used solely in a black-box manner. That is, they only use information from the specifications when designing the test cases. It is worth stressing that there can be no guarantees that all faults will be found by this approach. This was also the case in this experiment, where three faults were missed because they were implementation related, as described in section 5.2.

BC is different from the other strategies because semantic (or domain) knowledge is used to choose base values. Choosing base values is very simple, but the choice of base values directly affects all of the tests. Ammann and Offutt (Ammann & Offutt 1994) recommend choosing the most commonly used value (based on expected user behavior). Analysis and the data indicate that this means BC tests will be more likely to find faults in parts of the program that are most often used. As a result of BC satisfying single error coverage, no test case will contain more than one invalid parameter value, making it less likely that faults will be masked. This conclusion is based on analysis of the faults that only BC revealed.

A weakness of BC showed up in test problems where some parameters contain two or more values that will be used about the same number of times by the user, that is, there is more than one base choice candidate. In a BC test suite, each non-base choice value will only occur once, which discriminates the other commonly used values. Thus, BC is probably most effective when each parameter contains one obvious base choice value.

Although OA and AETG generate test suites with different contents, they both satisfy pair-wise coverage, and thus have similar performance. Also, the sizes of test suites generated by OA and AETG are similar. Hence, the practicing tester should choose based on other factors, for instance ease-of-use.

It seems easier to automate the AETG strategy than the OA strategy (this is supported by the fact that there is a tool for AETG). It is also straightforward to extend AETG to generate t-wise tests, where t is an arbitrary number. Automating OA is quite difficult.

AETG can also start with an existing test suite and extend it to satisfy pair-wise (or t-wise) coverage. For example, BC tests could be created, and then extended with AETG to satisfy pair-wise coverage.

The pair-wise coverage property of OA and AETG gives best results for test subjects where there are many valid values of each parameter, which is when BC is least effective. For test problems with many invalid values for each parameter, when BC is most effective, the pair-wise strategies may mask faults, that is, when tests cover several parameter value pairs with multiple invalid values, the effects of some parameters may be masked by others. This was also observed by Cohen et al. (Cohen et al. 1997). In test subjects where parameters contain both several valid and several invalid values it seems to be the case that a combination of BC and a pair-wise strategy is required to yield the best effectiveness.

To summarize, our assessment is that BC and AETG should be combined to get the best effects. The BC tests will provide a focus on user behavior and reduce the risk of fault masking, and then extended with pair-wise tests for faults that appear in parts of the program that are not used as much.

6.2 Input Parameter Modeling

A final observation relates to modeling the input parameters. Test engineers often need to choose between representing some aspect of the input space as either a separate parameter or by adding more values to another parameter. The choice should depend on which combination strategy is used.

Given a fixed number of parameters values, for EC, BC, OA, and AETG, it is better to have many parameters with few values, but for AC it is better to have few parameters with many values. It should be noted that this analysis is purely based on cost; no data exists on the relative fault finding abilities.

Piwowarski, Ohba, and Caruso (Piwowarski et al. 1993) showed the applicability of refining the input parameter model by monitoring the code coverage achieved by the generated test suite. This suggests that code coverage may be used both to validate the effectiveness of the input parameter model and to identify parts of the code that should be tested with other means.

6.3 Recommendations

The following recommendations summarize the findings.

- Combination strategies are useful test methods but need to be complemented by code coverage.
- When time is a scarce resource, use BC. Its advantages are its low cost and its user orientation.
- When there is enough time, combine BC with AETG. BC reduces the likelihood of masking of faults, and AETG guarantees pair-wise coverage.
- For the less demanding strategies, when identifying parameters and equivalence classes the test suite will be smaller if more parameters with few values are used than if few parameters with many values are used.

7 Related Work

An interesting example of how different testing methods can be compared is described by Reid (Reid 1997). Software from a real avionics system was used. All existing trouble reports from the first six months of operation were collected. These faults were investigated and any fault that could have been found during component testing were identified and used in the study. Each fault was analyzed to determine the complete set of input values that would trigger that fault, call the *fault-finding set*. Independently of the fault analysis, boundary value analysis and equivalence partitioning were used to partition the input space into test case sets. Under the assumption that each test case has the same probability of being selected, the effectiveness of each testing method was calculated as the probability that all faults will be revealed. Reid found that the mean probability for fault detection for boundary value analysis is 0.73 and for equivalence partitioning it is 0.33. An important contribution by this research is the method of comparison, which can be used to compare any type of testing methods.

Several studies have compared the effectiveness of different combination strategies. By far the most popular property used to compare combination strategies is number of test cases generated for a specific test subject. This is easy to compute and particularly interesting for the non-deterministic and greedy combination strategies, since the size of the test suite cannot be determined algebraically. Several papers have compared a subset of the combination strategies that satisfy 2-wise and 3-wise coverage (Lei & Tai 1998, Williams 2000, Shiba, Tsuchiya & Kikuno 2004, Cohen, Gibbons, Mugridge & Colburn 2003). The combination strategies perform similarly with respect to the number of test cases generated in all of these comparisons.

Since the number of test cases does not clearly favor one particular combination strategy, some authors have also compared the strategies with respect to time consumption, that is, the execution time for the combination strategy to generate its test cases. Lei and Tai (Lei & Tai 2001) show that the time complexity of their combination strategy called In-Parameter-Order (IPO) is superior to the time complexity of AETG. IPO has a time complexity of $O(v^3N^2log(N))$ and AETG has a time complexity of $O(v^4N^2log(N))$, where N is the number of parameters, each of which has v values.

Further, Williams (Williams 2000) reports on a refinement of OA called Covering Arrays (CA) that outperforms IPO by almost three orders of magnitude for the largest test subjects in their study, in terms of time taken to generate the test suites. Finally, Shiba et al. (Shiba et al. 2004) show some execution times but the executions have been made on different target machines so the results are a bit inconclusive.

8 Future Work

There are still several questions to be answered before concluding that combination strategies can be used in an industrial setting, and other questions about how best to apply them. A first question is about the best method to use for input parameter modeling. This experiment kept the model stable and modified the strategy used to generate tests. It would also be useful to precisely describe different (repeatable) methods for generating the input model and then perform another experiment that varies them. A secondary question, of course, would be whether the two variables interact, that is, whether changing the input parameter modeling method would affect which combination strategies work best.

The experiment in this paper also did not directly handle conflicts among parameter values. The best way to handle conflicts needs to be determined, and how they interact with strategies and input modeling methods also needs to be investigated. AETG has a conflict handling mechanism already built into the algorithm (Cohen et al. 1997). Other conflict handling mechanisms are independent of the combination strategies (Grindal et al. 2005), and a follow-up study of the performance of these is currently in process.

This experiment was performed in a laboratory setting and used fairly small programs. This limits the external validity of the results. It would be helpful to try some of these techniques in an industrial setting, both to assess their effectiveness and to assess their true cost.

An obvious result from this study is the conclusion that the contents of the input parameter model affects the test results in a major way. For instance, in section 5.2 it was shown that the number of t-factor faults for each t may differ for different contents of the input parameter model. One plan is to determine how sensitive the test results are to the actual contents of the input parameter model.

In the future, we hope to further develop the tools used in this experiment to be more robust, include more automation, and to be more general. An ultimate goal, of course, is to completely automate the creation of tests. It seems likely that the test engineer would always need to develop the input parameter model and the expected results for each selected test case, but it should be possible to automate the rest of the test process.

We also hope to formalize the description of different types of faults. It would also be interesting to examine variants of combination strategies, for example, using the complete BC test suite as input to the AETG algorithm.

It might also be useful to empirically compare combination testing strategies with other test methods, for instance by the method described by Reid (Reid 1997).

Related to the fault-finding sets studied by Reid is the number of combinations that will reveal a certain fault. It was shown in section 5.2 that two factors influence whether combination strategies reveal a fault. One is the number of parameters involved in triggering the failure (t-factor), and the other is how many combination of values of those parameters that will trigger that fault. As was also observed by Cohen et al. (Cohen et al. 1994), it would be interesting to study faults in real-world applications to determine the properties with respect to fault detection.

9 Acknowledgments

First and foremost we owe our gratitude to Dr Christopher Lott for generously supplying all the material concerning the test subjects and being very helpful in answering our questions. Next, we are indebted to the anonymous referees who offered many helpful suggestions.

Then, we are also indebted to many of our colleagues in the computer science department at the University of Skövde. Louise Ericsson gave us helpful hints when we were lost trying to debug the Prolog program for generating orthogonal arrays. Marcus Brohede and Robert Nilsson supplied valuable tips during the set up of the compiler environment. Sanny Gustavsson, Jonas Mellin, and Gunnar Mathiason gave us general support and good ideas.

Finally, Thomas Lindvall and Åsa Grindal gave helpful advice during the analysis of the results and the writing process respectively.

References

Ammann, P. E. & Offutt, A. J. (1994). Using formal methods to derive test frames in category-partition testing, *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS'94), Gaithersburg MD*, IEEE Computer Society Press, pp. 69–80.

Anderson, T., Avizienis, A., Carter, W., Costes, A., Christian, F., Koga, Y., Kopetz, H., Lala, J., Laprie, J., Meyer, J., Randell, B. & Robinson, A. (1994). Dependability: Basic Concepts and Terminology, *Technical report*, IFIP. WG 10.4.

Archibald, R. (1992). *Managing High-Technology Programs and Projects*, John Wiley and sons, Inc.

Bache, R. (1997). The effect of fault size on testing, *The Journal of Software Testing, Verification, and Reliability* **7**: 139–152.

Basili, V. R. & Rombach, H. D. (1988). The TAME project: Towards improvement-oriented software environments, *IEEE Transactions on Software Engineering* **SE-14**(6): 758–773.

Basili, V. & Selby, R. (1987). Comparing the Effectiveness of Software Testing Strategies, *IEEE Transactions on Software Engineering* **SE-13**(12): 1278–1296.

Basili, V., Shull, F. & Lanubile, F. (1999). Using experiments to build a body of knowledge, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference (PSI 99), Akademgorodok, Novosibirsk, Russia, July 6-9, 1999, Proceedings*, pp. 265–282.

Briand, L. & Pfahl, D. (1999). Using simulation for assessing the real impact of test coverage on defect coverage, *Proceedings of the International Conference on Software Maintenance (ICSM99), 30th of Aug - 3rd Sept, 1999, Oxford, The UK*, pp. 475–482.

Brownlie, R., Prowse, J. & Phadke, M. (1992). Robust Testing of AT&T PMX/StarMAIL using OATS, *AT&T Technical Journal* **71**(3): 41–47.

Burr, K. & Young, W. (1998). Combinatorial test techniques: Table-based automation, test generation and code coverage, *Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR'98)*, San Diego, CA, USA, pp. 26–28.

Cohen, D., Dalal, S., Fredman, M. & Patton, G. (1997). The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Transactions on Software Engineering* **23**(7): 437–444.

Cohen, D., Dalal, S., Kajla, A. & Patton, G. (1994). The automatic efficient test generator (AETG) system, *Proceedings of Fifth International Symposium on Software Reliability Engineering (ISSRE'94), Los Alamitos, California, USA, November 6-9, 1994*, IEEE Computer Society, pp. 303–309.

Cohen, D., Dalal, S., Parelius, J. & Patton, G. (1996). The Combinatorial Design Approach to Automatic Test Generation, *IEEE Software* **13**(5): 83–89.

Cohen, M., Gibbons, P., Mugridge, W. & Colburn, C. (2003). Constructing test cases for interaction testing, *Proceedings of the 25th International Conference on Software Engineering, (ICSE'03), Portland, Oregon, USA, May 3-10, 2003*, IEEE Computer Society, pp. 38–48.

Dalal, S. & Mallows, C. (1998). Factor-Covering Designs for Testing Software, *Technometrics* **50**(3): 234–243.

Dunietz, I., Ehrlich, W., Szablak, B., Mallows, C. & Iannino, A. (1997). Applying design of experiments to software testing, *Proceedings of 19th International Conference on Software Engineering (ICSE'97), Boston, MA, USA 1997*, ACM, pp. 205–215.

Frankl, P., Hamlet, R., Littlewood, B. & Stringini, L. (1998). Evaluating Testing Methods by Delivered Reliability, *IEEE Transactions on Software Engineering* **24**: 586–601.

Grindal, M., Lindström, B., Offutt, A. J. & Andler, S. F. (2003). An Evaluation of Combination Strategies for Test Case Selection, Technical Report, *Technical Report HS-IDA-TR-03-001*, Department of Computer Science, University of Skövde.

Grindal, M., Offutt, A. J. & Andler, S. F. (2005). Combination testing strategies: A survey, *Software Testing, Verification, and Reliability* **15**(3): 167–199.

Harman, M., Hierons, R., Holocombe, M., Jones, B., Reid, S., Roper, M. & Woodward, M. (1999). Towards a maturity model for empirical studies of software testing, *Proceedings of the 5th Workshop on Empirical Studies of Software Maintenance (WESS'99)*, Keble College, Oxford, UK.

Kamsties, E. & Lott, C. (1995a). An Empirical Evaluation of Three Defect Detection Techniques, *Technical Report ISERN 95-02*, Dept of Computer Science, University of Kaiserslauten.

Kamsties, E. & Lott, C. (1995b). An empirical evaluation of three defect detection techniques, *Proceedings of the 5th European Software Engineering Conference (ESEC95), Sitges, Barcelona, Spain, September 25-28, 1995.*

Kropp, N., Koopman, P. & Siewiorek, D. (1998). Automated robustness testing of off-the-shelf software components, *Proceedings of FTCS'98: Fault Tolerant Computing Symposium, June 23-25, 1998 in Munich, Germany*, IEEE, pp. 230–239.

Lei, Y. & Tai, K. (1998). In-parameter-order: A test generation strategy for pair-wise testing, *Proceedings of the third IEEE High Assurance Systems Engineering Symposium*, IEEE, pp. 254–261.

Lei, Y. & Tai, K. (2001). A Test Generation Strategy for Pairwise Testing, *Technical Report TR-2001-03*, Department of Computer Science, North Carolina State University, Raleigh.

Lindström, B., Grindal, M. & Offutt, A. J. (2004). Using an existing suite of test objects: Experience from a testing experiment, *Workshop on Empirical Research in Software Testing*, ACM SIGFOST Software Engineering Notes, Boston, MA, USA.

Lott, C. (n.d.). A repeatable software experiment, *URL:www.chris-lott.org/work/exp/* .

Lott, C. & Rombach, H. (1996). Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques, *Journal of Empirical Software Engineering* **1**(3): 241–277.

Mandl, R. (1985). Orthogonal Latin Squares: An application of experiment design to compiler testing, *Communications of the ACM* **28**(10): 1054–1058.

Miller, J., Roper, M., Wood, M. & Brooks, A. (1995). Towards a benchmark for the evaluation of software testing techniques, *Information and Software Technology* **37**(1): 5–13.

Musa, J. (1993). Operational profiles in software-reliability engineering, *IEEE Software* **10**: 14–32.

Myers, G. (1979). *The Art of Software Testing*, John Wiley and Sons.

Ntafos, S. C. (1984). On Required Element Testing, *IEEE Transactions on Software Engineering* **SE-10**: 795–803.

Offutt, A. J. & Hayes, J. H. (1996). A semantic model of program faults, *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*, ACM Press, San Diego, CA, pp. 195–200.

Offutt, A. J., Xiong, Y. & Liu, S. (1999). Criteria for generating specification-based tests, *Fifth IEEE International Conference on Engineering of Complex Systems (ICECCS'99),Las Vegas NV*, IEEE, pp. 119–129.

Ostrand, T. & Balcer, M. (1988). The Category-Partition Method for Specifying and Generating Functional Tests, *Communications of the ACM* **31**(6): 676–686.

Piwowarski, P., Ohba, M. & Caruso, J. (1993). Coverge measure experience during function test, *Proceedings of 14th International Conference on Software Engineering (ICSE'93), Los Alamitos, CA, USA 1993*, ACM, pp. 287–301.

Reid, S. (1997). An empirical analysis of equivalence partitioning, boundary value analysis and random testing, *Proceedings of the 4th International Software Metrics Symposium (METRICS'97), Albaquerque, New Mexico, USA, Nov 5-7, 1997*, IEEE, pp. 64–73.

Shiba, T., Tsuchiya, T. & Kikuno, T. (2004). Using artificial life techniques to generate test cases for combinatorial testing, *Proceedings of 28th Annual International Computer Software and Applications Conference (COMPSAC'04) 2004, Hong Kong, China, 28-30 September 2004*, IEEE Computer Society, pp. 72–77.

So, S., Cha, S., Shimeall, T. & Kwon, Y. (2002). An empirical evaluation of six methods to detect faults in software, *Software Testing, Verification and Reliability* **12**(3): 155–171.

Wallace, D. R. & Kuhn, D. (2001). Failure Modes In Medical Device Software: An Analysis Of 15 Years Of Recall Data, *International Journal of Reliability, Quality, and Safety Engineering* **8**(4): 351–371.

Williams, A. (2000). Determination of test configurations for pair-wise interaction coverage, *Proceedings of the 13th International Conference on the Testing of Communicating Systems (TestCom 2000), Ottawa, Canada, August 2000*, pp. 59–74.

Williams, A. & Probert, R. (1996). A practical strategy for testing pair-wise coverage of network interfaces, *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE96), White Plains, New York, USA, Oct 30 - Nov 2, 1996*, pp. 246–254.

Wood, M., Roper, M., Brooks, A. & Miller, J. (1997). Comparing and combining software defect detection techniques: A replicated study, *Proceedings of the Sixth European Software Engineerng Conference (ESEC/FSE 97)*, Springer Verlag, Lecture Notes in Computer Science Nr. 1013, pp. 262–277.

Woodward, M. R. & Al-Khanjari, Z. A. (2000). Testability, fault size and the domain-to-range ratio: An eternal triangle, *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00)*, ACM Press, Portland, Oregon, USA, pp. 168–172.

Yin, H., Lebne-Dengel, Z. & Malaiya, Y. (1997). Automatic Test Generation using Checkpoint Encoding and Antirandom Testing, *Technical Report CS-97-116*, Colorado State University.

Zweben, S. & Heym, W. (1992). Systematic Testing of Data Abstractions Based on Software Specifications, *Journal of Software Testing, Verification, and Reliability* **1**(4): 39–55.