An Evaluation of Combination Strategies for Test Case Selection

Mats Grindal[*], Birgitta Lindström[*], A. Jefferson Offutt[†], and Sten F. Andler[*]

2004-10-15

Technical Report HS-IDA-TR-03-001
Department of Computer Science
University of Skövde

Abstract

In this report we present the 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 object to form test cases. One of the investigated combination strategies, namely the Each Choice strategy, satisfies 1-wise coverage, i.e., each interesting value of each parameter is represented at least once in the test suite. Two of the strategies, the Orthogonal Arrays and Heuristic Pair-Wise strategies both satisfy pair-wise coverage, i.e., every possible pair of interesting values of any two parameters are included in the test suite. The fourth combination strategy, the All Values strategy, generates all possible combinations of the interesting values of the input parameters. The fifth and last combination strategy, the Base Choice combination strategy, satisfies 1-wise coverage but in addition makes use of some semantic information to construct the test cases.

Except for the All Values strategy, which is only used as a reference point with respect to the number of test cases, the combination strategies are evaluated and compared with respect to number of test cases, number of faults found, test suite failure density, and achieved decision coverage in an experiment comprising five programs, similar to Unix commands, seeded with 131 faults.

As expected, the Each Choice strategy finds the smallest number of faults among the evaluated combination strategies. Surprisingly, the Base Choice strategy performs as well, in terms of detecting faults, as the pair-wise combination strategies, despite fewer test cases. Since the programs and faults in our experiment may not be representative of actual testing problems in an industrial

[*]Department of Computer Science, University of Skövde, email: {magr,birgitta,sten}@ida.his.se

[†]Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA 22030, USA, email: ofut@isse.gmu.edu

setting, we cannot draw any general conclusions regarding the number of faults detected by the evaluated combination strategies. However, our analysis shows some properties of the combination strategies that appear significant in spite of the programs and faults not being representative. The two most important results are that the Each Choice strategy is unpredictable in terms of which faults will be detected, i.e., most faults found are found by chance, and that the Base Choice and the pair-wise combination strategies to some extent target different types of faults.

Contents

1 Introduction

Combination strategies is a class of test case selection methods where test cases are identified by combining values of the test object input parameters based on some combinatorial strategy. Using all combinations of all parameter values generally results in an infeasibly large set of test cases. Combination strategies is one means of selecting a smaller but yet effective set of test cases. This is accomplished in two steps. In the first step, a small set of "interesting values are identified for each one of the input parameters of the test object. The term "interesting" may seem insufficiently precise and also a little judgmental. However, several, less complex, test selection methods, such as Equivalence Partitioning [Mye79] and Boundary Value Analysis [Mye79] support the task of making this choice, but in order not to limit the use of combination strategies, in this paper, "interesting values" are whatever values the tester decides to use. In the second step, a subset of all combinations of the interesting values is selected based on some coverage criterion.

Some reports on the effectiveness of combination strategies have been produced [BPP92, CDFP97, KKS98]. These reports indicate the usefulness of combination strategies. However, few results report on relative merits of the different combination strategies. Thus, it is difficult, from the reported results, to determine which combination strategy to use. In this report we present the results of a comparative evaluation of five combination strategies.

Four of the investigated combination strategies are based on pure combinatorial reasoning. The Each Choice strategy satisfies 1-wise coverage, which requires each interesting value of each parameter be represented at least once in the test suite. The Orthogonal Arrays and Heuristic Pair-Wise strategies both satisfy pair-wise coverage. 100% pair-wise coverage require every possible pair of interesting values of any two parameters be included in the test suite. The All Values combination strategy generates all possible combinations of interesting values of the input parameters. The fifth, and last strategy, the Base Choice strategy, satisfies 1-wise coverage but in addition makes use of some semantic information to construct the test cases.

The five combination strategies investigated in this research were identified by a literature search. Except for the All Values strategy, which is only used as a reference point with respect to the number of test cases, the combination strategies are evaluated and compared with respect to the number of test cases, number of faults found, test suite failure density, and achieved decision coverage in an experiment comprising five programs, similar to Unix commands, seeded with 131 faults.

A surprising result is that the Base Choice combination strategy, despite fewer test cases than the Orthogonal Arrays and Heuristic Pair-Wise strategies, is the most effective in detecting

faults with the test objects and faults used in our experiment. Although no general conclusions about the fault detecting effectiveness of the combination strategies can be drawn from this, some interesting properties of the combination strategies are revealed by analyzing the faults detected by the combination strategies. In particular, those faults detected by some, but not all, of the combination strategies give insight into these properties.

For instance, the Base Choice strategy appears to target, to some extent, different types of faults than the other combination strategies. Further, the results of using the Each Choice combination strategy is too unpredictable to make the Each Choice strategy a viable option for the tester. The insight that different combination strategies target different types of fault led us to the conclusion that the most effective testing using combination strategies is accomplished if the Base Choice strategy is used together with the Heuristic Pair-Wise strategy. The Heuristic Pair-Wise and Orthogonal Arrays combination strategies perform very similar in all respects but the Heuristic Pair-Wise strategy is easier to automate than the Orthogonal Arrays strategy.

The remainder of this article presents and explains these results in greater depth. Section 2 gives a more formal background to testing, test case selection methods and how these are related to the combination strategies evaluated in this work. In section 3, each one of the investigated combination strategies are described. To ensure reproducibility of the results reported, section 4 contains the rest of the details of the conducted experiments. Section 5 describes the results of the experiment organized in terms of the used metrics. Section 6 contains our analysis of the achieved results. 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 to the work of others. In section 8 we summarize our contributions and finally in section 9 a number of areas for future work are outlined. Appendix B shows all of the faults used in investigation and appendix A contain all test cases generated by the combination strategies.

2 Background

Testing and other fault detection activities are crucial to the success of a software project [BS87]. A *fault* in the general sense is the adjudged or hypothesized cause of an error [AAC+94]. Further an *error* is the part of the system state which is liable to lead to a subsequent failure [AAC+94]. Finally a *failure* is a deviation of the delivered service from fulfilling the system function. These definitions come from the dependability community and are constructed with fault-tolerant systems in mind, in which erroneous states may be detected and corrected before failures occur. The motivation for using these definitions of fault, error, and failure is that these definitions are general enough to be used for any type of system, not restricted to software-only systems, while precise enough to allow investigations like this one. In the context of this investigation we know where (in the software) the faults reside and we also know which failures are caused by which faults. Further we are not interested in keeping track of erroneous states. Thus, the reader may safely think of a fault as an incorrectly written line of code, which if executed with the appropriate input will result in an observable failure.

This informal restraint of the meaning of the terms faults and failures align with the definitions used in the description of the test objects used in our experiment [LR96]. Further, these definitions align with much of the already performed empirical research of software testing. In particular,

the number of found faults is a common metric for evaluation of test case selection methods e.g. [BP99, Nta84, OXL99, ZH92].

A *test case selection method* is a means of identifying a subset of the possible test cases according to a selection criterion. Common to most test case selection methods is that they attempt to cover some aspect of the test object when selecting test cases. Some test case selection methods are based only on information available in specifications of the test object, by for instance requiring all requirements to be covered by test cases. Others are based on the actual implementation of the test object, by for instance requiring all lines of code to be executed at least once during the execution of the selected test cases. Although the assumption is that by covering a specific aspect of the test object, faults will be detected. An important question for both the practitioner and the researcher is: Given a specific test problem which test case selection methods are favorable to use?

There seems to be a common view among researchers that we need to perform experiments and make the results public in order to advance the common knowledge of software engineering [LR96, HHH+99]. For results to be useful in this respect it is important that the experiments performed are controlled and documented in such away that the results can be reproduced [BSL99]. A major reason is that results of different experiments are comparable only if all differences between the experiments are known.

Thus we have opted for comparing a family of test case selection methods, i.e., combination strategies, by creating and performing a repeatable empirical experiment.

3 Combination Strategies

A literature survey over the area of combination strategies revealed five different combination strategies: Each-Choice (EC), Base-Choice (BC), Orthogonal Arrays (OA), Heuristic Pair-Wise (HPW), and All-Combinations (AC). A prerequisite for all combination strategies is the creation of an input parameter model of the test object. The input parameter model contain the parameters of test object and for each parameter a number of interesting values have been selected. The main focus of this paper is to evaluate and compare different combination strategies. This means that the mechanism used to create the input parameter models of the test objects is of minor importance as long as the same input parameter model is used to evaluate all combination strategies. The specific algorithms of each one of these combination strategies are described in more detail in the forthcoming sections. To illustrate the different strategies we will use an example with three parameters A, B, and C, where A has three interesting values, B has two, and C has two.

Like many test case selection methods, combination strategies are based on coverage. In the case of combination strategies, coverage is determined with respect to the combinations of the parameter values included in the input parameter model.

AC require every combination of values to be covered by test cases. Due to the number of test cases required for AC it has been excluded from the experimental part of our experiment and only used as a reference in terms of the number of test cases for the different test objects in the study.

3.1 Each Choice (EC)

The basic idea behind the *Each Choice (EC)* combination strategy is to include each value of each parameter in the input parameter value in at least one test case [AO94]. This is also the definition of 1-wise coverage. An interesting property of the combination strategies is the number of test cases required to satisfy the associated coverage criterion. Let a test problem 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 satisfying 1-wise coverage will have at least $Max_{i=1}^{N} V_i$ test cases.

In our experiments we have applied EC by letting the first value of each parameter form the first test case, the second value of each parameter form the second test case, and so on. In the case where parameters have different number of values some parameters will be completely covered while other parameters still have unused values. In those cases we have for each completely covered parameter identified one value as the most common. These most common values have then been used in all the remaining test cases until all values of all parameters have been used in at least one test case. As an illustration of this method, consider the example with three parameters A, B, and C, where A has three interesting values, B has two, and C has two. The two first test cases will be [1,1,1] and [2,2,2]. Both parameter B and C are then completely covered so the most common values of both these parameters need to be determined. Suppose they are [1] for B and [2] for C. The final test case is then [3,1,2].

3.2 Base Choice (BC)

The algorithm for the *Base Choice (BC)* combination strategy [AO94] starts by identifying one base test case. The *base test case* may be determined based on any predefined criterion such as simplest, smallest, first etc. A criterion suggested by Ammann and Offutt is the most likely value from an end-user point of view.

From the base test case new test cases are created by varying the interesting values of one parameter at a time keeping the interesting values of the other parameters fixed on the base test case. For example, assume a three parameters A, B, and C, where A has three interesting values, B has two, and C has two. Further assume that the base test case is [1,1,2]. Varying the interesting values of parameter A yield two more test cases [2,1,2] and [3,1,2]. Parameter B contributes with one more test case [1,2,2] and the final test case [1,1,1] is the result from varying parameter C. A test suite satisfying 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.

Base Choice satisfies 1-wise coverage since each value of every parameter is included in some test case. However, some semantic information that is taken into account by the algorithm may also affect the coverage of the base choice combination strategy. 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 object to perform some of its intended functions. *Error* values are values outside the scope of the normal working of the test object. If the base choices of the parameters all are normal values, the test suite will, in addition to satisfying 1-wise coverage, also satisfy *single error coverage*.

To satisfy single error coverage, for each error value of a parameter, there is a test case combining that value with a normal value of each of the other parameters.

1	2	3
3	1	2
2	3	1

Figure 1: A 3×3 Latin Square

1	2	3
3	1	2
2	3	1

1	2	3
2	3	1
3	1	2

1,1	2,2	3,3
3,2	1,3	2,1
2,3	3,1	1,2

Figure 2: Two orthogonal 3×3 Latin Squares and the resulting combined matrix

According to the recommendation by Ammann and Offutt, in our experiments we have selected the base choices of each parameter based on an anticipated "most common" choice from a user perspective. A reasonable assumption is that "most common" values are also "normal". Thus, BC in our evaluation satisfy single error coverage.

3.3 Orthogonal Arrays (OA)

In the *Orthogonal Arrays (OA)* combination strategy all test cases in the whole test suite are created simultaneously. This property makes OA unique in this study. In the other four investigated combination strategies the algorithms add one test case at a time to the test suite. Orthogonal Arrays is a mathematical concept that has been known for quite some time. The application of orthogonal arrays in testing was first introduced by Mandl [Man85] and later more thoroughly described by Williams and Probert [WP96].

The foundation of OA is Latin Squares. A *Latin Square* is an $n \times n$ matrix completely filled with symbols from a set with cardinality n such that the same symbol occurs exactly once in each row and column. Figure 1 contains an example of a 3×3 Latin Square with the symbols $\{1, 2, 3\}$.

Two Latin Squares are *orthogonal* iff the combined matrix, formed by superimposing one Latin Square on another, has no repeated elements. Figure 2 shows an example of two orthogonal 3×3 Latin Squares and the resulting combined matrix.

If indexes are added to the rows and the columns of the matrix each position in the matrix can be described as a tuple $< X, Y, V_i >$, where V_i represents the values of the $< X, Y >$ position. Figure 3 contains the indexed Latin Square from figure 1 and figure 4 contains the resulting set of tuples. The set of all tuples constructed by a Latin Square satisfies pair-wise coverage.

To illustrate how orthogonal arrays are used to create test cases consider the test problem used as an example in the descriptions of the previous combination strategies. The input parameter model of the test problem has three parameters A, B, and C, where A has three values, B has two, and C has two. To create test cases from the orthogonal array tuples a mapping between the co-ordinates of the tuples and the input parameter model must be established. Let each co-ordinate represent one parameter and the different interesting values of the parameter map to the

Co-ordinates	**1**	**2**	**3**
1	1	2	3
2	3	1	2
3	2	3	1

Figure 3: A 3×3 Latin Square augmented with co-ordinates

tuple	**Tuple** $< XYV >$
1	111
2	123
3	132
4	212
5	221
6	233
7	313
8	322
9	331

Figure 4: Tuples from the 3×3 Latin Square satisfying pair-wise coverage

different values of that co-ordinate. In the case of the example, map A onto X, B onto Y, and C onto V. This mapping presents no problem for parameter A, but for parameters B and C there are more co-ordinate values than there are values. To resolve this situation, each test case where a co-ordinate value without corresponding value need to be changed. Consider tuple 7 from 4, the value of co-ordinate V is 3 and only values 1 and 2 are defined in the mapping to parameter C. To create a test case from tuple 7, the undefined value should be replaced by an arbitrary defined value, i.e., 1 or 2 in this case. In some cases it is possible to replace undefined values in such a way that a test case can be removed. As an example of this consider tuple 6, the values for both Y and V are undefined and should thus be changed to valid values. Set Y to 1 and V to 2 and the changed tuple is identical to tuple 4 and thus unnecessary to include in the test suite.

A test suite based on orthogonal arrays satisfies pair-wise coverage, even after undefined values have been replaced and possibly some duplicate tuples have been removed. This means that the approximate number of test cases generated by the orthogonal arrays combination strategy is V_i^2, where $V_i = Max_{j=1}^{N} V_j$ and N is the number of parameters in the input parameter model where parameter P_i has V_i values.

Williams and Probert [WP96] give further details on how test cases are created from orthogonal arrays.

3.4 Heuristic Pair-Wise (HPW)

The Automatic Efficient Test Generator (AETG) system was presented by Cohen, Dalal, Kajla, and Patton [CDKP94]. It contains a *heuristic algorithm (HPW)* for generating a test suite satisfying 100% pair-wise coverage, which was described in more detail in [CDFP97]. It is shown in figure 5.

The number of test cases generated by the algorithm for a specific test problem is, among other things, related to the number of candidates (n in the algorithm in figure 5) generated for each test case. In general, the higher the value of n, the smaller amount of test cases is generated. However, Cohen et al. [CDPP96] report that using values higher than 50 will not give any dramatic decrease in the number of test cases.

To illustrate this algorithm consider the example test problem used previously with three parameters A, B, and C, where A has three values, B has two, and C has two. Suppose test case [1,1,1] has already been selected. The three parameter value pairs [1,1,-], [1,-,1], and [-,1,1] are all covered by this test case. The remaining parameter pairs: [1,2,-], [2,1,-], [2,2,-], [3,1,-], [3,2,-], [1,-,2], [2,-,1], [2,-,2], [3,-,1], [3,-,2], [-,1,2], [-,2,1], and [-,2,2] are thus uncovered (in figure 5 represented by the *UC* data structure).

The first step in the algorithm is to select the parameter and the value included in most pairs in UC, i.e. least covered. $A = 1$ is included in 2 pairs, $A = 2$ in 4, $A = 3$ in 4, $B = 1$ in 3, $B = 2$ in 5, $C = 1$ in 3, and finally $C = 2$ in 5. Either $B = 2$ or $C = 2$ can be selected. This illustrates the first point in which the randomness of the algorithm is present. Suppose $C = 2$ is selected. The second step is to make a random order of the remaining variables. This is the second point where the randomness is present. Suppose the order B, A is selected. The third step is to find values for B and A, in that order, such that most uncovered pairs will be covered, given that C is already

```
Assume test cases TC_1 − TC_{i−1} already selected
Let UC be a set of all pairs of values of any two parameters that are
not yet covered by the test cases TC_1 − TC_{i−1}
A) Select candidates for TC_i by
     1) Selecting the variable and the value included in most
        pairs in UC.
     2) Making a random order of the rest of the
        variables.
     3) For each variable, in the sequence determined
        by step two, select the value included in most
        pairs in UC.
B) Repeat steps 1-3 n times and let TC_i be the test
   case that covering most pairs in UC. Remove those pairs from UC.

Repeat until UC is empty.
```

Figure 5: Heuristic algorithm for achieving pair-wise coverage

set to 2. $B = 1$ and $B = 2$ yield the same result, i.e., covering one new pair in UC, so this is the third point where the randomness of the algorithm plays a role. Suppose $B = 1$ is selected. Given $C = 2$ and $B = 1$, $A = 1$ covers only one new pair since [1,1,-] is already covered from the first test case. Both $A = 2$ and $A = 3$ will cover two pairs each, which again demonstrates the randomness in the third step. Suppose $A = 3$ is selected yielding a final test case candidate [3,1,2].

The algorithm is now repeated $n - 1$ more times exploiting the randomness of the algorithm to create test case candidates in the same manner. In the final step of the algorithm the test case candidates are evaluated and the candidate that covers most new pairs will be promoted to a test case. Suppose that our first test case candidate was the best test case. (In this example covering three pairs in UC is the best we can do). The three pairs [3,1,-], [3,-,2], and [-,1,2] are then removed from the set of uncovered pairs (UC) resulting in the following pairs remaining uncovered after the second test case has been decided: [1,2,-], [2,1,-], [2,2,-], [3,2,-], [1,-,2], [2,-,1], [2,-,2], [3,-,1], [-,2,1], and [-,2,2]. The complete algorithm is then restarted to find test case three and so on until no more uncovered pairs exist.

This heuristic nature of the algorithm makes it impossible to calculate the minimal number of test cases generated by this algorithm.

In our experiments we have used $n = 50$ according Cohen et al. [CDFP97].

3.5 All Combinations (AC)

All-combinations (AC) requires that every combination of values of each one of the n different parameters be used in a test case [AO94]. This is also the definition of n-wise coverage. To illustrate the all combinations strategy, consider the same test problem as previously with three

parameters A, B, and C, where A has three values, B has two, and C has two. This test problem results in 12 test cases: [1,1,1], [2,1,1], [3,1,1], [1,2,1], [2,2,1], [3,2,1], [1,1,2], [2,1,2], [3,1,2], [1,2,2], [2,2,2], and [3,2,2]. In the general case a test suite satisfying n-wise coverage will have will have $\prod_{i=1}^{N} V_i$ test cases, where N is the number of parameters of the input parameter model where parameter P_i has V_i values.

Due to the large number of test cases required for 100% n-wise coverage no empirical tests have been performed with this combination strategy. This strategy is only included in the theoretical comparisons as a reference.

3.6 Combined strategies (BC+OA),(BC+HPW)

One of the early results from our experiments is that BC and the pair-wise combination strategies (OA and HPW) to some extent target different types of faults. A possible test strategy would thus be the combined use of BC and either OA or HPW.

In our results and the following analysis we have included these two possibilities. The results for (BC+OA) and (BC+HPW) have been derived from the individual results for the three combination strategies by taking the unions of the test suites of the included combination strategies. That way, no duplicate test cases are counted.

4 Experimental Setting

Empirical evaluations of test case selection methods have been conducted by researchers for more than 20 years. Hetzel [Het76], Howden [How78], and Myers [Mye78] are all examples of early empirical work, while So et al. [SCSK02] is a recent empirical study. Harman et al. [HHH+99] claim that in order for results of empirical investigations to be useful the experiments performed need to be conducted in a standardized way and both the experiments and the results need to be described in such detail that the experiments can be independently repeated. This view is also supported by Miller et al. [MRWB95]. The two purposes of repeatable experiments are described by Wood, Roper, Brooks and Miller [WRBM97] as making it possible to validate other people's work and extending the knowledge by making controlled changes to already performed experiments. Our hope with this research is to make a small contribution to the set of repeatable experiments.

To achieve structure for our experiments we have opted for the GQM (Goal, Questions, Metrics) method [BR88] since it enables classification and description of empirical studies in an unambiguous way while being easy to use. Further, the GQM paradigm supports the definition of goals and their refinement into concrete metrics. The use of the GQM method is supported by Lott and Rombach [LR96] from which the test objects used in our experiments have been borrowed.

Applying the goal template of the GQM paradigm on this study yield the following description:

Our goal is to analyze **five combination strategies** for the purpose of **comparison** with respect to their **effectiveness and efficiency** from the point of view of the **practicing tester** in the context of **a faulty program benchmark suite**.

Based on this overview of our experiments, the following sections describe the different properties of our experiments in more detail and motivate some of the specific implementation decisions

that were taken during the course of designing and setting up the experiments.

4.1 Evaluation Metrics

As outlined in the GQM template definition of these experiments the purpose of the experiments is to compare effectiveness and efficiency of the different combination strategies from the point of view of the practicing tester.

Frankl et al. state that the goal of testing is either to measure or increase the reliability [FHLS98]. Testing to measure the reliability is usually based on statistical methods and operational profiles, while testing to increase the reliability relies on test case selection methods that are assumed to generate test cases which are good at revealing failures. The test case selection methods investigated in this research are definitely of the second category, that is, targeted towards detecting failures. At first glance, the number of detected failures would be a good metric for evaluation of effectiveness. However, a test suite that detects X failures that are the results of the same fault can be argued to be less effective in increasing the reliability than a test suite that detects X failures that are the results of X different faults. Thus, assessing the actual number of faults detected by a test suite is a better effectiveness metric.

The efficiency of a test case selection method from a practicing tester's point of view is related to the resource consumption of applying that test case selection method. The two basic types of resources available to the tester are time and money [Arc92]. Models of both types of resource consumption are difficult to create and validate. Variation over time of the cost of computers and personnel is one example of the problems when using money related efficiency metrics. Time related metrics need, among other things, to consider the variation in human ability, the increasing performance of computers and the level of automation of the tasks. Nevertheless, some researchers have used "normalized utilized person-time" as a measure of the efficiency of a test case selection method [SCSK02].

In our experiments we have assumed that each evaluated combination strategy has approximately the same set-up time, and that the cost of executing a single test case is equal across the combination strategies. These assumptions make it possible to disregard the varying factors of the time consumption and approximate efficiency with the number of test cases in each test suite.

In summary our main metrics for measuring effectiveness and efficiency are number of faults found and number of test cases generated by the test strategies.

To aid the analysis of our results we also collected some secondary metrics, i.e., code coverage and test suite failure density.

Beizer, among others, point out the need for both black-box and white-box testing techniques [Bei90]. The evaluated combination strategies in these experiments represent black-box techniques, that is, test case selection methods purely based on specifications. To understand the need for white-box techniques as a complement to combination strategies, we also included code coverage as a metric to be evaluated in our experiments. We chose between statement and decision coverage [ZHM97]. Both metrics are simple to implement and understand. The main difference between the two is that an if-statement without else-branch may be covered by one single test case with statement coverage but requires two test cases to be covered with decision coverage. Since several of the test objects contained if-statements without an else-branch we decided to use

decision coverage.

Test suite failure density is the percentage of test cases in a test suite that fail for a specific fault. The use of test case failure density is to help demonstrate differences in the behavior of the combination strategies.

Finally, we also to some extent consider the difficulty to automate the combination strategies since a test method without tool support has little relevance for the industry.

In our experimental implementation we have used a number of variants of each program, where each variant contains exactly one fault. The main reason is to isolate the effects of each fault and make the analysis from failure to fault simpler. When presenting our results, we have chosen to present the number of faults found rather than the percentage of faults found. The reason is that the percentages can be derived from the presented results but not the other way around. In the experiments we have measured achieved decision coverage on the fault-free versions of each program. Moreover, the decision coverage achieved by executing the test suites on the faulty programs were also monitored and compared with the achieved decision coverage for the fault-free versions.

4.2 Test Objects

A defined goal of this research was to use a benchmark suite of programs containing a set of faults for the evaluation of the different combination strategies. The main argument of the choice of a defined benchmark suite is provided by Miller et al. [MRWB95]. Their argument is that a unification of experiment results is only possible if experiments are based on a commonly accessible repository of correct and faulty benchmark programs.

Following the advice by Miller et al. we had two options. Either we could use an existing suite of programs or we could manufacture our own benchmark programs and make them public. A literature study over a large number of experiments that has been performed and reported was conducted. The survey aimed to identify candidate suites of programs that could be used in our experiments. However, this survey only revealed one such candidate suite of programs that was accessible to other researchers. Kamsties, Lott and Rombach developed a suite of six programs for a repeatable experiment comparing different defect detection mechanisms [LR96]. The development of this experiment package was inspired by an experiment performed by Basili and Selby [BS87]. The benchmark program suite was used in two independent experiments staged by Kamsties and Lott [KL95a, KL95b]. Later on the experiment was repeated by Wood et al. [WRBM97] using the same program benchmark suite.

Although this program benchmark suite was originally intended for comparing black-box and white-box test techniques with code reading techniques the well documented programs complete with specifications and descriptions of existing faults more than enough fulfilled our requirements on the program benchmark suite. Using an existing benchmark suite, rather than writing an own, would also help preserve the independence in the experiment.

The program benchmark suite[1] contain six programs similar to Unix commands.

[1]The complete documentation of the Repeatable Software Experiment may be retrieved from URL: www.chris-lott.org/work/exp/

count is an implementation of 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 separated by one or more white-spaces (space, tab, or line break).

tokens reads its input from the standard input and counts all alphanumeric tokens and prints their counts in increasing lexicographic order. A number of flags can be given to the command to control which tokens that should be counted.

The command **series** requires a start and an end argument and prints the real numbers from start to end in a step regulated by an optional step size argument.

nametbl reads commands from a file and processes them in order to test a few functions. Considered together, the functions implement a symbol table for a certain computer language. The symbol table stores for each symbol its name, the object type of the symbol, and the resource type of the symbol. The commands enable the user to insert a new symbol, enter the object type, enter the resource type, search for a symbol, and print the entire symbol table.

ntree reads commands from a file and processes them in order to test a few functions. Considered together, 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 content. The commands enable the user to add a root, add a child, search for a node, check if two nodes are siblings, and print the tree.

The sixth program, **cmdline**, was left out from the experiment since it did not contain enough details in the specification to fit our experimental set-up.

4.3 Test Object Parameters and Values

To create input parameter models for the test problems we used equivalence partitioning [Mye79] to identify parameters and interesting values. The specifications of the five test objects were analyzed and an equivalence class model was created for each program. The identification of parameters was not restricted to actual input parameters, for instance the use of a flag. Abstract parameters, such as the number of arguments or number of same tokens in the tokens test object, were also considered. From each equivalence class we picked a representative value. One value for each parameter was picked as the base choice value of that parameter.

At one point in the process of identifying parameters we discovered conflicts between values of different parameters. A conflict has occured when the value of one parameter can not occur with all the values of another parameter. An example of a conflict is when the value of one parameter requires that all flags should be used while the values of another parameter states that a certain flag should be turned off. Of the evaluated combination strategies in this experiment only HPW contains a complete mechanism to handle such conflicts. BC contains an outline for parameter conflict handling. EC and OA have no way of handling parameter value conflicts built-in.

In a real test problem, in which fault detection is the main goal, parameter value conflicts need to be handled. However, in our experiment our primary aim was to compare the different combination strategies. In order to make this comparison as fair as possible we decided to neglect parameters and values that would cause conflicts, i.e., we have adapted our input parameter model such that all possible combinations of values are conflictfree. The consequence of this decision was that some properties of the test objects may not be included in our input parameter model. Thus,

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	$>$ **1**(**4**)	$>$ **1**(**2**)	tab	1
2	-	$>$ **1**(**2**)	-	-	both	$>$ **1**(**2**)

Table 1: Equivalence classes and values chosen (in parentheses) for the count test problem. Base choices indicated by bold.

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

Table 2: Equivalence classes and values chosen (in parentheses) for the tokens test problem. Base choices indicated by bold.

some faults may not be possible to detect since the required values for the detection of those faults are not used in the testing.

The tables below give a short description of the used parameters and values used for each test object. The base choices of each parameter is indicated in **bold**. Appendix A contain the test cases generated by the combination strategies based on the contents of the tables.

4.4 Faults

The five studied programs in the used benchmark suite contain 33 known faults. The combination strategies exhibit only small differences in fault detection of these 33 faults. To increase the confidence in the results another 118 faults were created. The programs were seeded with mutation like faults, e.g., changing operators in decisions, changing orders in enumerated types, turning post-increment into pre-increment etc. The result is that each function of each program contains some faults.

The only thing known to the person creating these additional faults were the algorithms of the combination strategies. In particular, knowledge of the less complex algorithms, EC and BC, may introduce bias in the fault seeding process since the test case generated by these algorithms are quite straight forward. To minimize the risk of bias we made sure that neither specifications of the programs nor actual parameters, selected values or complete test cases were known to the seeder.

I	start	end	stepsize
0	$< 0(-10)$	$< 0(-5)$	no
1	0	0	$< 0(-1)$
2	$>$ **0(15)**	$>$ **0(5)**	0
3	real (5.5)	real (7.5)	1
4	non-number	non-number	$>$ **0(2)**
5	-	-	real (1.5)
6	-	-	non-number

Table 3: Equivalence classes and values chosen (in parentheses) for the series test problem. Base choices indicated by 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 4: Equivalence classes and values chosen (in parentheses) for the nametbl test problem. Base choices indicated by 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 5: Equivalence classes and values chosen (in parentheses) for the ntree test problem. Base choices indicated by bold.

Twenty of the 151 faults were functionally equivalent to the correct program so these were omitted from the experiment. Thus our experiment used 131 faults.

Prior to execution we tailored the benchmark suite in two ways to be able to extract the information we needed. The first adjustment concerned handling of the faults in the programs. The original programs came in one version including all faults at once. We implemented a number of copies of each program, one correct version and one version for each fault in the program. This was done to avoid dependencies among the faults. A bonus with this approach was that we could also to a large extent automate the comparison of actual and expected outcome by using the outcome produced by the correct program as a reference and using the Unix command "diff" to identify the deviations.

The second adjustment concerned how to measure the achieved code coverage. When analyzing the code of the different programs we discovered that the programs had been supplied with some extra code for the purpose of the original experiment by Lott et al. This functionality was not specifically described nor did it contain any of the original faults. In fact, there were explicit comments in the code instructing the testers not to test those parts. We decided that it would be unfair to include this extra code in the measurement of code coverage, so we decided to only monitor decision coverage of the parts of the code that were part of the implementation of the specification. For these, rather small, programs it was deemed easier to make this selective monitoring by manually inserting code coverage instrumentation instructions rather than employing a tool for this purpose.

Appendix B contain listings of the programs including the faults used in this experiment.

An obstacle in any software experiment is the issue of representativity of the subjects of the experiments. In this experiment this issue applies both to the programs used and the faults contained in those programs. Thus, the generality of the conclusions that can drawn from the results of this experiment are limited.

4.5 Infrastructure for Test Case Generation and Execution

The experiments were conducted in a number of semi-automatic tasks. Both test case generation and test case execution were included in the semi-automatic approach. Figure 6 gives an overview of the experiment tasks and the intermediate representations of the information between each pair of tasks. Omitted from the figure but included in the experiments is preparation of the test objects prior to execution. This task includes instrumentation of the test object for monitoring of code coverage. It also includes ensuring that the different faults of a certain test object are separated into different versions of that test object.

The following sections give a brief overview of each of the different tasks in the process. Figure 7 contains the specification of a simplified version of the Unix command "ls" which will be used to illustrate the tasks in our test experiment process.

Input parameter modeling is the first task in the test case generation and execution process. This is a completely manual step in which the specification of the test object is analyzed in order to determine the parameters and the values of each parameter of the test object. Equivalence Partitioning [Mye79], Boundary Value Analysis [Mye79], and the Category Partition Method [OB88] are all methods that can be used to accomplish this task.

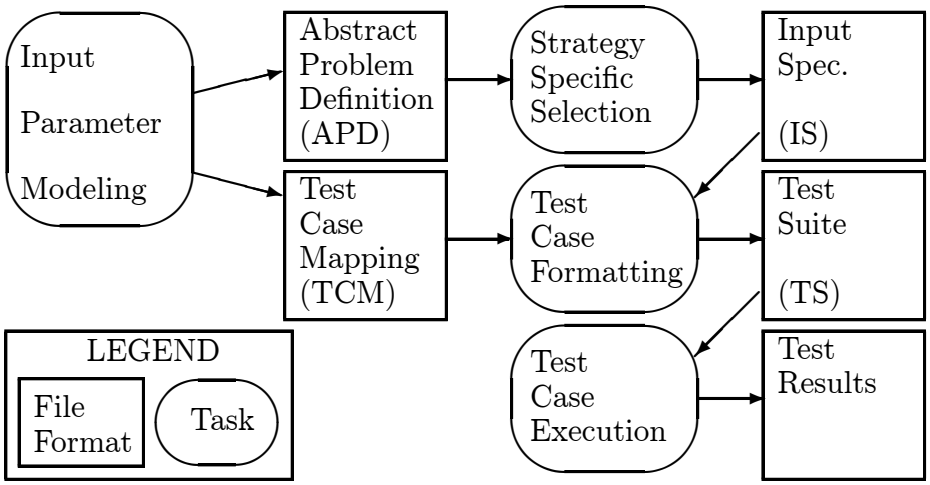

Figure 6: Test Case Generation and Execution

Specification

```
Format:  ls [-a] [-l]
Description:  ls prints the contents of current directory.
              -a all files including hidden files will be printed.
              -l long version of each file name, including file size,
                 permission, owner, and date will be printed.
              Incorrect use of the command will result in an error message.
              For simplicity reasons, this version of ls can only list the
              contents of the current directory.
```

Figure 7: Specification of a simplified version of the Unix command "ls".

```
#dimensions 3
#values 3
FlagA 2 1
FlagL 2 0
Use   3 0
```

Figure 8: Abstract Problem Definition (APD) file of the "ls" example

Equivalence class partitioning applied to the "ls" example in figure 7 is used to illustrate this step. The formal parameters of the command are the two flags. For each one of these flags the equivalence classes are (1) flag used and (2) flag not used respectively. An informal parameter of the command is the use of the parameter (correct/incorrect). Three equivalence classes are identified for this informal parameter: (1) Correct usage - any valid combination of the flags, (2) Incorrect use - unknown flag, and (3) Incorrect use - existing flags repeated.

Other parameters and equivalence class partitions are possible for the "ls" command example, for instance number of arguments to the "ls" command. It is not within the scope of this investigation to identify the optimal input parameter model. Our objective is to identify a reasonable input parameter model to be used as baseline for the comparison. Thus, we limit ourselves to the three parameters and their identified equivalence classes.

The results of this first step are documented in two different files: the *Abstract Problem Definition (APD)* and the *Test Case Mapping (TCM)* file. The APD contains an abstract description of the test problem expressed in terms of number of parameters, number of values for each parameter and identification of the base choice value of each parameter. The TCM contains the mapping between the abstract representation of the test problem and the actual values for each parameter of the test problem. The specific APD and TCM files for a certain test problem are generic and are used for all of the evaluation of all of the different combination strategies. Thus, input parameter value modeling is only performed once for each test problem.

Suppose that the most common use of the "ls" command is "ls -a". Figure 8 shows the resulting APD file for the "ls" example.

The key-words "#dimensions" and "#values" denote the number of parameters of the test problem and the maximum number of values of any parameter. Each identified parameter is described on a separate row with its name, the total number of values for that parameter, and the index (using zero count) of the base choice value for that parameter.

The corresponding TCM file for the "ls" example is shown in 9. The contents and structure of the TCM are test problem specific since the idea is to generate executable test cases for the test object and the test objects may differ. However, the underlying structure of the TCM is a hash function, which uses the name of the parameter and the index of the value as the key.

The second task of the test case generation and execution process is the strategy specific selection of abstract test cases. Input to this step is the APD file which defines the size of the test problem to be executed. Output from this step is the Input Specification (IS). Except of the

```
#command ls
#FlagA - Whether or not to use the -a flag
#0 - No
#FlagA0
#1 - Yes
#FlagA1 -a

#FlagL - Whether or not to use the -l flag
#0 - No
#FlagL0
#1 - Yes
#FlagL1 -l

#Use - How the arguments of the ls command are configured
#0 - Correct use
#Use0
#1 - Unknown flag
#Use1 -d
#2 - repeated flag
#Use2 -a
```

Figure 9: Test Case Mapping (TCM) file of the "ls" example

```
#dimensions 3
#values 3
#dim-names FlagA FlagL Use
TC1 1 0 0
TC2 0 0 0
TC3 1 1 0
TC4 1 0 1
TC5 1 0 2
```

Figure 10: Input Specification (IS) file generated by the base choice combination strategy for the "ls" example

orthogonal arrays combination strategy, the strategy specific selection is completely automated. The algorithms of each combination strategy is implemented in separate modules. For orthogonal arrays the IS file is created manually since the algorithm for orthogonal arrays is difficult to implement.

Figure 10 shows the IS file generated by the base choice combination strategy module for the "ls" example. The key-words "#dimensions" and "#values" have the same meaning as in the APD file i.e., the number of parameters of the test problem and the maximum number of values of any parameter. The key-word "#dim-names" denotes an enumerated list of parameter names. Each row in the remainder of file contain one test case with test case identity followed by the a tuple containing indexes (using zero-count) for each parameter in the same order as they occur in the dim-names list.

The third task of the test case generation and execution process is the test case formatting. In this step the abstract test cases of the IS file are automatically converted into executable test cases via the contents of the TCM file.

For every parameter of a test case the appropriate value in the TCM file is located by using the parameter name and value as index in the TCM file. The values identified by the different parameter values of a test case are appended to form the complete input of that test case. The actual test cases are stored in the Test Suite (TS) file. Figure 11 contains the TS file resulting from the IS and TCM files of the "ls" example. Each test case is documented over three rows in the IS file. The key-word "#name" identifies the test case. The key-word "#input" represent the actual input of the test case. In the example a command invocation. The key-word "#expect" is optional and can be used to represent the expected result of the test case. In the example the "#expect" is not used.

The final step in the test case generation and execution process is the test case execution. The test case executor Perl script takes the TS file as input and executes each test case in the file by firing the command after the reserved word "#input" in a normal Unix shell. The test case executor also creates a log file in which the names of the different test cases are logged together with any response from the test object. Although the file format of the TS file is prepared for automatic

```
#name TC1
#input ls -a
#expect

#name TC2
#input ls
#expect

#name TC3
#input ls -a -l
#expect

#name TC4
#input ls -a -d
#expect

#name TC5
#input ls -a -a
#expect
```

Figure 11: Test Suite (TS) file for the “ls” example

comparison of actual and expected outcome, this functionality has not been implemented in the test case executor. In our experiments we have instead created a reference log by executing the test suite on a correct version of the program. This reference is then compared with the logs from the faulty programs using the Unix "diff" command off-line.

A general problem with our approach to append the values of the different parameters to form the input of a test case is that there may be dependencies between values of different parameters. Consider the "ls" example and the test case [0 0 2]. The values of the first and second parameters indicate that none of the a and l flags should be used while the value of the third parameter indicate that the same flags should be repeated, which of course is a conflict. A number of different schemes to handle such conflicts have been suggested. Ammann and Offutt [AO94] suggest a BC algorithm specific solution. Similarly, Cohen et al. [CDFP97] describe a solution tailored to the HPW algorithm. Cohen et al. [CDFP97] also describe a more general approach in which the test problem is described as two or more conflict free sub-relations instead of one large relation as outlined in this article. The drawback with this approach as reported by Cohen et al. [CDFP97] is that the number of test cases is greatly affected and may vary unintuitively by how the relations are described.

As was described in section 4.3 our approach in this experiment was to create a conflictfree input parameter model. The immediate result of this is that we have sometimes ignored parameters that in a real test problem would be obvious to test. A specific example is that there are no test cases in our experiment with an illegal number of parameters. Our choice to ignore some properties is justified by the fact that the main aim of our experiment is to compare different combination strategies. Even if some properties are ignored we believe that our aim can be met as long as all combination strategies have a reasonable starting point which is the same for all. This view is further strengthened by the fact that all of the combination strategies detect a vast majority of all of the faults with the given set of parameter and parameter values.

5 Results

The results of our experiment are presented in the subsequent subsections. To form a base line for the comparison of the combination strategies we start by investigating the subsumption relations among the coverage criteria satisfied by the investigated combination strategies. The subsequent sections contain the results organized according to the different metrics assessed.

5.1 Subsumption

Subsumption is used to establish partial orders among coverage criteria and can thus be used to compare test coverage criteria. The definition of *subsumption* is: coverage criterion X *subsumes* coverage criterion Y iff 100% X coverage implies 100% Y coverage [RW85] (In Rapp's and Weyuker's early terminology subsumption was called inclusion).

Figure 12 shows the subsumption hierarchy for the coverage criteria of the studied combination strategies. From section 3 we have that EC satisfies 1-wise coverage, OA and HPW satisfy pair-wise (2-wise) coverage, and AC satisfies n-wise coverage. The fifth combination strategy, BC, as implemented in this experiment satisfies both 1-wise coverage and single error coverage. These

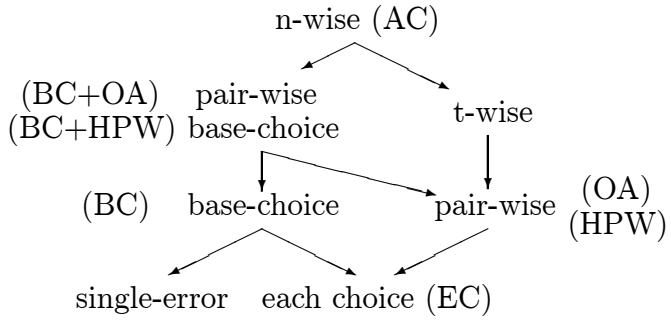

Figure 12: Subsumption hierarchy of the algorithms for the different combination strategies (denoted by the different acronyms) and their respective coverage levels.

two coverage criteria are incomparable, so to include BC in the subsumption hierarchy we have added base-choice coverage as defined by Ammann and Offutt [AO94]. The effect of combining BC with HPW or OA is a test suite satisfying 2-wise coverage and base choice coverage at the same time. To incorporate this into the subsumption hierarchy we also introduce a new coverage criterion called pair-wise base-choice coverage, which is the union of the pair-wise and base-choice coverage criteria.

5.2 Number of Test Cases

Table 6 shows the number of test cases generated by the five basic algorithms and the two combination algorithms investigated. For EC, BC, and AC the obtained number of test cases agree with the theoretical values when using the formulas to calculate the number of test cases.

The results for HPW were only obtained empirically due to the heuristic nature of HPW. The results for the two combined strategies (BC+OA and BC+HPW) were calculated from the combined results of the simple strategies. In this calculation duplicate test cases were removed.

The increase in the number of test cases as the coverage criteria get more demanding is expected. This property is visible for all five test objects. A first observation is the similarity of the two pair-wise combination strategies OA and HPW. This similarity is directly reflected in the similar behavior of the two combined strategies.

Another observation is that the relative order among the five test objects is preserved for all combination strategies except for AC, where the test objects series and tokens behave differently. The explanation for this can be found in the sizes of the test objects. Table 7 shows the number of parameters and the number of values of each parameter for the five test objects. The test objects count and tokens have many parameters with few values in each parameter, while series, nametbl, and ntree have fewer parameters but more values in each parameter. In the less demanding coverage criteria, the number of values of each parameter (in particular the parameter with the largest number of values) is more important than the actual number of parameters for the size

	Combination Strategy						
Test Object	**EC**	**BC**	**OA**	**HPW**	**BC+OA**	**BC+HPW**	**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 6: Number of test cases generated by the combination strategies for the test objects.

Test Object	**Number of Values of each Parameter**
count	$2, 3, 2, 2, 3, 3$
tokens	$2, 2, 3, 4, 3, 3, 2, 2$
series	$5, 5, 7$
nametbl	$7, 7, 7, 5$
ntree	$6, 9, 7, 7$

Table 7: Sizes of test problems.

of the test suite. In the more demanding coverage criteria both the number of parameters and the number of values of each parameter is important since the number of test cases generated is approaching the product of the number of values of each parameter.

A third observation from table 6 is that HPW generates fewer test cases than OA in all cases except one i.e., nametbl. Whenever a test problem fits the orthogonal Latin Squares perfectly the OA method will generate a minimal test suite for pair-wise coverage [WP96]. The results of the HPW algorithm depends on the number of candidates evaluated for each test case. No guarantees can be made due to the heuristic nature of the algorithm. In our experiment the nametbl almost fits two orthogonal 7 x 7 Latin Squares perfectly so the good performance of OA for that test object is not surprising.

The AC strategy was included in the investigation to act as a point of reference for the number of test cases. Thus, no results are reported for AC in the remainder of the results section.

5.3 Faults Found

An important property of a test method is its fault detection ability. One of the key issues in this experiment is to examine this property for the evaluated combination strategies. Table 8 shows the number of faults found in each of the test objects by the different combination strategies. The first column "known" contains the number of faults for each test object. The second column

Test Object	Faults		Combination Strategy					
	known	detectable	**EC**	**BC**	**OA**	**HPW**	**BC+OA**	**BC+HPW**
count	15	12	11	12	12	12	12	12
tokens	16	11	11	11	11	11	11	11
series	19	19	14	18	19	19	19	19
nametbl	49	49	46	49	49	49	49	49
ntree	32	29	26	29	26	26	29	29
total	131	120	108	119	117	117	120	120
% of detectable			90	99	98	98	100	100

Table 8: Number of faults detected by the combination strategies.

"detectable" contains the number of faults for each test object that are detectable by our choice of parameters and values identified from the specification. There are two main reasons why the values of the two columns differ.

As been described in section 4.5 the results of the input parameter modeling form the base for all combination strategies. In our experiment the input parameter models were created before any knowledge of the known faults was acquired. This is the first reason why some of the known faults cannot be detected by any of the combination strategies. The second reason, as been described in section 4.5 is that some identified parameters were ignored due to conflicts with other values.

In total 120 of the 131 known faults are detectable with the used input parameter models.

The most surprising result is that EC is so effective. Despite having so few test cases for each test problem it found 90% of the detectable faults only missing twelve. Examination of the missed faults reveal that all of these faults require two or more parameters to have specific values. Further, samples of the detected faults show that some faults depend only on the value of one single parameter to be detected while other faults depend on values of more than one parameter. These observations are all in line with EC satisfying 1-wise coverage, i.e., any fault depending on the value of one parameter only are guaranteed to be detected by EC. Faults depending on the values of more than one parameter may or may not be detected by EC depending on the combinations that happened to be included in the test suite. In other words, the fault detection ability of the EC combination strategy is sensitive to how the values of the different parameters are combined.

That BC should detect more faults than EC was expected since both combination satisfy 1-wise coverage and the test suites generated by BC contain more test cases than the corresponding test suites generated by EC. More of a surprise was the fact that BC performed similar, with respect to detected faults, to the more demanding pair-wise combination strategies OA and HPW. Again a more detailed look at the faults is needed to explain these results. First it should be noted that the only fault missed by BC was detected, surprisingly enough, by EC and also by both OA and HPW. Let us examine this fault in more detail. This fault is located in the series program. Detection of this fault depends on the values of all three parameters. 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. Six of this combinations will trigger this fault. Parameter one and two both need to have one specific value and parameter three can have any value except one. None of the values of parameters one and two are base choices and this explains why BC missed this fault. Any test case generated by BC contains at the most one non-base choice value. Thus, BC can never detect this fault.

OA and HPW both satisfy pair-wise coverage. This means that all combinations of values of two parameters are 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 with this combination to detect the fault but six of the seven values of the third parameter has this property. Although there are no guarantees that OA and HPW will detect this fault the chances are large (six of seven attempts) that parameter three will have a favorable value.

The chance of EC detecting this fault is small but nevertheless existing, which is proven by the fact that EC actually managed to detect this fault.

The three faults detected by BC alone are all located in the ntree program. Four parameters with six, nine, seven, and seven values were identified for the testing of ntree. For the first fault, initially it looked like only two parameters were involved in causing a failure by activating this fault. The first parameter needed one specific value, that happened to be the base-choice and the second parameter, needed one specific value that was not a base-choice. This would explain why BC would detect the fault. However, OA and HPW satisfying pair-wise coverage should guarantee the detection of this fault, but apparently they both missed. A further analysis revealed that a third parameter was involved indirectly. The first step in the ntree program is a sanity check of the parameter values. If a parameter value is found to be outside the defined scope of the application, the execution is terminated with an error message. Thus, the requirement on the third parameter to be normal in order to even reach the point in the code where the fault was located. Incidentally, the pair-combination of the two first parameters in the test suites of OA and HPW occured in test cases where the third parameter was erroneous, effectively masking this particular fault. In the case of BC the test case in which parameters one and two had the right values, the third value was normal since the base choice is a special case of normal values. Remember that only one parameter at a time deviates from the base choice.

The second and third fault only detected by BC are located in the same source code line. Both faults fail for the same combinations of parameter values. For this fault to result in a failure three parameters need to have one specific value each. This means that neither EC, OA, nor HPW can guarantee the detection of these faults. The results also confirm this theory since neither of these combination strategies detected this fault. However, two of the three required parameter values happened to be base choices, which means that BC is guaranteed to detect this fault. As already been mentioned this is also the case.

OA and HPW perform exactly the same when it comes to detecting faults. This means that combining BC with OA or HPW can also be expected to perform the same. In this experiment both combinations detect all detectable faults.

	Combination Strategy			
Test Object	**EC**	**BC**	**OA**	**HPW**
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 9: Achieved decision coverage for the correct versions of the test objects.

	Combination Strategy			
Test Object	**EC**	**BC**	**OA**	**HPW**
count	[75, 83]	[75, 83]	[75, 83]	[75, 83]
tokens	[48, 86]	[48, 86]	[48, 86]	[48, 86]
series	[76, 90]	[71, 90]	[76, 95]	[76, 95]
nametbl	[43, 100]	[80, 100]	[83, 100]	[83, 100]
ntree	[33, 83]	[33, 88]	[33, 88]	[33, 88]

Table 10: Ranges of achieved decision coverage for the faulty versions of the test objects.

5.4 Decision Coverage

Table 9 shows the decision coverage achieved for the correct versions of the test objects. Table 10 shows the ranges of decision coverage achieved for the faulty versions of each test object.

Our first observation regarding achieved decision coverage is that there is very little difference in the performance of the combination strategies. A slight increase in achieved decision coverage can be seen as the complexity of the combination strategies increases. The reason for this is simply that the more complex combination strategies generate more test cases than the less complex. However, a few faulty versions exhibit a different behavior. A faulty version of the test object series is one example of this. Despite fewer test cases for EC, the minimum achieved decision coverage is higher than the minimum achieved decision coverage for BC. The reason for this is that the EC test suite is not a subset of the BC test suite. Thus, it may be the case that EC but not BC contain test cases that will execute a part of the code uniquely.

Our second observation is that the maximum decision coverage for the faulty versions sometimes exceed the decision coverage for the corresponding correct version. See, for instance, tokens tested by EC. The reason for this is to be found in faults that affect the control flow of the program. Such faults may results in parts of the code, otherwise unexercised by the test suite, to be executed.

The third observation is that EC, despite the low amount of test cases, manages to cover more of the code than we expected. In order to understand this result better we investigated the number of decision points and the maximum number of parallel subpaths due to nesting of decision points

Test Object	Max Parallel paths
count	3
tokens	4
series	4
nametbl	4
ntree	3

Table 11: Maximum number of parallel paths of the test objects

in the test objects. The two outcomes of a decision point represent two parallel subpaths since the execution of both these subpaths requires two different sets of values entering the decision point. Thus, nesting decision points will increase the number of parallel subpaths. The maximum number of parallel subpaths corresponds to the maximum nesting level and is a theoretical lower bound for the number of test cases needed to reach 100% decision coverage since parallel paths represent different outcomes of decisions. In test objects with many decision points there is an increased chance of dependencies between the decision points, possibly increasing the number of test cases needed in practice to achieve 100% decision coverage.

Table 11 shows the maximum number of subpaths for the test objects. From table 6 it can be deduced that all EC test suites contain enough test cases, at least in theory, to reach 100% decision coverage. The high decision coverage achieved for all test objects 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 objects.
2) The selected parameters and parameter values used for testing the test objects are good representatives of the total input space.
3) The actual test cases generated by EC are well scattered over the implementations.

5.5 Test Suite Failure Density

The failure density i.e., the percentage of test cases in a test suite that fails for a certain fault, reveals that BC behaves very differently from EC, OA, and HPW. Tables 12 and 13 contain the failure densities of the 26 detectable faults in the original benchmark suite. A high percentage indicates that many of the test cases failed due to a certain fault. 0 indicates that none of the test cases failed for that fault.

For 25 of the 26 investigated faults BC has either the highest or the lowest failure density. In many of these cases, for instance count1, series4, and ntree7 the failure densities of BC differ largely from the others. In all combination strategies except BC the distributions of the parameter values in the test suites are approximately uniform, that is, all parameter values occur about the same number of times in the test suite. In a BC test suite the base-choice values are over-represented since every test case is derived from the base test case, changing only one parameter value.

Faults that are triggered by base-choice values will result in BC having a higher failure density

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
HPW	33	75	100	67	25	31	6	17	11	6	20

Table 12: % of test cases that failed for a each fault for the different combination strategies applied to the test objects count, tokens, and series.

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
HPW	2	69	26	30	57	13	15	43	0	23	12	31	27	28	58

Table 13: % of test cases that failed for a each fault for the different combination strategies applied to the test objects nametbl and ntree.

that the other combination strategies due to this over-representation of base-choice values. The opposite is also true, when faults are triggered by non base-choice values BC will have lower fault density than the other combination strategies for the same reason.

6 Discussion and Conclusions

The following sections discuss the results described in the previous sections. Some conclusions are formulated based on these results.

6.1 Application of Combination Strategies

As was noted in sections 4.3 and 5.3 some faults remain undetected by all of the combination strategies. The major reason for this is our choice of parameters and parameter values and our attempt to avoid conflicting values. Despite this all of the investigated combination strategies detect a vast majority of the faults. Our first conclusion from this is that further research into the applicability of combination strategies is well motivated. However, our results also indicate that some faults may be hard or even impossible to detect by the sheer use of combination strategies. The fact that we only reach 100% decision coverage in one of the five test objects is an argument for this. The same observation is also made by Burr and Young [BY98]. Hence, our second conclusion is that combination strategies should not be used in isolation, but employed with for instance code coverage based test methods.

6.2 Parameter Conflicts

In our experiment, conflicts between parameter values has been solved by ignoring some parameter and values, which otherwise should have been included. This is considered a minor problem in the scope of our experiment. The reason is that all strategies are treated equally. However, in a real testing problem, parameters and values cannot be ignored. Hence these conflicts have to be handled properly. The HPW and BC algorithms have built-in functionality to handle such conflicts but none of the conflict handling methods built into these algorithms are general enough to be used with any combination strategy. Further, Cohen et al. [CDFP97] suggest a general mechanism to handle conflicts. The basic idea is that one big test problem with conflicts are transformed to several smaller test problems without conflicts. Their conclusion is that the general solution is more expensive than the HPW built-in function. It is obvious that conflicts need to be handled but how to do this is considered out of scope for our investigation.

6.3 Properties of Combination Strategies

The performance of a combination strategy depends to a large extent on the nature and number of the faults in the test object. Knowledge about the faults in a test object is generally not known in advance, thus it is impossible to know the ultimate combination strategy for a test object. However by analyzing the faults included in this experiment with respect to how the different

investigated combination strategies behaved, it is possible to make some general statements about the combination strategies abilities to detect different types of fault.

Many of the faults detected by EC were detected by coincidence. This leads us to belive that the results of applying EC is too unpredictable to be really useful for the tester.

The difference between BC and all of the other combination strategies is the possibility to include some semantic information into the test case selection mechanism. The appointment of a base choice value for each parameter makes it possible for the tester to affect the contents of the generated test suite. Ammann and Offutt recommend the base choice value to reflect the usage by for instance using the most commonly used value as the base-choice [AO94]. This focus on the end-user is one of two main strengths of BC. It means that faults in commonly used parts of the program are more likely to be detected than faults in remote parts of the source code. The second main strength of BC is that no test case will contain more than one invalid parameter value, eliminating masking of faults. This conclusion is based on the faults detected only by BC.

The main weakness of BC is accentuated in test problems with many valid values of each parameter. BC will not generate any test case with more than one value differing from the base-choice. Each non-base-choice value any the parameter will only occur in one single test case. The end result for these values is only slightly better than for EC. Nevertheless we belive that BC is a worthwhile combination strategy to use, especially for test problems with few valid values of each parameter. BC is also interesting in situations when there is not enough time to use a more complex strategy.

OA and HPW performs very similar in all aspects. Their main strength is the guarantee to satisfy pair-wise coverage. This is especially important for test objects where there are many valid values of each parameter, i.e., exactly in the case when BC performs its worst. The main drawback with these two combination strategies is the risk of masking faults. Both strategies try to minimize the number of test cases while reaching pair-wise coverage. This means that most test cases are designed to cover many parameter value pairs. In particular when several parameters contain values that are invalid the effects of some parameters may be masked by others. Thus, the effect of a certain pair may never be found out despite the fact that the pair is included in the test suite.

Although the performance of OA and HPW makes it impossible to decide which one should be used there are other factors which favors HPW from a tester's perspective. The most important being the possibility automate the whole test case generation process. Trying to automate the work with OA seems to be much more difficult than programming the algorithm presented in fig. 5. Another aspect that favor HPW is its greater generality. Not only its ability to handle test problems of any size but also the possibilities of extending the algorithm to t-wise coverage where t is an arbitrary number speaks in favor of HPW over OA.

Another argument in favor of using HPW is that it is possible to with HPW to start from an existing test suite and just add enough test cases to satisfy pair-wise coverage. For instance, it would be possible to create a BC test suite and then use HPW to create a test suite satisfying single error coverage and pair-wise coverage at the same time.

Thus, we reach our conclusion that when the time budget permits, BC and HPW should be combined to get the best of the two worlds. BC sets focus on the end-user and guarantees the detection of all faults relating to a single parameter being invalid while HPW adds to the confidence

in detecting the faults that reside in code that is not most likely used but still important.

6.4 Input Parameter Modeling

A final observation worth discussing relates to the input parameter models created. In this experiment equivalence class partitioning was used to identify both actual and abstract parameters of the test problem. When identifying equivalence classes and their corresponding values, the test engineer sometimes can choose between representing a certain aspect of the test problem as an own parameter or including that aspect in an already existing parameter increasing the number of values of that parameter. By examining the time complexity of the combination strategies we reach the conclusion that when using any of the combination strategies EC, BC, OA, or HPW it is favorable with respect to the number of test cases to add a new parameter over increasing the number of values of an existing parameter. In other words, from the perspective of cost more parameters with fewer values are better than fewer parameters with more values.

6.5 Recommendations

The following recommendations summarize our findings.

- Combination strategies are viable test methods but need to be complemented by code coverage based test methods.
- When time is a scarce resource, use BC. The reasons are its low cost and is its user orientation.
- When there is enough time, use BC in combination with HPW. In this case BC is used as a guarantee for freeness of masking as a complement to the pair-wise coverage achieved by HPW.
- 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

The foundation of Combination strategies is the analysis of parameters one-by-one. Equivalence Partitioning [Mye79] and Boundary Value Analysis [Mye79] are well known and frequently used test case selection methods [Rei97]. In the basic forms of these two test case selection methods, the input spaces of each parameter are analyzed and partitioned one by one to identify a small number of values of each parameter that are interesting to test. For test problems containing more than one variable, these methods offer little information on how to combine the values of the different parameters into complete inputs of test cases.

The category partition method [OB88] starts to address the issue of how to combine values of different parameters to form complete test cases. In particular the user may specify constraints on how to combine values by disallowing certain combinations to be used. Apart from the possibility to use constraints, the category partition method offers little insight into which combinations to use whenever the set of possible combinations is infeasibly large.

A formalized version of boundary value analysis called domain testing [Bei90] specifies how complete inputs of test cases may be chosen when the input space of the test object is subdivided into domains by continuous mathematical functions intersecting each other. Domain testing is not well suited for test problems defined by discrete functions. Also some classes of continuous functions are difficult to handle by domain testing. For instance when a domain is concave, when the domain contains "holes", or when a domain is under-specified. These constraints limit the generality of the method.

The seminal works on the combination strategies evaluated in this report has primarily been aimed at describing the strategies and their applicability to test problems. Thus, little focus has been placed on evaluations of the different combination strategies which is the main focus of this work.

The seminal work on BC by Ammann and Offutt [AO94] theoretically compares the number of test cases needed to fulfill base-choice coverage with the number of test cases needed to fulfill 1-wise coverage and n-wise coverage, see sections 3.1 and 3.5 for more details of the coverage criteria. Ammann and Offutt also demonstrate the feasibility of BC by a small experiment in which a complete test suite was generated for a system. The test suite contained 72 test cases and detected 7 out of 10 known faults. Being too sparse with details of the experiment it is difficult to compare the results with the experiment performed in this paper. However, the fact that not all faults were detected seems to point in the same direction as this experiment, i.e., that BC should not be used alone if there are high requirements on the test quality.

HPW by Cohen et al. [CDFP97] is a heuristic algorithm, which in each step adds one new test case to the test suite by evaluating a number of candidates and choosing the best one. The authors claim that evaluating more than 50 candidates for each test case has no significant effects on the number of test cases in the final test suite. Our experiment is based on evaluating exactly 50 candidates. HPW generating the same number of test cases or less than OA in four of the five is in line with Cohen et al.'s claim since OA in some cases even finds the minimal test suite for pair-wise coverage [WP96].

In describing HPW, Cohen et al. reports the results of two experiments. The reported results of the first experiment is that faults were found using HPW despite the fact that the test object had been tested prior to the experiment. Faults were detected both in the code and in the specifications. Too little information is given for this experiment to make any comparisons with this experiment. In the second experiment number of test cases and achieved code coverage were monitored for the HPW, "All" and random choice. Our assumption is that the strategy called "all" by Cohen is the same as our AC but the given information in the paper is not enough to be certain. Cohen report 200 test cases for HPW compared to 436 test cases for All. This result is significantly different from our results, if our assumption that "all" is the same as AC. In our case the number of test cases for AC is an order of magnitude more than for HPW. Achieved decision coverage for both HPW and All are 85%. The achieved decision coverage for HPW is totally in line with our results. Further, not achieving 100% for All is again an indication that the sole use of combination strategies does not guarantee the detection of all faults. Further interpretations of the results reported by Cohen et al. are hard to make due to lack of information.

Brownlie, Prowse, and Phadke [BPP92] used OA to test a PMX/StarMAIL release. The faults found were analyzed and 12% of them were deemed unlikely to be found by conventional

testing. According to the authors, this shows the superiority of the orthogonal arrays strategy over a hypothesized conventional testing technique. Not enough details are given on the performed experiment for the reader to judge this claim. Brownlie et al. also report that 68 test cases had to created by hand in addition to the 354 test cases generated by OA to cover special conditions and tests not applicable to all configuration parameters. Again this is in line with our claim that using combination strategies alone is not a guarantee for good test quality.

In an experiment Dunietz, Ehrlich, Szablak, Mallows, and Iannino [DES+97] evaluated the effect of balanced t-wise coverage of input combinations on achieved code coverage. They show that with $t \geq 2$, t-wise coverage and AC obtains comparable results in terms of statement coverage. This is in line with the results of the experiment performed by Cohen et al. previously mentioned. Further Dunietz et al. show that for t-wise coverage and AC to achieve comparable path coverage higher values of t are needed.

8 Contributions

The main contributions of this research are:

- A comprehensive description and comparison of a number of combination strategies.
- The description of a repeatable experiment comparing test methods.
- The formulation of some recommendations on the applicability of the combination strategies.
- Support for some claims and observations in previous research.

Specifically, our work supports the claim by Cohen et al. [CDFP97] that evaluating more than 50 candidates for each test case has no significant effects on the number of test cases in the final test suite. Also our results as well as previous work indicate that combination strategies, in general, should be combined with other test case selection methods to increase the effectiveness of the testing.

In addition, we have also introduced a new metric called failure density, which has been shown to be a good tool for detecting and explaining the difference in behavior of the investigated combination strategies.

9 Future Work

Although this investigation has given some interesting results, it is far from answering the question whether or not combination strategies are feasible in a real test setting. To answer this question a number of issues need to be investigated.

To start with, the user needs a repeatable input parameter modeling method. It is not clear that equivalence class partitioning is the optimal choice.

Secondly, conflicts among parameter values need to be handled in the general case. Some of the combination strategies investigated in this report contain specific conflict handling methods but it is unclear if these are possible to generalize to fit an arbitrary combination strategy. Further, the performance of these different conflict handling strategies need to be investigated.

Thirdly, are there other test methods that are more effective and efficient than combination strategies? Thus it would be interesting to perform the same or a similar experiment using other test strategies. An example of such an experiment is the one performed by Reid [Rei97]. Performing this experiment with the test strategies investigated by Reid or performing his experiment with the combination strategies in this experiment would enable interesting possibilities of cross-experimental comparison.

Fourthly, the contents of this experiment, i.e., programs and faults, are probably not representative of real testing problems. To be able to make a statement of the feasibility of combination strategies it is necessary to investigate combination strategies in an industrial setting. Not only to assess the effectiveness but also to assess the true cost of applying combination strategies.

The tools used for automating these experiments can be refined and further developed to be more general. It is interesting to identify the main requirements on a tool for automatic generation and execution of test cases.

In addition to these issues related work involves trying to formalize the description of different types of faults. It is would also be interesting to examine variants of combination strategies for instance to use the complete BC test suite as input to the HPW algorithm.

10 Acknowledgments

First and foremost we owe our gratitude to Dr Christopher Lott for generously supplying all the material concerning the test objects and being very helpful in answering our questions.

Then we are also indebted to many of our colleagues at 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 for supplying valuable tips during the set up of the compiler environment. Sanny Gustavsson, Jonas Mellin, and Gunnar Mathiason for general support and good ideas. Also we are very grateful for all the feed-back on this report.

11 Bibliography

References

[AAC+94] T. Anderson, A. Avizienis, W.C. Carter, A. Costes, F. Christian, Y. Koga, H. Kopetz, J.H. Lala, J.C. Laprie, J.F. Meyer, B. Randell, and A.S. Robinson. Dependability: Basic Concepts and Terminology. Technical report, IFIP. WG 10.4, 1994.

[AO94] Paul E. Ammann and A. Jefferson Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS'94),Gaithersburg MD*, pages 69–80. IEEE Computer Society Press, June 1994.

[Arc92] Russel D. Archibald. *Managing High-Technology Programs and Projects.* John Wiley and sons, Inc, 1992.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[BP99] Lionel Briand and Dietmar Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proceedings of the International Conference on Software Maintenance (ICSM99), 30th of Aug - 3rd Sept, 1999, Oxford, The UK*, pages 475–482, 1999.

[BPP92] Robert Brownlie, James Prowse, and Mahdav S. Phadke. Robust Testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, May/June 1992.

[BR88] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, SE-14(6):758–773, June 1988.

[BS87] Victor R. Basili and Richard W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, December 1987.

[BSL99] Victor Basili, Forrest Shull, and Filippo Lanubile. Using experiments to build a body of knowledge. In *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference (PSI 99), Akademgorodok, Novosibirsk, Russia, July 6-9, 1999, Proceedings*, pages 265–282, 1999.

[BY98] Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generatio and code coverage. In *Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR'98), San Diego, CA, USA, October 26-28, 1998*, 1998.

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (AETG) system. In *Proceedings of Fifth International Symposium on Software Reliability Engineering (ISSRE'94), Los Alamitos, California, USA, November 6-9, 1994*, pages 303–309. IEEE Computer Society, 1994.

[CDPP96] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, 13(5):83–89, September 1996.

[DES+97] 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 19th International Conference on Software Engineering (ICSE'97), Boston, MA, USA 1997*, pages 205–215. ACM, May 1997.

[FHLS98] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Stringini. Evaluating Testing Methods by Delivered Reliability. *IEEE Transactions on Software Engineering*, 24:586–601, August 1998.

[Het76] William C. Hetzel. *An Experimental Analysis of Program Verification Methods.* PhD thesis, University of North Carolina, Chapel Hill, 1976.

[HHH+99] Mark Harman, Robert Hierons, Mike Holocombe, Bryan Jones, Stuart Reid, Marc Roper, and Martin Woodward. Towards a maturity model for empirical studies of software testing. In *Proceedings of the 5th Workshop on Empirical Studies of Software Maintenance (WESS'99)*. Keble College, Oxford, UK, September 1999.

[How78] William E. Howden. An evaluation of the effectiveness of symbolic testing. *Software-Practice and Experience*, 8(4):381–397, 1978.

[KKS98] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of FTCS'98: Fault Tolerant Computing Symposium, June 23-25, 1998 in Munich, Germany*, pages 230–239. IEEE, 1998.

[KL95a] Erik Kamsties and Christoffer Lott. An Empirical Evaluation of Three Defect Detection Techniques. Technical Report ISERN 95-02, Dept of Computer Science, University of Kaiserslauten, May 1995.

[KL95b] Erik Kamsties and Christoffer Lott. An empirical evaluation of three defect detection techniques. In *Proceedings of the 5th European Software Engineering Conference (ESEC95), Sitges, Barcelona, Spain, September 25-28, 1995*, September 1995.

[LR96] Christopher M. Lott and H. Dieter Rombach. Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques. *Journal of Empirical Software Engineering*, 1(3):241–277, 1996.

[Man85] Robert Mandl. Orthogonal Latin Squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, October 1985.

[MRWB95] James Miller, Marc Roper, Murray Wood, and Andrew Brooks. Towards a benchmark for the evaluation of software testing techniques. *Information and Software Technology*, 37(1):5–13, 1995.

[Mye78] Glenford J. Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspection. *Communcations of the ACM*, 21(9):760–768, September 1978.

[Mye79] Glenford J. Myers. *The art of Software Testing.* John Wiley and Sons, 1979.

[Nta84] Simeon C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, SE-10:795–803, nov 1984.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[OXL99] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Fifth IEEE International Conference on Engineering of Complex Systems (ICECCS'99),Las Vegas NV*, pages 119–129. IEEE, October 1999.

[Rei97] Stuart Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings of the 4th International Software Metrics*

Symposium (METRICS'97), Albaquerque, New Mexico, USA, Nov 5-7, 1997, pages 64–73. IEEE, 1997.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, april 1985.

[SCSK02] Sun Sup So, Sung Deok Cha, Timothy J. Shimeall, and Yong Rae Kwon. An empirical evaluation of six methods to detect faults in software. *Software Testing, Verification and Reliability*, 12(3):155–171, September 2002.

[WP96] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE 96), White Plains, New York, USA, Oct 30 - Nov 2, 1996*, pages 246–254, Nov 1996.

[WRBM97] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques: A replicated study. In *Proceedings of the Sixth European Software Engineerng Conference (ESEC/FSE 97)*, pages 262–277. Springer Verlag, Lecture Notes in Computer Science Nr. 1013, September 1997.

[ZH92] Stuart H. Zweben and Wayne D. Heym. Systematic Testing of Data Abstractions Based on Software Specifications. *Journal of Software Testing, Verification, and Reliability*, 1(4):39–55, 1992.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

A Test Cases

The following tables contain the test cases generated by the combination strategies for the test objects.

TC	Parameter values			
	EC	BC	OA	HPW
TC1	0 0 0 0 0 0	0 2 1 1 0 2	0 0 0 0 0 0	0 2 1 1 0 2
TC2	1 1 1 1 1 1	1 2 1 1 0 2	0 1 1 0 1 2	1 0 0 0 0 0
TC3	0 2 1 1 2 2	0 0 1 1 0 2	0 2 1 0 2 1	0 1 1 0 1 1
TC4	-	0 1 1 1 0 2	1 0 0 1 0 0	1 2 0 1 2 1
TC5	-	0 2 0 1 0 2	1 1 1 1 1 2	0 0 1 1 2 0
TC6	-	0 2 1 0 0 2	1 2 1 1 2 1	1 1 0 1 1 2
TC7	-	0 2 1 1 1 2	0 0 0 1 1 1	0 0 0 0 2 2
TC8	-	0 2 1 1 2 2	0 0 0 1 2 2	0 1 0 0 0 0
TC9	-	0 2 1 1 0 0	1 1 1 0 0 0	0 2 0 0 1 0
TC10	-	0 2 1 1 0 1	1 1 1 1 1 1	0 0 0 0 1 1
TC11	-	-	1 1 1 1 2 2	1 0 1 0 0 1
TC12	-	-	0 2 1 0 0 0	0 1 0 0 2 0
TC13	-	-	0 2 1 1 1 1	-
TC14	-	-	0 2 1 1 2 2	-

Table 14: Test cases for the count test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC1	0 0 0 0	2 2 2 1	0 0 0 0	2 2 2 1
TC2	1 1 1 1	0 2 2 1	0 1 6 1	0 0 0 0
TC3	2 2 2 2	1 2 2 1	0 2 5 2	0 1 1 2
TC4	3 3 3 3	3 2 2 1	0 3 4 3	1 0 1 3
TC5	4 4 4 4	4 2 2 1	0 4 3 4	1 1 0 4
TC6	5 5 5 1	5 2 2 1	0 5 2 1	3 0 3 1
TC7	6 6 6 1	6 2 2 1	0 6 1 1	4 1 2 0
TC8	-	2 0 2 1	1 0 1 1	5 0 2 2
TC9	-	2 1 2 1	1 1 0 2	6 0 4 4
TC10	-	2 3 2 1	1 2 6 3	1 3 3 0
TC11	-	2 4 2 1	1 3 5 4	2 4 1 0
TC12	-	2 5 2 1	1 4 4 1	3 5 4 0
TC13	-	2 6 2 1	1 5 3 1	2 6 0 2
TC14	-	2 2 0 1	1 6 2 0	0 2 5 3
TC15	-	2 2 1 1	2 0 2 2	0 3 6 1
TC16	-	2 2 3 1	2 1 1 3	4 4 0 1
TC17	-	2 2 4 1	2 2 0 4	5 1 3 3
TC18	-	2 2 5 1	2 3 6 1	6 6 5 0
TC19	-	2 2 6 1	2 4 5 1	6 5 0 3
TC20	-	2 2 2 0	2 5 4 0	5 2 6 0
TC21	-	2 2 2 2	2 6 3 1	0 4 2 4
TC22	-	2 2 2 3	3 0 3 3	3 3 5 2
TC23	-	2 2 2 4	3 1 2 4	4 2 3 2
TC24	-	-	3 2 1 1	1 5 5 1
TC25	-	-	3 3 0 1	3 6 1 4
TC26	-	-	3 4 6 0	5 6 4 1
TC27	-	-	3 5 5 1	1 4 6 2
TC28	-	-	3 6 4 2	2 5 3 4
TC29	-	-	4 0 4 4	4 3 4 3
TC30	-	-	4 1 3 1	6 1 1 1

Table 15: Test cases 1(2) for the nametbl test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC31	-	-	4 2 2 1	2 6 6 3
TC32	-	-	4 3 1 0	3 4 2 3
TC33	-	-	4 4 0 1	4 0 5 4
TC34	-	-	4 5 6 2	5 3 0 4
TC35	-	-	4 6 5 3	6 3 2 2
TC36	-	-	5 0 5 1	1 2 4 4
TC37	-	-	5 1 4 1	4 5 1 2
TC38	-	-	5 2 3 0	3 1 6 4
TC39	-	-	5 3 2 1	2 1 4 2
TC40	-	-	5 4 1 2	6 4 3 0
TC41	-	-	5 5 0 3	1 6 2 0
TC42	-	-	5 6 6 4	0 6 3 0
TC43	-	-	6 0 6 1	5 4 5 0
TC44	-	-	6 1 5 0	3 2 0 0
TC45	-	-	6 2 4 1	0 5 2 0
TC46	-	-	6 3 3 2	4 0 6 0
TC47	-	-	6 4 2 3	2 0 5 0
TC48	-	-	6 5 1 4	5 2 1 0
TC49	-	-	6 6 0 1	6 2 6 0
TC50	-	-	-	2 3 1 0
TC51	-	-	-	5 5 6 0
TC52	-	-	-	0 4 4 0
TC53	-	-	-	4 6 0 0
TC54	-	-	-	0 1 5 0

Table 16: Test cases 2(2) for the nametbl test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC1	0 0 0 0	1 3 1 1	0 0 0 0	1 3 1 1
TC2	1 1 1 1	0 3 1 1	0 3 6 3	0 0 0 0
TC3	2 2 2 2	2 3 1 1	0 4 5 4	2 1 0 1
TC4	3 3 3 3	3 3 1 1	0 5 4 5	3 1 1 0
TC5	4 4 4 4	4 3 1 1	0 6 3 6	4 2 2 0
TC6	5 5 5 5	5 3 1 1	1 0 1 1	5 2 0 2
TC7	1 6 6 6	1 0 1 1	1 1 0 2	2 3 3 0
TC8	1 7 1 1	1 1 1 1	1 2 1 3	0 2 4 1
TC9	1 8 1 1	1 2 1 1	1 3 1 4	1 4 5 0
TC10	-	1 4 1 1	1 4 6 5	1 0 6 2
TC11	-	1 5 1 1	1 5 5 6	2 0 1 3
TC12	-	1 6 1 1	1 6 4 1	3 0 2 4
TC13	-	1 7 1 1	1 7 3 1	4 0 3 5
TC14	-	1 8 1 1	1 8 2 0	1 5 0 6
TC15	-	1 3 0 1	2 0 2 2	5 0 4 6
TC16	-	1 3 2 1	2 1 1 3	0 6 1 2
TC17	-	1 3 3 1	2 2 0 4	0 7 2 3
TC18	-	1 3 4 1	2 3 1 5	5 8 6 0
TC19	-	1 3 5 1	2 4 1 6	0 1 5 4
TC20	-	1 3 6 1	2 5 6 1	3 3 0 5
TC21	-	1 3 1 0	2 6 5 1	4 5 5 1
TC22	-	1 3 1 2	2 7 4 0	0 4 6 5
TC23	-	1 3 1 3	2 8 3 1	1 6 2 5
TC24	-	1 3 1 4	3 0 3 3	4 7 0 4
TC25	-	1 3 1 5	3 1 2 4	3 8 3 1
TC26	-	1 3 1 6	3 2 1 5	4 1 4 2
TC27	-	-	3 3 0 6	1 1 3 3
TC28	-	-	3 6 6 0	4 4 1 6
TC29	-	-	3 7 5 1	2 2 5 5
TC30	-	-	3 8 4 2	5 4 2 1

Table 17: Test cases 1(3) for the ntree test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC31	-	-	4 0 4 4	5 5 1 4
TC32	-	-	4 1 3 5	2 6 4 4
TC33	-	-	4 2 2 6	1 7 4 0
TC34	-	-	4 3 1 1	2 8 2 2
TC35	-	-	4 4 0 1	2 7 6 6
TC36	-	-	4 5 1 0	3 2 6 3
TC37	-	-	4 6 1 1	3 3 5 2
TC38	-	-	4 7 6 2	0 5 3 2
TC39	-	-	4 8 5 3	4 6 0 3
TC40	-	-	5 0 5 5	0 8 5 6
TC41	-	-	5 1 4 6	5 3 4 3
TC42	-	-	5 2 3 1	2 4 0 2
TC43	-	-	5 3 2 1	2 5 2 0
TC44	-	-	5 4 1 0	3 6 3 6
TC45	-	-	5 5 0 1	5 7 1 5
TC46	-	-	5 6 1 2	1 8 0 4
TC47	-	-	5 7 1 3	4 3 6 4
TC48	-	-	5 8 6 4	5 1 2 6
TC49	-	-	1 0 6 6	1 2 3 4
TC50	-	-	0 1 5 1	3 4 4 3
TC51	-	-	0 2 4 2	3 5 4 5
TC52	-	-	1 3 3 0	5 6 5 0
TC53	-	-	3 4 2 1	3 7 3 1
TC54	-	-	3 5 1 2	4 8 1 3
TC55	-	-	1 6 0 3	0 6 6 1
TC56	-	-	0 7 2 4	0 3 2 6
TC57	-	-	0 8 1 5	0 0 5 3
TC58	-	-	1 1 6 0	0 1 6 5
TC59	-	-	1 2 5 0	0 2 1 6
TC60	-	-	1 3 4 1	5 4 3 4

Table 18: Test cases 2(3) for the ntree test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC61	-	-	1 4 3 2	0 5 6 3
TC62	-	-	1 5 2 3	0 7 5 2
TC63	-	-	1 6 1 4	0 8 4 5
TC64	-	-	1 7 0 5	-
TC65	-	-	1 8 0 6	-
TC66	-	-	1 2 6 1	-
TC67	-	-	1 3 5 2	-
TC68	-	-	1 4 4 3	-
TC69	-	-	1 5 3 4	-
TC70	-	-	1 6 2 5	-
TC71	-	-	1 7 1 6	-

Table 19: Test cases 3(3) for the ntree test object generated by the combination strategies

B Faults

This section contains the faults used in this experiment. The faults are described in their original context i.e., the program listings. However, the faults included in the original benchmark suite by Lott et al. have been omitted since publishing these faults would ruin the intention of the original experiment. Information about these faults may be obtained from the web site describing the repeatable software experiment[2].

B.1 tokens

```
/*COPYRIGHT (c) Copyright 1992 Gary Perlman */

#include <stdio.h>
#include <string.h>
#include <assert.h>

int     Ignore = 0;
int     Mincount = 0;
int     Alpha = 0;
char    MapAllowed[256];

typedef struct tnode
{
```

[2]URL: www.chris-lott.org/work/exp/

TC	Parameter values			
	EC	BC	OA	HPW
TC1	0 0 0	2 2 4	0 0 0	2 2 4
TC2	1 1 1	0 2 4	0 1 6	0 0 0
TC3	2 2 2	1 2 4	0 2 5	1 0 1
TC4	3 3 3	3 2 4	0 3 4	3 0 2
TC5	4 4 4	4 2 4	0 4 3	4 0 3
TC6	2 2 5	2 0 4	1 0 1	0 1 1
TC7	2 2 6	2 1 4	1 1 0	0 3 2
TC8	-	2 3 4	1 2 6	0 4 3
TC9	-	2 4 4	1 3 5	1 1 0
TC10	-	2 2 0	1 4 4	2 0 5
TC11	-	2 2 1	2 0 2	3 2 0
TC12	-	2 2 2	2 1 1	4 3 0
TC13	-	2 2 3	2 2 0	2 4 0
TC14	-	2 2 5	2 3 6	0 2 6
TC15	-	2 2 6	2 4 5	1 2 2
TC16	-	-	3 0 3	3 1 3
TC17	-	-	3 1 2	4 1 2
TC18	-	-	3 2 1	1 3 3
TC19	-	-	3 3 0	3 4 1
TC20	-	-	3 4 6	0 0 4
TC21	-	-	4 0 4	0 1 5
TC22	-	-	4 1 3	1 4 6
TC23	-	-	4 2 2	2 3 1
TC24	-	-	4 3 1	4 2 1
TC25	-	-	4 4 0	1 1 4
TC26	-	-	3 0 5	1 2 5
TC27	-	-	3 1 4	2 1 6
TC28	-	-	1 2 3	3 3 4
TC29	-	-	0 3 2	4 4 4

Table 20: Test cases 1(2) for the series test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC30	-	-	0 4 1	3 3 5
TC31	-	-	4 0 6	3 0 6
TC32	-	-	4 1 5	2 4 2
TC33	-	-	2 2 4	4 4 5
TC34	-	-	2 3 3	2 2 3
TC35	-	-	1 4 2	4 3 6

Table 21: Test cases 2(2) for the series test object generated by the combination strategies

TC	Parameter values			
	EC	BC	OA	HPW
TC1	0 0 0 0 0 0 0 0	0 0 0 0 2 2 1 1	0 0 0 0 0 0 0 0	0 0 0 0 2 2 1 1
TC2	1 1 1 1 1 1 1 1	1 0 0 0 2 2 1 1	0 1 1 0 1 1 0 1	1 1 0 1 0 0 0 0
TC3	0 0 2 2 2 2 1 1	0 1 0 0 2 2 1 1	0 0 2 0 2 2 0 1	0 1 1 2 1 1 1 0
TC4	0 0 0 3 2 2 1 1	0 0 1 0 2 2 1 1	1 0 1 1 0 1 1 0	1 0 2 3 1 1 0 1
TC5	-	0 0 2 0 2 2 1 1	1 1 2 1 1 2 1 1	0 0 1 2 0 0 0 1
TC6	-	0 0 0 1 2 2 1 1	1 0 0 1 2 0 1 1	1 1 2 0 2 2 0 0
TC7	-	0 0 0 2 2 2 1 1	0 0 2 2 0 2 1 0	0 0 2 1 1 0 1 0
TC8	-	0 0 0 3 2 2 1 1	0 1 1 2 1 1 1 1	1 1 1 3 0 2 1 0
TC9	-	0 0 0 0 0 2 1 1	0 0 0 2 2 0 1 1	0 1 1 1 2 1 0 1
TC10	-	0 0 0 0 1 2 1 1	0 1 0 3 1 0 1 1	0 0 0 0 0 1 0 0
TC11	-	0 0 0 0 2 0 1 1	0 0 2 3 0 2 1 0	1 0 0 2 1 2 0 0
TC12	-	0 0 0 0 2 1 1 1	0 0 0 1 1 1 1 1	0 0 0 3 2 0 0 0
TC13	-	0 0 0 0 2 2 0 1	0 0 0 2 2 2 1 1	0 0 1 0 1 0 0 0
TC14	-	0 0 0 0 2 2 1 0	0 0 0 3 2 2 1 1	0 0 2 2 0 0 0 0
TC15	-	-	1 1 1 0 0 0 1 1	0 0 0 1 0 2 0 0
TC16	-	-	1 1 1 1 1 1 1 1	0 0 0 2 2 0 0 0
TC17	-	-	1 1 1 2 2 2 1 1	-
TC18	-	-	1 1 1 3 2 1 0 0	-
TC19	-	-	0 0 2 0 0 0 1 1	-
TC20	-	-	0 0 2 2 2 2 0 0	-
TC21	-	-	0 0 2 3 2 2 1 1	-
TC22	-	-	0 0 2 1 1 1 0 0	-

Table 22: Test cases for the tokens test object generated by the combination strategies

```
  char   *contents;
  int    count;
  struct tnode *left;
  struct tnode *right;
} TNODE;

void treeprint(tree) TNODE *tree;
{
  if (tree != NULL)
    {
      treeprint(tree->left);
      /* if (tree->count == Mincount)      / * FAULT 1 */
      if (tree->count >= Mincount)     /* Corrects FAULT 1 */
        {
          printf("%7d\t%s\n", tree->count, tree->contents);
        }
      else
      treeprint(tree->right);     /* Corrects FAULT 2 */
      /* treeprint(tree->left);        / * FAULT 2 */
    }
}

TNODE *
install(string, tree) char *string; TNODE * tree;
{
  int cond;
  assert(string != NULL);  /* Corrects FAULT 3 */
  /* FAULT 3 - Remove the assert statement above */
  if (tree == NULL)
    {
      if (tree = (TNODE *) calloc(1, sizeof(TNODE)))
        {
          tree->contents = strdup(string);
          tree->count = 1; /* Corrects FAULT 4 */
          /* tree->count = 0; / * FAULT 4 */
        }
    }
  else
    {
      cond = strcmp(string, tree->contents);
      if (cond < 0)
        {
          tree->left = install(string, tree->left);
```

```
        }
      else
        {
          if (cond == 0) /* Corrects FAULT 5 */
            /* if (cond = 0) / * FAULT 5 */
            {
              tree->count++;
            }
          else
            {
              tree->right = install(string, tree->right);
            }
        }
    }
  return(tree);
}

char *
getword(ioptr) FILE *ioptr;
{
  static char string[1024];
  char *ptr = string;
  register int c;
  assert(ioptr != NULL);
  for (;;)
    {
      c = getc(ioptr);
      if (c == EOF)
        {
          if (ptr == string)
            {
              return(NULL); /* Corrects FAULT 6 */
              /* FAULT 6 - remove the return statement above */
            }
          else
            {
              break;
            }
        }
      else

      if (!MapAllowed[c])
        {
```

```
          if (ptr == string) /* Corrects FAULT 7 */
            /* if (ptr = string) / * FAULT 7 */
          {
            continue;
          }
        else
          {
            break;
          }
        }
      else
      *ptr++ = MapAllowed[c];
    }
  *ptr = '\0';
  return(string);
}

void tokens(ioptr) FILE *ioptr;
{
  TNODE *root = NULL;
  char *s;
  assert(ioptr != NULL);
  while (s = getword(ioptr))
    root = install(s, root);
  treeprint(root);
}

int main(argc, argv) int argc; char ** argv;
{
  int c, errcnt = 0;
  extern char *optarg;

  int i;
  for (i=0;i<256;MapAllowed[i++]=0);  /* Corrects FAULT8 and FAULT9 */
  /* for (i=0;i<255;MapAllowed[i++]=0);   / * FAULT8 */
  /* for (i=0;i<256;MapAllowed[++i]=0); / * FAULT9 */

  while ((c = getopt(argc, argv, "ac:im:")) != EOF)
    switch(c)
      {
      case 'a':  Alpha = 1; break;
      case 'c':
        while (*optarg)
```

```
          {
            MapAllowed[*optarg] = *optarg; /* Corrects FAULT10 */
            /* MapAllowed[*optarg] = *(++optarg); / * FAULT10 */
            optarg++;
          }
        break;
      case 'i': Ignore = 1; break; /* Corrects FAULT11 */
          /* case 'i': Ignore = 0; break; / * FAULT11 */
      case 'm': Mincount = atoi(optarg); break; /* Corrects FAULT12 */
        /* case 'm': Mincount = atoi(optarg++); break; / * FAULT12 */
      default:  errcnt++; /* Corrects FAULT13 */
        /* FAULT13 remove the default statement above */
      }
  if (errcnt)
    {
      fprintf(stderr, "Usage: %s [-ai] [-c chars] [-m count]\n", *argv);
      return(1);
    }
  else
  for (c = 'a'; c <= 'z'; c++)
    MapAllowed[c] = c;
  for (c = 'A'; c <= 'Z'; c++)
    MapAllowed[c] = Ignore ? c - 'A' + 'a' : c; /* Corrects FAULT14 */
  /* MapAllowed[c] = Ignore ? c - 'A' : c; / * FAULT14*/
  if (!Alpha)
    {
      /* for (c = '0'; c <= '9'; c++) / * Corrects FAULT15 and FAULT16 */
      /* for (c = 'O'; c <= '9'; c++) / * FAULT15 */
      for (c = '0'; c < '9'; c++) /* FAULT16 */
        MapAllowed[c] = c;
    }

  tokens(stdin);
  return(0);
}
```

B.2 count

```
#include <stdio.h>

main (argc, argv)
     int argc;
     char *argv[];
{
  int   c, i, inword;
  FILE  *fp;
  long  linect, wordct, charct;
  long  tlinect = 0, twordct = 0, tcharct = 0;

  i = 1;
  do {
    if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL)
      {
        fprintf (stderr, "can't open %s\n", argv[i]);
        /*      int j;
        exit (1);
      }
    else
    linect = wordct = charct = 0; /* This corrects FAULT1 */
    /* linect = wordct = charct = 1; / * FAULT1 */
    inword = 0;
    while ((c = getc(fp)) != EOF) {
      ++charct;
      if (c == '\n')
        {
          ++linect;
        }
      else
      if (c == ' ' || c == '\t' || c == '\n')
/* The above line corrects FAULT2, FAULT3 and FAULT4 */
      /* if (c == '\t' || c == '\n') / * FAULT2 */
      /* if (c == ' ' || c == '\n') / * FAULT3 */
      /* if (c == ' ' || c == '\t') / * FAULT4 */
        {
          inword = 0;
        }
      else
        {
          if (inword == 0) { /* This corrects FAULT5 */
            /* if (inword = 0) { /* FAULT5 */
```

```
            inword = 1;
            ++wordct;
          }
          else
        }
    }
    printf("%7ld %7ld %7ld", linect, wordct, charct);
    if (argc > 1)
      {
        printf(" %s\n", argv[i]);
      }
    else
      {
        printf("\n");
      }
    fclose(fp);
    tlinect += linect;
    twordct += wordct;
    tcharct += charct;
  } while (++i < argc);
  if (argc > 2)
    {
      printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
/* The above line corrects FAULT6 and FAULT7*/
      /* printf("%7ld %7ld %7ld total\n", twordct, twordct, tcharct); / * FAULT6 */
      /* printf("%7ld %7ld %7ld total\n", tlinect, twordct, twordct); / * FAULT7 */
    }
  exit(0);
}
```

B.3 series

```
/*  Copyright 1982 Gary Perlman */
#include <stdio.h>
#include "number.c"

#define startstr  argv[1]    /* Corrects FAULT1 */
/* #define      startstr  argv[2]    / * FAULT1 */
#define endingstr argv[2]   /* Corrects FAULT2 */
/* #define      endingstr argv[3]   / * FAULT2 */
#define stepstr   argv[3]  /* Corrects FAULT3 */
/* #define      stepstr   argv[1]  / * FAULT3 */

#define GFORMAT "%g\n"
#define IFORMAT "%.0f\n"

char    *Format = GFORMAT;
int     Onlyint;
double  Start;
double  Ending;
#define RANGE (Ending-Start)
#define FZERO   10e-10
#define fzero(x) (fabs (x) < FZERO) /* Corrects FAULT4 */
/* #define fzero(x) (fabs (x) <= FZERO) / * FAULT4 */
double  Step = 1.0;

extern double fabs();
extern double atof();

int main (argc, argv)
     int argc;
     char **argv;
{
  long  nitems;
  long  item;
  double        value;
  int   nargs = argc - 1;

  switch (nargs)
    {
    case 3:
      if (! number(stepstr))
        {
          printf("Argument #3 isn't a number: %s\n", stepstr);
```

```
/* The above line corrects FAULT5 and FAULT6 */
          /* printf("Argument #3 isn't a number: %s\n", startstr); / * FAULT5 */
          /* printf("Argument #3 isn't a number: %s\n", endingstr); / * FAULT6 */
          exit(1);
        }
    case 2:
      if (! number(startstr))
        {
          /* printf("Argument #1 isn't a number: %s\n", stepstr); / * FAULT7 */
          printf("Argument #1 isn't a number: %s\n", startstr);
/* The above line corrects FAULT 7*/
          exit(1);
        }
      else
      if (! number(endingstr))
        {
          printf("Argument #2 isn't a number: %s\n", endingstr);
/* The above line corrects FAULT8 and FAULT9 */
          /* printf("Argument #2 isn't a number: %s\n", endingstr); / * FAULT8 */
          /* printf("Argument #2 isn't a number: %s\n", endingstr); / * FAULT9 */
          exit(1);
        }
        else
      break;
    default:
      printf("USAGE start end stepsize\n");
      exit(1);
    }

  Start   = atof(startstr);
  Ending  = atof(endingstr);
  Onlyint = isinteger(startstr) && isinteger(endingstr);
/* The above line corrects FAULT10 and FAULT11 and FAULT12 */
  /* Onlyint = isinteger(endingstr) && isinteger(endingstr); / * FAULT10 */
  /* Onlyint = isinteger(startstr) || isinteger(endingstr); / * FAULT11 */
  /* Onlyint = isinteger(startstr) && isinteger(startstr); / * FAULT12 */
  if (nargs == 3)
    {
      Step     = fabs(atof(stepstr));
      if (! fzero(RANGE) && fzero(Step)) /* This corrects FAULT13 */
      /* if (! fzero(RANGE)) / * FAULT13 */
        {
          printf("stepsize must be non-zero\n");
```

```
        exit(0);
      }
    else
    Onlyint &= isinteger(stepstr); /* This corrects FAULT14 */
    /* FAULT14 - Remove the assingment statement above */
  }
else

if (Onlyint)
  {
    Format = IFORMAT;
  }
else
if (fzero(RANGE))
  {
    nitems = 1;
  }
else
  {
    nitems = fabs(RANGE) / Step + 1.0 + FZERO; /* This corrects FAULT15 */
    /* nitems = fabs(RANGE) / Step + FZERO; / * FAULT15 */
  }

/* for (item = 1; item < nitems; item++) / * FAULT16 */
for (item = 0; item < nitems; item++) /* This corrects FAULT16 */
  {
    if (RANGE > 0)
      {
        /* value = Start + Step * item; / * This corrects FAULT17 */
        value = Start - Step * item; /* FAULT17 */
      }
    else
      {
        value = Start - Step * item;
      }
    if (fzero(value))
      {
        printf(Format, 0.0);
      }
    else
      {
        printf(Format, value);
      }
```

```
    }
}
```

B.4 ntree

```
#include <stdio.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
#include "ntree.h"

static TN *init_tree_node(key, data, parent)
     char *key;
     char *data;
     struct tree_node *parent;
{
  struct tree_node *node;

  assert(key != NULL); /* This corrects FAULT1 */
        /* FAULT1 - remove the above assert-statement */
  assert(data != NULL); /* This corrects FAULT2 */
        /* FAULT2 - remove the above assert-statement */
  node = (TN *)malloc(sizeof(TN));
  assert(node != NULL);
  node->key = key; /* This corrects FAULT3 */
        /* FAULT3 - remove the above assignment statement */
  node->parent = parent;               /* This corrects FAULT4 */
        /* FAULT4 - remove the above assignment statement */
  node->nchildren = 0; /* This corrects FAULT5 */
  /*  node->nchildren = 1; / * FAULT5 */
  node->maxchildren = INITIAL_CAPACITY;
  node->children = (TN **)malloc(sizeof(TN *) * INITIAL_CAPACITY);
  assert(node->children != NULL);
  return node;
}

TREE *t_root(key, data)
     char *key;
     char *data;
{
  TREE *tree;
  assert(key != NULL);
  assert(data != NULL);
  tree = (TREE *)malloc(sizeof(TREE));
  assert(tree != NULL);
  tree->root = init_tree_node(key, data, NULL); /* This corrects FAULT6 */
  /* tree->root = init_tree_node(data, key, NULL); / * FAULT6 */
```

```
  assert(tree->root != NULL);

  return tree;
}

static TN *find_node(tn, key)
     TN *tn;
     char *key;
{
  int i, rc;
  struct tree_node *found;

  assert(tn != NULL);
  assert(key != NULL);
  found = NULL; /* This corrects FAULT7 */
  /* found = tn; / * FAULT7 */
  rc = strcmp(key, tn->key);
  if (rc == 0) {
    found = tn;
  }
  else {
    for (i = 0; i < tn->nchildren; ++i) {
      found = find_node(tn->children[i], key); /* This corrects FAULT9 */
      /* found = find_node(tn->children[i], key); / * FAULT9 */
      if (found) {
        break; /* This corrects FAULT8 */
/* FAULT8 - remove the break statement above */
      }
    }
  }
  return found;
}

int t_add_child(t, parent_key, child_key, child_data)
     TREE *t;
     char *parent_key, *child_key;
     char *child_data;
{
  struct tree_node *parent, *child;
  struct tree_node **space;
  int rc;

  assert(t != NULL);
```

```
  assert(parent_key != NULL);
  assert(child_key != NULL);
  parent = find_node(t->root, parent_key); /* This corrects FAULT11 */
  /* parent = find_node(t->root, child_key); / * FAULT11 */
  if (parent == NULL) { /* This corrects FAULT10 */
/*  if (child == NULL) { / * FAULT10 */
    printf("Vaterknoten mit Schluessel %s nicht gefunden\n", parent_key);
    rc = -1; /* This corrects FAULT12 */
    /*  rc = 1; / * FAULT12 */
  }
  else {
    if (parent->nchildren == parent->maxchildren) { /* This corrects FAULT13 */
    /* if (parent->nchildren > parent->maxchildren) { / * FAULT13 */
      bcopy(parent->children, space, sizeof(TN*) * parent->maxchildren);
      free(parent->children);
      parent->children = space;
      (parent->maxchildren) *= 2; /* This corrects FAULT14 */
      /* FAULT14 - Remove the multiplication statement above */
    }
    child = init_tree_node(child_key, child_data, parent);
    assert(child != NULL);
    parent->children[parent->nchildren] = child; /* This corrects FAULT15 */
    /*    parent->children[child->nchildren] = child; / * FAULT15 */
    ++(parent->nchildren); /* This corrects FAULT16 */
    /*  FAULT16 - remove the increment statement above */
    rc = 0; /* This corrects FAULT17 */
/* FAULT17 - remove the assignment statement above */
  }
  return rc;
}

int t_search(t, key)
     TREE *t;
     char *key;
{
  int rc;
  struct tree_node *node;

  rc = -1;
  assert(t != NULL);
  assert(key != NULL);
  node = find_node(t->root, key);
  if (node != NULL) {
```

```
    printf("Der Inhalt ist %s\n", node->data);
    rc = 0; /* This corrects FAULT18 */
/* FAULT18 - remove the assignment statement above */
  }
  if (rc == -1)
    {
      printf("Knoten mit Schluessel %s nicht gefunden\n", key);
    }
  return rc;
}

int t_are_siblings(t, key1, key2)
     TREE *t;
     char *key1, *key2;
{
  struct tree_node *node1, *node2;
  int rc;

  assert(t != NULL);
  assert(key1 != NULL);
  assert(key2 != NULL);
  rc = 0;
  node1 = find_node(t->root, key1); /* This corrects FAULT19 */
  /*  node1 = find_node(t->root, key2); / * FAULT19 */
  if (node1 == NULL) {  /* This corrects FAULT20 */
    /* if (node1 = NULL) {  / * FAULT20 */
    printf("Knoten mit Schluessel %s nicht gefunden\n", key1);
    rc = -1;
  }
  else {
    node2 = find_node(t->root, key2); /* This corrects FAULT22 */
    /* node2 = find_node(t->root, key1); / * FAULT22 */
    if (node2 == NULL) { /* This corrects FAULT21 */
/*    if (node2 = NULL) { / * FAULT21 */
      printf("Knoten mit Schluessel %s nicht gefunden\n", key2);
      rc = -1;
    }
    else {
      printf("Die Knoten %s und %s sind %sGeschwister.\n", key1, key2,
             (node1->parent == node2->parent ? "" : "keine "));
/* The above line corrects FAULT23 and FAULT24 */
/* printf("Die Knoten %s und %s sind %sGeschwister.\n", key1, key2,
         (node2->parent == node2->parent ? "" : "keine ")); / * FAULT23 */
```

```
/*      printf("Die Knoten %s und %s sind %sGeschwister.\n", key1, key2,
        (node1->parent == node1->parent ? "" : "keine ")); / * FAULT24 */
    }
  }
  return rc;
}


static void print_tree_nodes(tn, level)
     TN *tn;
     int level;
{
  int i;

  assert(tn != (struct tree_node *)0);
  for (i=0; i < level; ++i) /* This corrects FAULT25 */
  /* for (i=0; i <= level; ++i) / * FAULT25 */
    printf("    ");
  printf("Knoten (Ebene %d): Schluessel '%s', Inhalt '%s'\n",
         level, tn->key, tn->data);
  for (i = 0; i < tn->nchildren; ++i) { /* This corrects FAULT26 */
    /*  for (i = 0; i <= tn->nchildren; ++i) { / * FAULT26 */
    print_tree_nodes(tn->children[i], level + 1);
/* The above line corrects FAULT27 and FAULT28 */
    /* print_tree_nodes(tn->children[i], level - 1); / * FAULT27 */
    /* print_tree_nodes(tn->children[i], level); / * FAULT28 */
  }
}

int t_print(t)
     TREE *t;
{
  int rc = -1; /* This corrects FAULT29 */
  /*  int rc = 0; / * FAULT29 */

  assert(t != NULL);
  if (t->root != NULL) {
    print_tree_nodes(t->root, 0);
    /*    rc = 0; / * This corrects FAULT30 */
/* FAULT30 - Remove the assignment statement above */
  }
  return rc;
}
```

```
/*
 * Hier beginnt die Testumgebung.
 * Bitte die Testumgebung nicht testen, keine Abstraktionen bilden etc.
 */
void fuehre_kommandos_aus(filep)
     FILE *filep;
{
  char *ptr;
  char buf[BUFSIZ];
  char kommando[BUFSIZ], arg1[BUFSIZ], arg2[BUFSIZ], arg3[BUFSIZ];
  TREE *mytree = NULL;

  while (fgets(buf, sizeof(buf), filep) != NULL) {
    if ( (ptr = strchr(buf, '\n')) != NULL)
      *ptr = '\0';
    printf("\nDie Zeile `%s' wird ausgewertet:\n", buf);
    *kommando = '\0';
    *arg1 = '\0';
    *arg2 = '\0';
    *arg3 = '\0';
    sscanf(buf, "%s %s %s %s", kommando, arg1, arg2, arg3);
    if (strcmp(kommando, "root") == 0)
        mytree = t_root(strdup(arg1), strdup(arg2));
    else if (strcmp(kommando, "child") == 0)
         t_add_child(mytree, arg1, strdup(arg2), strdup(arg3));
         else if (strcmp(kommando, "search") == 0)
              t_search(mytree, arg1);
              else if (strcmp(kommando, "sibs") == 0)
                   t_are_siblings(mytree, arg1, arg2);
                   else if (strcmp(kommando, "print") == 0)
                        t_print(mytree);
                        else
                            printf("Kommando `%s' nicht erkannt\n", kommando);
  }
}

int main(argc, argv)
     int argc; char **argv;
{
  FILE *filep;
  char *file;
```

```
  if (argc != 2)
    fprintf(stderr, "Verwendung: %s Datei\n", *argv);
  else {
    file = argv[1];
    if ((filep = fopen(file, "r")) == NULL)
      perror(file);
    else {
      printf("Eingabedatei ‘%s’ wird bearbeitet.\n", file);
      fuehre_kommandos_aus(filep);
      printf("Ende der Eingabedatei ‘%s’.\n", file);
      fclose(filep);
    }
  }
  return 0;
}
\newpage
```

B.5 nametbl

```
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <search.h>
/* extern char *tsearch(), *tfind(), *tdelete(), twalk(); */
#include "nametbl.h"

NT *newtable(/*void*/)
{
  NT *ptr;

  ptr = (NT *)malloc(sizeof(NT));
  assert(ptr != NULL);
  ptr->rootp = NULL;
  ptr->numitems = 0; /* Thic corrects FAULT1 and FAULT2*/
  /*  ptr->numitems = 1; / * FAULT1 */
/* FAULT2 - Remove the assignment statement above */
  return ptr;
}

int how_many(nt)
     NT *nt;
{
  return nt->numitems;
}

int compare_entry(p1, p2)
     NTE *p1, *p2;
{
  return(strcmp(p1->name, p2->name)); /* This corrects FAULT3 and FAULT4 */
  /* return(strcmp(p2->name, p2->name)); / * FAULT3 */
  /* return(strcmp(p1->name, p1->name)); / * FAULT4 */
}

void print_entry(node)
     NTE *node;
{
  printf("Name    : %s\n", node->name);
  printf("oType   : %s\n", objectTypeName[node->ot]);
  printf("rType   : %s\n", resourceTypeName[node->rt]);
  printf("-----\n");
}
```

```
void print_on_last_visit(nodep, order, level)
     NTE **nodep;
     VISIT order;
     int level;
{
   if (order == postorder || order == leaf) /* This corrects FAULT5 - FAULT15 */
    /*  if (order != postorder || order == leaf) / * FAULT5 */
  /*  if (order > postorder || order == leaf) / * FAULT6 */
  /* if (order < postorder || order == leaf) / * FAULT7 */
  /*  if (order >= postorder || order == leaf) / * FAULT8 */
  /*  if (order <= postorder || order == leaf) / * FAULT9 */
  /*  if (order == postorder && order == leaf) / * FAULT10 */
  /*  if (order == postorder || order != leaf) / * FAULT11 */
  /*  if (order == postorder || order > leaf) / * FAULT12 */
  /* if (order == postorder || order < leaf) / * FAULT13 */
  /*  if (order == postorder || order >= leaf) / * FAULT14 */
  /* if (order == postorder || order <= leaf) / * FAULT15 */
    {
      (void) print_entry(*nodep);
    }
}

void insert_entry(nt, new_name, new_ot, new_rt)
     NT                   *nt;
     char                 *new_name;
     enum objectType       new_ot;
     enum resourceType     new_rt;
{
  NTE *nte, **result;

  nte = (NTE *) malloc(sizeof(NTE));
  assert(nte != NULL);
  nt->numitems++;                /* This corrects FAULT16 */
  /* nt->numitems = nt->numitems + 1;                / * FAULT16 */
  nte->name = strdup(new_name);
  assert(nte->name != NULL);
     nte->ot = new_ot; /* This corrects FAULT17 */
  /*  nte->ot = new_ot; / * FAULT17 */
  nte->rt = new_rt; /* This corrects FAULT18 */
     /*  nte->rt = new_rt; / * FAULT18 */

  result = (NTE **) tsearch((char *)nte, &(nt->rootp), compare_entry);
```

```
  assert((char *) *result == (char *) nte);
}

NTE *retrieve_entry(nt, searchname)
     NT *nt;
     char *searchname;
{
  NTE **nte, *retval, key;

  key.name = searchname;
  nte = (NTE **) tfind( (char *)&key, &(nt->rootp), compare_entry);
  if (nte != NULL)
    {
      retval = *nte;
    }
  else
    {
      retval = NULL;
    }
  return retval; /* This corrects FAULT19 */
  /*  return *nte; / * FAULT19 */
}

int setObjType (nt, searchname, new_ot)
     NT *nt;
     char *searchname;
     enum objectType new_ot;
{
  int retval;
  NTE *nte;

    retval = -1; /* This corrects FAULT20 */
  /* FAULT20 - remove the assignement statement above */
  nte = retrieve_entry(nt, searchname);
  if (nte != NULL)
    {
      retval = 0;  /* This corrects FAULT21 */
      /* retval = 1;  / * FAULT21 */
       nte->ot = new_ot;   /* This corrects FAULT22 and FAULT23 */
      /* nte->rt = new_ot;   / * FAULT22 */
    }
  else
    {
```

```
      /*      nte->ot = new_ot; / * FAULT23 */
    }
  return retval;
}

int setResType (nt, searchname, new_rt)
     NT *nt;
     char *searchname;
     enum resourceType new_rt;
{
  int retval;
  NTE *nte;

  retval = -1; /* This corrects FAULT24 */
/* FAULT24 - Remove the assignment statement above */
  nte = retrieve_entry(nt, searchname);
  if (nte != NULL)
  {
      retval = 0; /* This corrects FAULT25 */
      /*      retval = 1; / * FAULT25 */
      nte->rt = new_rt; /* This corrects FAULT26 */
      /* nte->ot = new_rt; / * FAULT26 */
    }
  else
  return retval;
}

int ins(nt, new_name)
     NT *nt;
     char *new_name;
{
  int retval;
  NTE *nte;

  nte = retrieve_entry(nt, new_name);
  if (nte != NULL) {
    printf("Der Name ‘%s’ ist schon in der Tabelle.\n", new_name);
        retval = -1; /* This corrects FAULT27 */
/* FAULT27 - Remove the assignment statement above */
  }
  else {
    insert_entry(nt, new_name, OT_NO_INF, RT_NO_INF); /*  This corrects FAULT29*/
    /*  insert_entry(nt, new_name, RT_NO_INF, OT_NO_INF); / * FAULT29 */
```

```
    retval = 0; /* This corrects FAULT28 */
/* FAULT28 - remove the assignment statement above */
  }

  return retval;
}

int tot(nt, new_name, new_ot_char)
     NT *nt;
     char *new_name;
     char *new_ot_char;
{
  int rc;
  enum objectType new_ot;

    new_ot = -1; /* This corrects FAULT30 */
/* FAULT30 - remove the assignment statement above */
    if (strcmp("SYSTEM", new_ot_char) == 0) /* This corrects FAULT31 */
      /*  if (strcmp("SYSTEM", new_ot_char) != 0) / * FAULT31 */
    {
      new_ot = SYSTEM; /* This corrects FAULT32 */
      /* new_ot = RESOURCE; / * FAULT32 */
    }
  else {
    if (strcmp("RESOURCE", new_ot_char) == 0)
      {
        new_ot = RESOURCE; /* This corrects FAULT33 */
        /* new_ot = SYSTEM; / * FAULT33 */
      }
    else
  };
    if (new_ot == -1) { /* This corrects FAULT34 */
      /* if (new_ot = -1) { / * FAULT34 */
    printf("Typ `%s' nicht erkannt\n", new_ot_char);
    rc = -1; /* This corrects FAULT35 */
/*  FAULT35 remove the assignment statement above */
  }
  else {
    rc = setObjType(nt, new_name, new_ot); /* This corrects FAULT36 */
    /*  rc = setObjType(nt, new_ot, new_name); / * FAULT36 */
    if (rc)
      {
        printf("Der Name `%s' ist nicht in der Tabelle.\n", new_name);
```

```
      }
   }

  return rc;
}

int trt(nt, new_name, new_rt_char)
     NT *nt;
     char *new_name;
     char *new_rt_char;
{
  int rc;
  enum resourceType new_rt;

  new_rt = -1; /* This corrects FAULT37 */
/* FAULT37 - remove the assignment statement above */
   if (strcmp("RT_SYSTEM", new_rt_char) == 0) /* This corrects FAULT38 */
     /*  if (strcmp("RT_SYSTEM", new_rt_char)) / * FAULT38 */
    {
      new_rt = RT_SYSTEM; /* This corrects FAULT39 and FAULT40 */
      /*      new_rt = FUNCTION; / * FAULT39 */
      /*      new_rt = DATA; / * FAULT40 */
    }
  else
    {
      if (strcmp("FUNCTION", new_rt_char) == 0) /* This corrects FAULT41 */
      /* if (strcmp("FUNCTION", new_rt_char)) / * FAULT41 */
        {
          new_rt = FUNCTION; /* This corrects FAULT42 and FAULT43 */
          /*      new_rt = RT_SYSTEM; / * FAULT42 */
          /* new_rt = DATA; / * FAULT43 */
        }
      else
        {
          if (strcmp("DATA", new_rt_char) == 0)
            {
              new_rt = DATA; /* This corrects FAULT44 and FAULT45 */
              /*              new_rt = RT_SYSTEM; / * FAULT44 */
              /*              new_rt = FUNCTION; / * FAULT45 */
            }
        }
    }
  if (new_rt == -1) {
```

```
    printf("Typ ‘%s’ nicht erkannt\n", new_rt_char);
    rc = -1; /* This corrects FAULT46 */
/* FAULT46 - Remove the assignment statement above */
  }
  else {
    rc = setResType(nt, new_name, new_rt); /* This corrects FAULT47 */
    /*    rc = setResType(nt, new_rt, new_name); / * FAULT47 */
    if (rc)
      {
        printf("Der Name ‘%s’ ist nicht in der Tabelle.\n", new_name);
      }
  }

  return rc;
}

void sch(nt, name)
     NT *nt;
     char *name;
{
  NTE *nte;

  printf("Suche nach Name ‘%s’:\n", name);
  nte = retrieve_entry(nt, name);
  /*  if (nte != NULL) / * This corrects FAULT48 */
  if (nte == NULL) /* FAULT48 */
    {
      print_entry(nte);
    }
  else
    {
      printf("Der Name ‘%s’ ist nicht in der Tabelle.\n", name);
    }
}

void prt(nt)
     NT *nt;
{
  printf("Die Tabelle hat die folgenden %d Eintraege:\n", how_many(nt));
  twalk((char *)nt->rootp, print_on_last_visit);
}

/*
```

```
 * Hier beginnt die Testumgebung.
 * Bitte die Testumgebung nicht testen, keine Abstraktionen bilden etc.
 */
void fuehre_kommandos_aus(filep)
     FILE *filep;
{
  NT *nt;
  char *ptr;
  char buf[BUFSIZ], kommando[BUFSIZ], name[BUFSIZ], typ[BUFSIZ];

  nt = newtable();
  assert (nt != NULL);

  while (fgets(buf, sizeof(buf), filep) != NULL) {
    if ( (ptr = strchr(buf, '\n')) != NULL)
      *ptr = '\0';
    printf("\nDie Zeile `%s' wird ausgewertet:\n", buf);
    *kommando = '\0';
    *name = '\0';
    *typ = '\0';
    sscanf(buf, "%s %s %s", kommando, name, typ);
    if (strcmp(kommando, "ins") == 0)
      ins(nt, name);
    else if (strcmp(kommando, "tot") == 0)
         tot(nt, name, typ);
         else if (strcmp(kommando, "trt") == 0)
              trt(nt, name, typ);
              else if (strcmp(kommando, "sch") == 0)
                   sch(nt, name);
                   else if (strcmp(kommando, "prt") == 0)
                        prt(nt);
                        else printf("Kommando `%s' nicht erkannt\n", kommando);
  }
}

int main(argc, argv)
     int argc; char **argv;
{
  NT *nt;
  FILE *filep;
  char *file;

  if (argc != 2)
```

```
    fprintf(stderr, "Verwendung: %s Datei\n", *argv);
  else {
    file = argv[1];
    if ((filep = fopen(file, "r")) == NULL)
      perror(file);
    else {
      printf("Eingabedatei ‘%s’ wird bearbeitet.\n", file);
      fuehre_kommandos_aus(filep, nt);
      printf("Ende der Eingabedatei ‘%s’.\n", file);
      fclose(filep);
    }
  }
  return 0;
}
```