Combination Testing Strategies: A Survey *

Mats Grindal
Computer Science
University of Skövde
SE-541 28 Skövde, Sweden
(+46) 8-50714230
(+46) 8-50714040 (fax)
magr@ida.his.se

Jeff Offutt
Info. and Software Engng
George Mason University
Fairfax, VA 22030, USA
(+1) 703-9934-1654
ofut@ise.gmu.edu

Sten F. Andler
Computer Science
University of Skövde
SE-541 28 Skövde, Sweden
(+46) 500-448313
sten@ida.his.se

July 2004

GMU Technical Report ISE-TR-04-05, July 2004

Abstract

Combination strategies are test-case selection methods where test cases are identified by combining values of the different test object input parameters based on some combinatorial strategy. This survey presents 15 different combination strategies, and covers more than 30 papers that focus on one or several combination strategies. We believe this collection represents most of the existing work performed on combination strategies. This survey describes the basic algorithms used by the combination strategies. Some properties of combination strategies, including coverage criteria and theoretical bounds on the size of test suites, are also included in this description.

This survey paper also includes a subsumption hierarchy that attempts to relate the various coverage criteria associated with the identified combination strategies. Finally, this survey contains short summaries of all the papers that cover combination strategies.

1 Introduction

Combination strategies define ways to select values for individual input parameters and combine them to form complete test cases. A literature search has revealed 15 different combination strategies described through more than 30 papers published between 1985 and 2003. This survey is an attempt to collect all this knowledge in one single place and form a basis for comparisons among the combination strategies.

*This research is funded in part by KK-stiftelsen and Enea Systems AB

Year	Authors	Ref.	Method
1985	Mandl	[1]	**OA**, Rand, AC
1988	Ostrand, Balcer	[2]	**CP**
1990	Toczki, Kocsis, Gyimóthy, Dányi, Kókai	[3]	CP
1992	Brownlie, Prowse, Phadke	[4]	OA
1993	Piwowarski, Ohba, Caruso	[5]	Rand
1994	Ammann, Offutt	[6]	**BC**, **EC**, AC
1994	Burroughs, Jain, Erickson	[7]	**AETG, PPW, AC, BC**
1995	Heller	[8]	DoE
1996	Cohen, Dalal, Parelius, Patton	[9]	AETG , Rand
1996	Williams, Probert	[10]	OA
1997	Cohen, Dalal Fredman, Patton	[11]	AETG, BC Rand, AC
1997	Dunietz, Ehrlich, Szablak, Mallows, Iannino	[12]	Rand, AC, OA
1997	Yin, Lebne-Dengel, Malayia	[13]	AntiRand

Table 1: A chronological overview of which combination strategies are mentioned in different papers. Bold indicates what we determined to be seminal work, where a combination strategy was introduced.

Table 1 lists the primary papers covered in this survey, in order of publication, and shows which combination strategies discussed in each paper. Combination strategies shown in bold face are thought to have originated in the associated paper. We were not able to determine where Rand, AC, and DoE were introduced.

Section 2 gives a brief background on testing in general and on the Category Partition method [2] in particular. The Category Partition method is a convenient place to start because it yields independent values for parameters and spawned much of the work in combination strategies. Section 3 start by explaining the coverage criteria usually associated with combination strategies then gives an overview of each combination strategy identified in this survey. To aid the reader we have structured the combination strategies into different groups based on different properties of the combination strategy algorithms. The overview of each algorithm includes an explanation of the algorithm. Whenever applicable, associated coverage criteria and theoretical bounds on the number of test cases are also included in the description. Section 4 compares the different combination strategies with respect to size of generated test suite. A new contribution of this paper is a subsumption hierarchy for the identified coverage criteria. It can be used to compare the combination strategies with respect to their associated coverage criteria.

Section 5 summarizes each combination strategy paper. Special attention has been given to comparative studies among combination strategies. This section also includes publications of experience and lessons learned from applying combination strategies. Finally, section 6 summarizes the main contributions of this paper.

Year	Authors	Ref.	Method
1998	Burr, Young	[14]	AETG, BC
1998	Kropp, Koopman, Siewiorek	[15]	**ACoRand**
1998	Dalal, Mallows	[16]	OA, AETG, AC, CATS
1998	Dalal, Jain, Karunanithi, Leaton, Lott	[17]	AETG
1998	Lei, Tai	[18]	**IPO**
1999	Dalal, Jain, Patton, Rathi, Seymour	[19]	AETG
1999	Dalal, Jain, Karunanithi, Leaton, Lott, Patton, Horowitz	[20]	AETG
1999	Hoffman, Strooper, White	[21]	k-bound, k-perim
2000	Williams	[22]	**CA**, OA, IPO
2000	Huller	[23]	IPO
2001	Lei, Tai	[24]	IPO, AETG
2001	Williams, Probert	[25]	-
2002	Williams, Probert	[26]	-

Table 1: (continued) A chronological overview of which combination strategies are mentioned in different papers. Bold indicates what we determined to be seminal work, where a combination strategy was introduced.

Year	Authors	Ref.	Method
2002	Tai, Lei	[27]	IPO, AETG
2002	Daley, Hoffman, Strooper	[28]	k-bound, k-perim
2002	Kuhn, Reilly	[29]	-
2003	Berling, Runesson	[30]	DoE
2003	Grindal, Lindstrom, Offutt, Andler	[31]	**BCAETG**, EC, BC, AC, AETG, OA

Table 1: (continued) A chronological overview of which combination strategies are mentioned in different papers. Bold indicates what we determined to be seminal work, where a combination strategy was introduced.

2 Background

Testing, loosely considered to be the dynamic execution of test cases, consumes a significant amount of the resources in development projects [32, 33]. Thus, it is of great importance to investigate ways of increasing the efficiency and effectiveness in testing [34, 35, 36, 37, 38].

Several existing test methods (e.g. Equivalence Partitioning [32], Category Partition [2], and Domain Testing [33]) are based on the model that the input space of the test object may be divided into subsets based on the assumption that all points in the same subset result in a similar behavior from the test object. This is called *partition testing*. Even if the partition test assumption is an idealization it has two important properties. First, it lets the tester identify test suites of manageable size by selecting one or a few test cases from each subset. Second, it allows test effectiveness to be measured in terms of coverage with respect to the partition model used.

One alternative to partition testing is *random testing*, in which test cases are chosen randomly from some input distribution (such as a uniform distribution) without exploiting information from the specification or previously chosen test cases. Duran and Ntafos [39] gave results that under certain very restrictive conditions, random testing can be as effective as partition testing. They showed consistent small differences in effectiveness between partition testing methods and random testing. These results were interpreted in favor of random testing since it is generally less work to construct test cases in random testing since partitions do not have to be constructed.

Hamlet and Taylor [40] later investigated the same question and concluded that the Duran/Ntafos model was unrealistic. One main reason was that the overall failure probability was too high. Hamlet and Taylor theoretically strengthened the case for partition testing but made an important point that partition testing can be no better that the information that defines the subdomains. Gutjahr [41] followed up on Hamlet and Taylor's results and showed theoretically that partition testing is consistently more effective than random testing under realistic assumptions.

More recent results have been produced that favor partition testing over random testing in practical cases. Reid [36] and Yin, Lebne-Dengel, and Malayia [13] both performed experiments with different partition testing strategies and compared them with random testing. In all cases random testing was found to be less effective than the investigated partition testing methods.

A key issue in any partition testing approach is how partitions should be identified and how values should be selected from them. In early partition test methods like Equivalence Partitioning (EP) [32] and Boundary Value Analysis (BVA) [32], parameters of the test problem are identified. Each parameter is then analyzed in isolation to determine suitable partitions of that parameter. Support for identifying parameters and their partitions from arbitrary specifications is rather limited in EP and BVA. Ostrand and Balcer proposed the Category Partition (CP) method partially to address this problem [2].

2.1 Category Partition

The category partition method [2] consists of a number of manual steps by which a natural language specification is systematically transformed into an equivalence class model for the parameters of the test object. Figure 1 shows the steps of the category partition method.

Constraints allow the tester to decrease the size of the test suite. Even with constraints, the number of test frames generated by all combinations can be combinatorial ($\prod_{i=1}^{N} v_i$, where v_i is the number of values

1. Identify functional units that may be tested separately.
   For each functional unit, identify parameters and environment variables
   that affect the behavior of the functional unit.
2. Identify choices for each parameter and environment individually.
3. Determine constraints among the choices.
4. Generate *all combinations*, so called test frames, of parameter choices
   that satisfy the constraints.
5. Transform the test frames into test cases by instantiating the choices.

Figure 1: Overview of the steps of the category partition method

for each one of the N parameters). Combination strategies can be used to decrease the number of test cases in the test suite.

3 Combination Strategies - Methods

Combination strategies is a class of test-case selection methods where test cases are identified by choosing "interesting" values[1], and then combining those values of test object parameters. The values are selected based on some combinatorial strategy. Some combination strategies are based on techniques from experimental design.

This section first explains the different coverage criteria, normally associated with combination strategies and then briefly describes the combination strategies that were identified in the literature. The combination strategies have been organized into different classes based on the amount of randomness of the algorithm and according to how the test suites are created. Figure 2 shows an overview of the classification scheme. The combination strategies labeled *non-deterministic* all depend to some degree on randomness. A property of these combination strategies is that the same input parameter model may lead to different test suites. The simplest non-deterministic combination strategy is pure *random* selection of test cases. The group of non-deterministic combination strategies also include two heuristic methods, *CATS* and *AETG*.

The *deterministic* combination strategies group is further divided into three subgroups, *instant*, *iterative*, and *parameter-based*. All of these combination strategies will always produce the same result from a specific input parameter model. The two instant combination strategies, *Orthogonal Arrays (OA)* and *Covering Arrays (CA)*, produce the complete test suite directly. The largest group of combination strategies is *iterative*. They share the property that the algorithms generate one test case at a time and adds it to the test suite. *Each Choice (EC)*, *Partly Pair-Wise (PPW)*, *Base Choice (BC)*, *All Combinations (AC)*, and *Anti-random (AR)* all belong to the iterative combination strategies. The parameter-based combination strategy, *In Parameter Order (IPO)*, starts by creating a test suite for a subset of the parameters in the input parameter model. Then one parameter at a time is added and the test cases in the test suite are modified to cover the new parameter. Completely new test cases may also need to be added.

[1]The term "interesting" may seem insufficiently precise and also a little judgmental. However, it appears frequently in the literature, so is also used in this paper. There are many ways to decide which values are "interesting" and some of the combination strategies discuss criteria for making those choices. In this paper, "interesting values" are whatever values the tester decides to use.

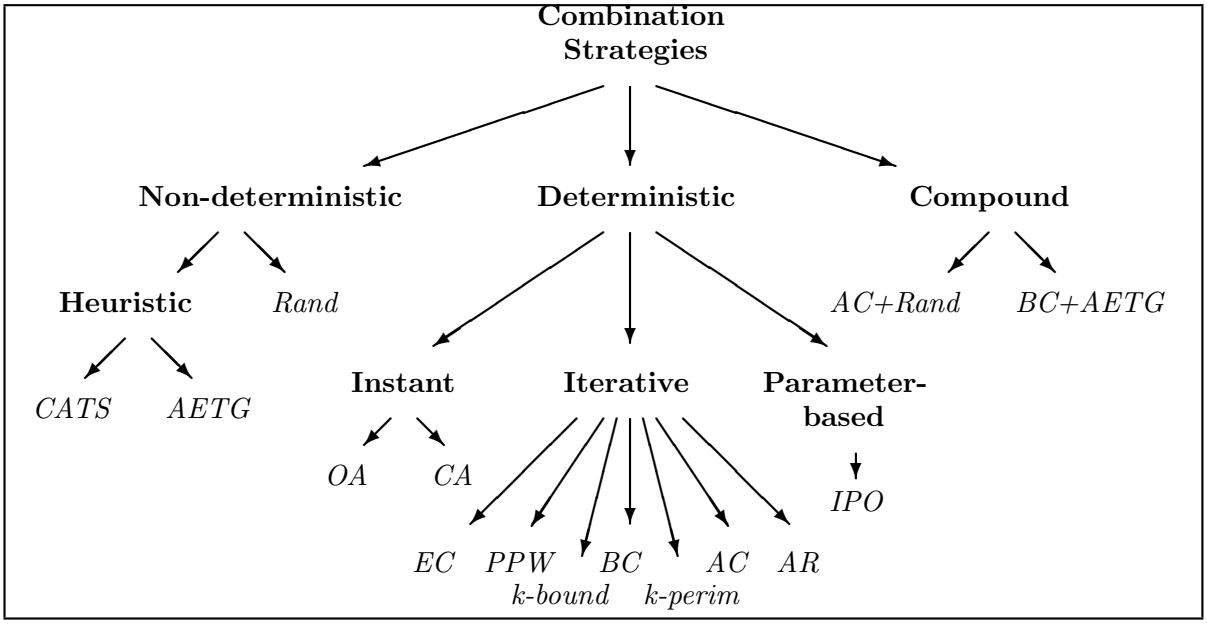

Figure 2: Classification Scheme for Combination Strategies

The third main group of combination strategies, *compound*, include all strategies in which two or more combination strategies are used together. Two such strategies from the literature are used to illustrate the concept.

3.1 Coverage Criteria for Combination Strategies

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 values of the parameters of the test object that the tester decides are interesting. The simplest coverage criterion, i.e., each-used coverage, does not take into account how interesting values of different parameters are combined, while the more complex coverage criteria, such as pair-wise coverage, is concerned with (sub-)combinations of interesting values of different parameters. The following subsections define the coverage criteria satisfied by combination strategies included in this paper.

Each-used (also known as 1-wise) coverage is the simplest coverage criterion. 100% each-used coverage requires that every interesting value of every parameter is included in at least one test case in the test suite.

100% *Pair-wise* (also known as 2-wise) coverage requires that every possible pair of interesting values of any two parameters are included in some test case. Note that the same test case may cover more than one unique pair of values.

A natural extension of pair-wise (2-wise) coverage is *t-wise* coverage, which requires every possible combination of interesting values of t parameters be included in some test case in the test suite. t-wise coverage is formally defined by Williams and Probert [25].

A special case of t-wise coverage is *N-wise* coverage, where N is the number of parameters of the test object. *N-wise coverage* requires all possible combinations of all interesting values of the N parameters be included in the test suite.

The each-used, pair-wise, t-wise, and N-wise coverage criteria are purely combinatorial and do not use any semantic information. More coverage criteria can be defined by using semantic information. Cohen et al. [11] indicate that valid and error parameter values should be treated differently with respect to coverage. *Normal* values lie within the bounds of normal operation of the test object, and *error* values lie outside of the normal operating range. Often, an error value will result in some kind of error message and the termination of the execution. To avoid one error value masking another Cohen et al. suggest that only one error value of any parameter should be included in each test case. This observation was also made and explained in an experiment by Grindal et al. [31].

By considering only the valid values, a family of coverage criteria corresponding to the general t-wise coverage criteria can be obtained. For instance, 100% *each valid used* coverage requires every valid value of every parameter be included in at least one test case in which the rest of the values also being valid. Correspondingly, 100% *t-wise valid* coverage requires every possible combination of valid values of t parameters be included in some test case, and the rest of the values are valid.

Error values may also be considered when defining coverage criteria. A test suite satisfies *single error* coverage if each error value of every parameter is included in some test case in which the rest of the values being valid.

A special case of normal values was used by Ammann and Offutt [6] to define base choice coverage. First, a base test case is identified by choosing the most frequently used value of each parameter. We assume here that the most frequently used value of each parameter is a normal value. 100% *base choice* coverage requires every interesting value of each parameter be included in a test case in which the rest of the values being base values. Further, the test suite must also contain the base test case.

3.2 Non-Deterministic Combination Strategies

Non-deterministic combination strategies all share the property that chance will play a certain role in determining which tests are generated. This means two executions of the same algorithm with the same preconditions may produce different results.

3.2.1 Heuristic Combination Strategies

Heuristic t-wise (CATS)
The CATS tool for generating test cases is based on a heuristic algorithm that can be custom designed to satisfy t-wise coverage. The algorithm was described by Sherwood [42] and a version of the algorithm that generates a test suite to satisfy pair-wise coverage is shown in figure 3.

The heuristic nature of the algorithm makes it impossible to exactly calculate the number of test cases in a test suite generated by the algorithm.

Heuristic t-wise (AETG)
Burroughs, Jain, and Erickson [7] and Cohen, Dalal, Kajla, and Patton [43] report on the use of a tool called Automatic Efficient Test Generator (AETG), which contains an algorithm for generating all pair-wise combinations. Cohen et al. [9, 11] later described a heuristic greedy algorithm for achieving t-wise coverage, where t is an arbitrary number. This algorithm was implemented in the AETG tool. Figure 4 shows an instance of the algorithm that will generate a test suite to satisfy pair-wise coverage.

```
Assume test cases t_1 − t_{i−1} already selected
Let Q be the set of all possible combinations not yet selected
Let UC be a set of all pairs of values of any two parameters that are
not yet covered by the test cases t_1 − t_{i−1}

A) Select t_i by finding the combination that covers most pairs in UC.
   If more than one combinations covers the same amount select the
     first one encountered.
   Remove the selected combination from Q.
   Remove the covered pairs from UC.
B) Repeat until UC is empty.
```

Figure 3: CATS heuristic algorithm for achieving pair-wise coverage

The number of test cases generated by the algorithm for a specific test problem is related to the number of candidates (k in the algorithm in Figure 4) for each test case. In general, larger values of k yield smaller numbers of test cases. However, Cohen et al. report that using values larger than 50 will not give any dramatic decrease in the number of test cases.

The heuristic nature of the algorithm makes it impossible to theoretically calculate the number of test cases in a test suite generated by the algorithm. However, empirical results show similar behavior as OA, i.e., $O(v^2_{max}$[31].

3.2.2 Random Combination Strategies

Random (Rand)
Creating a test suite by randomly sampling test cases from the complete set of test cases based on some input distribution (often uniform distribution) is an old idea with unclear origin. Duran and Ntafos [39] made an evaluation of random testing in 1984. In the scope of combination strategies, Mandl [1] may have been the first to mention the idea. Mandl states (without motivation) that a reasonable random selection would probably be about the same size as a test suite that simply takes the N variables in turn, and includes for each of them v test cases to cover the v interesting values of the variable. In 1997, Dunietz et al. [12] applied the random combination strategy with varying number of test cases.

3.3 Deterministic Combination Strategies

Deterministic combination strategies share the property that they produce the same test suite every time they are used. These strategies are divided into three subcategories: instant, iterative and parameter-based combination strategies. Instant combination strategies create the complete test suite all at once. Iterative combination strategies build up the test suite by adding one test case at a time based on what has already or not yet been covered. Parameter-based combination strategies build up the test suite by adding values to the test cases for one parameter at a time.

```
Assume test cases t_1 − t_{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 t_1 − t_{i−1}

A) Select candidates for t_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 k times and let t_i be the test
   case that covers the most pairs in UC. Remove those pairs from UC.

Repeat until UC is empty.
```

Figure 4: AETG heuristic algorithm for achieving pair-wise coverage

1	2	3
3	1	2
2	3	1

Figure 5: A 3×3 Latin Square

3.3.1 Instant Combination Strategies

Sometimes a tester wants to include her favorite test cases in the test suite. With instant combination strategies this will not affect the final result. The same test suite will be selected by the combination strategy regardless if other some [other] test cases have already been preselected.

Orthogonal Arrays (OA)
Orthogonal Arrays is a mathematical concept that has been known for quite some time. The application of orthogonal arrays to testing was first introduced by Mandl [1] and later more thoroughly described by Williams and Probert [10].

The foundation of OA is Latin Squares. A *Latin Square* is an $V \times V$ matrix completely filled with symbols from a set that has cardinality V. The matrix has the property that the same symbol occurs exactly once in each row and column. Figure 5 contains an example of a 3×3 Latin Square with the symbols $\{1, 2, 3\}$.

Two Latin Squares are *orthogonal* if, when they are combined entry by entry, each pair of elements occurs precisely once in the combined square. Figure 6 shows an example of two orthogonal 3×3 Latin Squares and the resulting combined square.

If indexes are added to the rows and the columns of the matrix, each position in the matrix can be described as a tuple $< x, y, z_i >$, where z_i represents the values of the $< x, y >$ position. Figure 7 contains the indexed Latin Square from Figure 5 and Figure 8 contains the resulting set of tuples. The set of all

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 6: Two orthogonal 3×3 Latin Squares and the resulting combined square

tuples constructed by a Latin Square satisfies pair-wise coverage.

Coordinates	**1**	**2**	**3**
1	1	2	3
2	3	1	2
3	2	3	1

Figure 7: A 3×3 Latin Square augmented with coordinates

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

Figure 8: Tuples from the 3×3 Latin Square that satisfies pair-wise coverage

To illustrate how orthogonal arrays are used to create test cases, consider a test problem with three parameters a, b, and c, where three values have been chosen as interesting for a, two for b, and two for c. To create test cases from the orthogonal array tuples a mapping between the coordinates of the tuples and the parameters of the test problem must be established. Let each coordinate represent one parameter and the different interesting values of the parameter map to the different values of that coordinate. In the case of the example, map a onto x, b onto y, and c onto z. This mapping presents no problem for parameter a, but for parameters b and c there are more coordinate values than there are values. To resolve this situation, each test case that has a coordinate value without a corresponding interesting value needs to be changed. Consider tuple 7 in Figure 8. The value of coordinate z 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

arbitrarily defined value, which in this case is 1 or 2. Sometimes 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 z are undefined and so should be changed to valid values. Set y to 1 and z to 2 and the changed tuple is identical to tuple 4 and thus is not needed 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_{max}^2, where $v_{max} = Max_{i=1}^{N} v_i$, N is the number of parameters, and v_i is the number of values of parameter i. Williams and Probert [10] give further details on how test cases are created from orthogonal arrays.

Sloane has collected a large number of precalculated orthogonal arrays of various sizes and made them available on the Web[2].

Covering Arrays (CA)
Covering Arrays [22] is a refinement of orthogonal arrays. A property of orthogonal arrays is that they are balanced, which means that each parameter value occurs the same number of times in the test suite. If only t-wise (for instance pair-wise) coverage is desired the balance property is unnecessary and will make the algorithm less efficient. In a covering array that satisfies t-wise coverage, each t-tuple occurs at least once but not necessarily the same number of times. Another problem with orthogonal arrays is that for some problem sizes there does not exist enough orthogonal arrays to represent the entire problem. This problem is also avoided by using covering arrays.

A covering array is constructed for a test problem from a set of building blocks:

- $O(k^2,\ k+1,\ k)$ is an orthogonal array with k^2 rows and $k+1$ parameters, which have at most k values each.
- $B(k^2-1,\ k+1,\ k,\ d) = O(k^2,\ k+1,\ k)$ with the first row removed and with the columns used d times each consecutively.
- $R(k^2-k,\ k,\ k,\ d) = O(k^2,\ k+1,\ k)$ with the k first rows and the first column removed and with the remaining columns used d times each consecutively.
- $I(c,\ d)$ is a matrix with c rows and d columns completely filled with "ones."
- $N(k^2-k,\ k,\ d)$ is a matrix with k^2-k rows and $k \times d$ columns, which of which consists of $k \times d$ submatrices filled with "twos", "threes", etc. up to k.

Depending on the size of the problem, these five building blocks are combined in different ways to create a test suite with pair-wise coverage over all parameters and parameter values. The number of test cases required to reach pair-wise coverage has a complexity of $N^2 + v_{max} log^2 v_{max}$ where N is the number of parameters and v_{max} is the maximum number of values of any one parameter.

Both the algorithms for finding suitable building blocks and for combining these to fit a specific test problem have been automated.

3.3.2 Iterative Combination Strategies

Iterative combination strategies are those in which one test case at a time is generated added to the test suite. Thus, a tester may start the algorithm with an already preselected set of test cases.

[2]URL: http://www.research.att.com/~njas/oadir, page visited March 2004.

Each Choice (EC)
The basic idea behind the *Each Choice* combination strategy is to include each value of each parameter in at least one test case. This is achieved by generating test cases by successively selecting unused values for each parameter. This strategy was invented by Ammann and Offutt [6], who also suggested that EC can result in an undesirable test suite since the test cases are constructed mechanically without considering any semantic information.

It is clear from the definition of EC that it satisfies each-used coverage. Further, the number of test cases in an EC test suite is at least $Max_{i=1}^{N} v_i$, where N is the number of parameters of the test problem and v_i is the number of values of parameter i.

Partly Pair-Wise (PPW)
Burroughs, Jain, and Erickson [7] sketch a "traditional" combination strategy in which all pair-wise combinations of the values of the two most significant parameters should be included, while at the same time including each value of the other parameters at least once. No details of the actual algorithm are given. Thus, it is impossible to assess the size of the generated test suite. However, it is clear that PPW satisfies each-used coverage.

Base Choice (BC)
The *Base Choice* (BC) combination strategy was proposed by Ammann and Offutt [6]. The essence of the idea behind BC was later described by Cohen et al. [43], and called "default testing." The first step of BC is to identify a base test case. The *base test case* combines the most "important" value for each parameter. Importance may be based on any predefined criterion such as most common, simplest, smallest, or first. Ammann and Offutt suggest "most likely used." Similarly, Cohen et al. call these values "default values." From the base test case, new test cases are created by varying the values of one parameter at a time while keeping the values of the other parameters fixed on the values in the base test case.

A test suite generated by BC satisfies each-used coverage since each value of every parameter is included in the test suite. If "most likely used" is used to identify the base test case, BC also satisfies the single error coverage criterion. The reason is that the base test case only consists of normal values and only one parameter at a time differs from the base test case.

Ammann and Offutt state that satisfying base-choice coverage does not guarantee the adequacy of the test suite for a particular application. However, a strong argument can be made that a test suite that does not satisfy base-choice coverage *is* inadequate.

A BC test suite has $1 + \sum_{i=1}^{N}(v_i - 1)$ test cases where N denotes the number of parameters of the test problem and v_i is the number of values of parameter i.

A variant of BC was proposed by Burr and Young [14], which they called *default testing*. In their version all parameters except one contains the default value, and the remaining parameters contain a maximum or a minimum. This variant does not necessarily satisfy each-used coverage.

All Combinations (AC)
The *All Combinations* (AC) combination strategy algorithm generates every combination of values of the different parameters. The origin of this method is impossible to trace back to a specific person due to its simplicity. It is often used as a benchmark with respect to the number of test cases [1, 6, 11].

An AC test suite satisfies N-wise coverage and contains $\prod_{i=1}^{N} v_i$ test cases, where N denotes the number of parameters of the test problem and v_i is the number of values of parameter i.

Antirandom (AR)
Antirandom testing was proposed by Malayia [44], and is based on the idea that test cases should be selected to have maximum "distance" from each other. Parameters and interesting values of the test object are encoded using a binary vector such that each interesting value of every parameter is represented by one or more binary values. Test cases are then selected such that a new test case resides on maximum Hamming or Cartesian distance from the already selected test cases. In the original description, the antirandom testing scheme is used to create a total order among all the possible test cases rather than attempting to limit the number of test cases. However, just like it is possible in random testing to set a limit to how many test cases that should be generated, the same applies to antirandom test cases. Antirandom testing can be used to select a subset of all possible test cases, while ensuring that they are as far apart as possible.

k-boundary (k-bound) and k-perimeter (k-perim)
With a single parameter restricted to values $[L, .., U]$, generating boundary values is easy. The tester can select the set $\{L,\ U\}$ or $\{L,\ L+1,\ U-1,\ U\}$. With multiple domains two problems arise: (1) the number of points grows rapidly and (2) there are several possible interpretations. Hoffman, Strooper and White [21] defined the k-boundary and k-perimeter combination strategies to handle these problems.

The *1-boundary* of an individual domain is the smallest and largest values of that domain (L and U). The *2-boundary* of an individual domain is the next to the smallest and the next to the largest values of that domain and so on ($L+1$ and $U-1$).

The *k-boundary* of a set of domains is the Cartesian product of the k-boundaries of the individual domains. Thus, the number of test cases of the k-boundary is 2^N, where N is the number of domains. In the general case, k-boundary does not satisfy any of the normal coverage criteria associated with combination strategies.

The *k-perimeter* test suite can be constructed by (1) including the k-boundary test suite, (2) forming all possible pairs of test cases from the k-boundary test suite such that the two test cases in a pair only differ in one dimension, e.g., $(L,\ L,\ L,\ L)$ and $(U,\ L,\ L,\ L)$, (3) For each identified pair of test cases, adding to the k-perimeter test suite all points in between the two test cases of the pair i.e., $\{(L+1,\ L,\ L,\ L), (L+2,\ L,\ L,\ L), ..., (U-2,\ L,\ L,\ L), (U-1,\ L,\ L,\ L)\}$.

k-perimeter does not satisfy any coverage criterion in the general case, however, 1-perimeter satisfies each-used coverage.

3.3.3 Parameter-Based Combination Strategies

There is only one parameter-based combination strategy, *In Parameter Order (IPO)*.

In Parameter Order (IPO)
For a system with two or more parameters, the in-parameter-order (IPO) combination strategy [18, 24, 27] generates a test suite that satisfies pair-wise coverage for the values of the first two parameters. The test suite is then extended to satisfy pair-wise coverage for the values of the first three parameters, and continues to do so for the values of each additional parameter until all parameters are included in the test suite.

To extend the test suite with the values of the next parameter, the IPO strategy uses two algorithms. The first algorithm, horizontal growth, as shown in figure 9, extends the existing test cases in the test suite with values of the next parameter The second algorithm, vertical growth, as shown in figure 10, creates additional test cases such that the test suite satisfies pair-wise coverage for the values of the new parameter.

Algorithm $IPO\_H\ (\tau, P_i)$
{
  Let τ be a test suite that satisfies pair-wise coverage for the values of parameters p_1 to p_{i-1}.
  Assume that parameter p_i contains the values $v_1, v_2, ..., v_q$
  $\pi = \{$ pairs between values of p_i and values of p_1 to $p_{i-1}\}$
  if $(|\ \tau\ | \leq q)$
  {
    for $1 \leq j \leq |\ \tau\ |$, extend the jth test in τ by adding value v_j and remove from π pairs covered by the extended test.
  }
  else
  {
    for $1 \leq j \leq q$, extend the jth test in τ by adding value v_j and remove from π pairs covered by the extended test
    for $q < j \leq |\ \tau\ |$, extend the jth test in τ by adding one value of p_i such that the resulting test covers the most number of pairs in π, and remove from π pairs covered by the extended test
  }
}

Figure 9: IPO_H – An algorithm for horizontal growth of a test suite by adding values for new parameters

```
Algorithm IPO_V (τ, π)
{
  Let τ′ be an empty set
  for each pair in π
  { assume that the pair contains value w of p_k, 1 ≤ k < i, and value u of p_i
  {
    if (τ′ contains a test case with ‘‘-’’ as the value of p_k
    and u and the value of p_i)
        modify this test case by replacing the ‘‘-’’ with w
  else
        add a new test case to τ′ that has w as the value of p_k, u as
        the value of p_i, and ‘‘-’’ as the value of every other parameter;
  }
  τ = τ ∪ τ′
}
```

Figure 10: IPO_V – An algorithm for vertical growth of a test suite by adding values for new parameters

The nature of the IPO algorithms makes it difficult to theoretically calculate the number of test cases in a test suite generated by the algorithm. An algorithm that closely resembles the IPO algorithm was informally described by Huller [23].

3.4 Compound Combination Strategies

When two or more combination strategies are used together they form a *compound combination strategy*. From the literature we have taken two examples to illustrate this.

All or Random (ACoRand)
One problem with the AC strategy is that it may generate infeasibly large test suites. For test problems with less that Y possible test cases the *All or Random* compound combination strategy generates all combinations of parameter values. If the number of possible test cases exceed Y, exactly Y test cases are (deterministically) pseudo-randomly selected. Kropp, Koopman, and Siewiorek [15] suggest 5000 as a suitable value for Y.

An ACoRand test suite contains $t = \prod_{i=1}^{N} v_i$ tests if $t \leq Y$ else Y test cases, where N denotes the number of parameters of the test problem, v_i is the number of values of parameter i, and Y is a predefined maximum number of test cases.

Base Choice and AETG (BCAETG)
In a comparative study, Grindal, Lindström, Offutt, and Andler [31] show that the union of the test suites generated by BC and AETG is more effective in finding faults than each one individually. A more efficient alternative than taking the union may be to use the BC test suite as input to the generation of the AETG test suite.

4 Comparing Combination Strategies

This section contains comparisons of combination strategies with respect to coverage criteria and size of generated test suite

4.1 Subsumption

A subsumption hierarchy for the combination coverage criteria reported in the literature is shown in Figure 11. The definition of subsumption is from Rapps and Weyuker [45]: coverage criterion X *subsumes* coverage criterion Y iff 100% X coverage implies 100% Y coverage (Rapps and Weyukers' original paper used the term inclusion instead of subsumption). We have included all known coverage criteria and in the light of subsumption tried to generalize them.

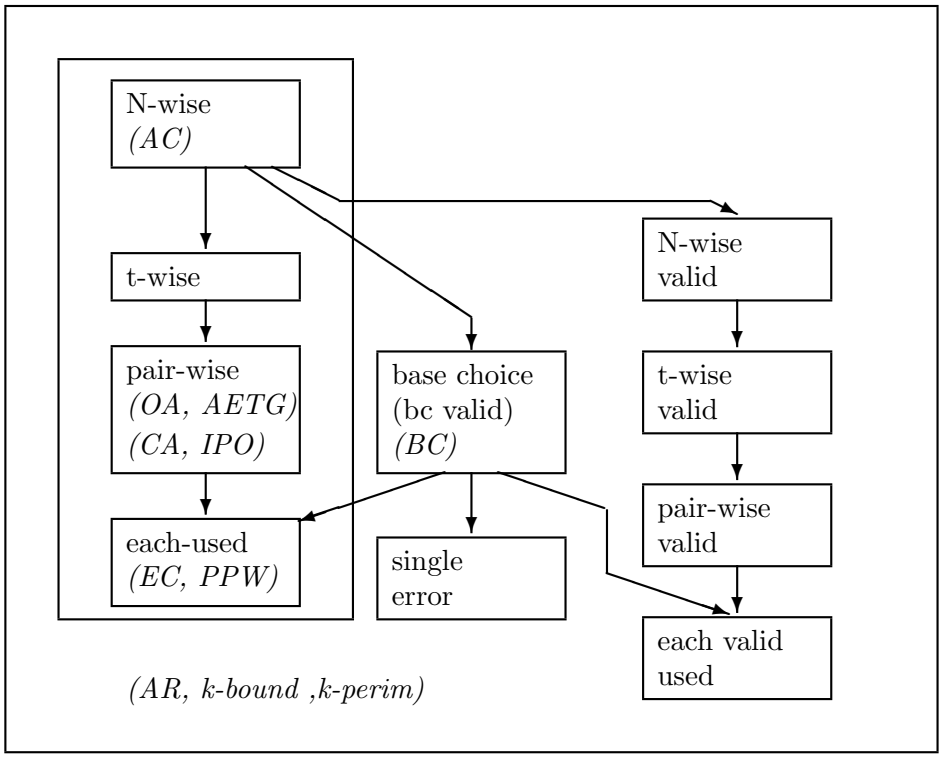

Figure 11: Subsumption hierarchy for the coverage criteria related to combination strategies. Italics indicate combination strategies.

The left, boxed-in, column of coverage criteria in Figure 11 represent criteria that do not use semantic information. N-wise coverage requires that every combination of all parameter values are included in the

test suite. "t-wise coverage" is used as a short form for every level of coverage from $N-1$ down to 2. It is straightforward to see that j-wise coverage subsumes $j-1$-wise coverage, for any j such that $2 \leq j \leq N$.

The right column contains the coverage criteria based only on valid values. For the same reason that j-wise coverage subsumes $j-1$-wise coverage, it is easy to see that the j-wise **valid** coverage criterion subsumes the $j-1$-wise valid coverage criterion.

Note that i-wise coverage where $1 \leq i \leq N-1$ does *not* subsume i-wise valid coverage. This is easily demonstrated for the case where $i = 2$. Consider a three parameter problem, each with one valid value v and one erroneous value e. The following four combinations satisfy pair-wise coverage since each possible pair of values are included $[(v,v,e),\ (v,e,v),\ (e,v,v),\ (e,e,e)]$. However, pair-wise valid coverage is not satisfied since a pair of two valid parameters only count if the remaining parameters also are valid. Since every combination in the example contains at least one erroneous value, the achieved pair-wise valid coverage is 0%. N-wise coverage contains every possible combination so by definition it subsumes all other coverage criteria, including N-wise valid coverage.

The single error coverage criterion represents a different class of coverage criteria. Although it is possible to generalize this idea into multiple error coverage this would introduce the possibility of masking, so it is omitted. At first it may seem that the single error coverage criterion would subsume the each-used coverage criterion. However, a counter example can be constructed. Consider a two-parameter problem where the first parameter has two valid values (v11 and v12) and one erroneous value (e11). The second parameter has one valid value (v21) and one erroneous value (e21). The two test cases $[(e11, v21),\ (v11, e21)]$ satisfy the single error coverage criterion but not the each-used coverage criterion.

An interesting property of base choice coverage, when assuming that all base choices are valid, is that it subsumes the each-used, each valid used, and single error coverage criteria. By definition, the base test case contains only valid values. From the definition of the base choice algorithm we get that all test cases differs from the base test case by the value of only one parameter. Further, we know that each interesting value of every parameter is included in some test case. Thus, the each-used criterion is subsumed. Further, every valid value of every parameter must appear in at least one test case in which the rest of the values are also valid, since the base test case contains only valid values. Thus, the each valid used criterion is subsumed. Finally, for the same reason, every invalid value of every parameter must occur in exactly one test case in which the rest of the values are valid. Thus, the single error criterion is subsumed.

4.2 Size of Generated Test Suite

Table 2 gives an overview of the sizes of the generated test suites. In some cases only approximate values are possible to give. The main reason for this is that those algorithms contain some degree of randomness why using the strategy on a test problem may yield different solutions. Also, only a subset of the described combination strategies is included in the table. There are several reasons. Some strategies, i.e., Rand and AntiRand, have no natural limit. Other strategies, i.e., k-bound and k-perim, differ greatly in the size of the generated test suite depending on the value of k. The algorithm of PPW is not described and finally CATS, having an element of randomness, is not investigated well enough to be described in terms of size of generated test suite.

In addition to the formulas relating the size of the test problem to the size of the generated test suite, table 2 contains two example test problems one with few parameters and many values and one with many parameters with few values.

Combination Strategy	**Test Suite Size**	**Example 1** $N = 8, V_i = 4$	**Example 2** $N = 4, V_i = 8$
AETG	$\sim V_{max}^2$	~ 16	~ 64
OA	$\sim V_{max}^2$	~ 16	~ 64
CA	$\sim N^2 + V_{max} log^2 V_{max}$	~ 65	~ 23
EC	V_{max}	4	8
BC	$1 + \sum(V_i - 1)$	25	29
AC	$\prod V_i$	65536	4096
IPO	$\sim V_{max}^2$	~ 16	~ 64

Table 2: Definite or approximate ($\sim$) test suite sizes for a problem with N parameters and V_i values of the i:th parameter. V_{max} is the largest V_i.

5 Combination Strategies – Experience and Evaluations

This section contains brief descriptions of the reported results relating to combination strategies. The papers described in this section are organized into eight groups according to their main focus (the combination strategy) in each paper. Within each group the papers are presented in chronological order. While we considered organizing all of the papers in chronological order, the number of papers (over 30!) convinced us that some sort of categorization was needed.

5.1 General Focus

The five papers in this group considered aspects that are general to all combination strategies.

5.1.1 Piwowarski, Ohba and Caruso

Piwowarski, Ohba, and Caruso [5] described how to successfully apply code coverage as a stopping criterion during functional testing. The authors formulated functional testing as a problem of selecting test cases from a set of possible test cases made up of all combinations of values of the input parameters. No algorithm was given to choose from this sample, but the authors discuss several methods informally. One method is to fix some parameter values based on some equivalence partition assumption. Another method is by random selection.

5.1.2 Dunietz, Ehrlich, Szablak, Mallows and Iannino

Dunietz, Ehrlich, Szablak, Mallows, and Iannino [12] examined the correlation between t-wise coverage and achieved code coverage. The investigation was based on a case study in which a maintenance system for an AT&T network was tested. For the purpose of comparison, two different models, both of which contained parameters and parameter partitions, were developed from the same specification. The models contained 72 and 162 combinations, C. For each i, $1 \leq i \leq C$, 10 abstract test suites were generated for each of the two models by random sampling without replacement. For every test suite, whether t-wise coverage with

Assume a test problem with three parameters a, b, and c, each with 2 possible values.

Let $tc1 = (a1,\ b1,\ c1)$, $tc2 = (a2,\ b2,\ c1)$, and $tc3 = (a1,\ b1,\ c2)$

The following nine pairs are covered by the three test cases:
$(a1, b1),\ (a1, c1),\ (b1, c1),\ (a2, b2),\ (a2, c1),\ (b2, c1),\ (a1, b1),\ (a1, c2),\ (b1, c2)$
Note that the pair $(a1, b1)$ occurs twice.

The following four pairs are not covered by the three test cases:
$(a1, b2),\ (a2, b1),\ (a2, c2),\ (b2, c2)$

$2 - coverage = 8/12 = 75\%$
$2 - diversity = 8/9 = 89\%$

Figure 12: An example of 2-coverage and 2-diversity of a test suite

$t = 1,\ 2,\ ..,\ n$ was calculated. Further, ten real test suite instances of each abstract test suite were created by randomly selecting a value for each parameter partition in each test case. The resulting test cases were then executed and the code coverage achieved by the complete test suite was monitored. Two code coverage measures were used, block (statement) coverage and path coverage.

The conclusion of this study was that a test suite that satisfies t-wise coverage, where $t \geq 2$, obtains comparable results in terms of block (statement) coverage. Thus, pair-wise coverage has comparable effectiveness as all-combination coverage with respect to block (statement) coverage. Larger values of t are required to achieve path coverage comparable to all-combination coverage.

5.1.3 Dalal and Mallows

Dalal and Mallows [16] provide a theoretical view of combination strategies. They provide a model for software faults in which faults are classified according to how many parameters (factors) need distinct values to cause the fault to result in a failure. A t-factor fault is triggered whenever the values of some t parameters are involved in triggering a failure. Each possible fault can be specified by giving the combination(s) of values relevant for triggering that fault. Further they provide two measures of efficiency of a combination strategy test suite. The t-wise coverage (in Dalal's and Mallow's words *t-coverage*) is the ratio of distinct t-parameter combinations covered to the total number of possible t-parameter combinations.

The *t-diversity* is the ratio of distinct t-parameter combinations covered to the total number of t-parameter combinations in the test suite. Thus, this metric summarizes to what degree the test suite avoids replication. Figure 12 shows an example of these two metrics.

5.1.4 Kuhn and Reilly

Kuhn and Reilly [29] have investigated 365 error reports from two large open source software projects, 194 from the Mozilla web browser and 174 from the Apache web server. Their aim was to find answers to the following three questions: (1) Is there a point in which a test suite that satisfies t-wise coverage is nearly as effective as a test suite that satisfies $t+1$ coverage? (2) What is the appropriate value for t for particular classes of software? (3) Does this value differ for different types of software and by how much? Their findings indicate that a test suite that satisfies 4-wise coverage would have found more than 95% of the errors. Further their results show that test suites that satisfies $2-6$ -wise coverage have similar effectiveness for both studied applications.

5.1.5 Grindal, Lindström, Offutt and Andler

Grindal, Lindström, Offutt, Andler [31] present the results of a comparative evaluation of five combination strategies. One of the combination strategies investigated, Each Choice, satisfies the each-used criterion. Two of the investigated combination strategies, Orthogonal Arrays and Heuristic Pair-Wise (AETG), satisfy the pair-wise criterion. The fourth, All Values, generates all possible combinations of the values of the input parameters. The fifth, Base Choice, satisfies the each-used criterion but also uses some semantic information to construct the test cases.

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

As expected, Each Choice finds the smallest amount of faults among the evaluated combination strategies. Base Choice performs as well, in terms of detecting faults, as the pair-wise combination strategies despite fewer test cases. Their analysis of the results showed that the performance of Each Choice is unpredictable in terms of which faults will be detected. Moreover, Base Choice and the pair-wise strategies target different types of faults to some extent. The conclusion of the experiment was that Base Choice should be used alone for less important software and together with Heuristic Pair-Wise in a compound combination strategy for more critical software.

5.2 EP, BVA, CP and All Combinations

The three papers in this group considered aspects of choosing "interesting values" by techniques such as equivalence partitioning (EP), boundary value analysis (BVA), category partitioning (CP), and choosing all combinations of values.

5.2.1 Toczki, Kocsis, Gyimóthy, Dányi and Kókai

Toczki, Kocsis, Gyimóthy, Dányi, and Kókai [3] described the state of SYS/3, a software development tool. SYS/3 contains work processes and tools for several activities during systems development, including project management and control, document handling, and testing. A test case generator called T-GEN has been implemented for the Category Partition method (CP) [2]. In addition to handling the input space, T-GEN may also be used to generate test cases based on the output space in a similar manner as the input space is handled in CP. Also, T-GEN can generate complete or partial test scripts under certain conditions.

5.2.2 Kropp, Koopman and Siewiorek

Kropp, Koopman, and Siewiorek [15] describe the Ballista testing methodology, which supports automated robustness testing of off-the-shelf software components. Robustness was defined as the degree to which a software component functions correctly in the presence of exceptional inputs or stressful conditions. The underlying testing strategy is an object-oriented approach based on parameter types and values instead of component functionality. The application of the Ballista methodology is demonstrated on several implementations of the POSIX operating system C language API, which consists of 233 calls. Together the POSIX calls use 190 test values across 20 data types. The expected results of the different tests are limited to "does not crash", "halts", and "returns a value". The simplest Ballista operating mode generates a test suite that has all combinations of all parameter values. If the total number of combinations is too large to be feasible, a deterministic pseudo-random sample is used. In the experiment reported in the paper, the limit for generating all combinations was set to 5000 test cases. Only seven of the POSIX functions tested require more than 5000 test cases.

5.2.3 Daley, Hoffman and Strooper

Daley, Hoffman, and Strooper [28] showed how combination strategies can be used to test Java classes. A main focus of their work was on automation of test generation and execution. This is accomplished with the "Roast" framework, which consists of four steps. (1) *Generate* creates the combinations of parameter values to be tested, that is, the input portions of each test case. (2) *Filter* removes invalid combinations from the test suite. (3) *Execute* drives the execution of the test cases. (4) *Check* compares the expected with the actual output. Apart from a parameter model of the test problem, the expected output is the only manual intervention.

The combination generation includes two general strategies, both based on boundary value analysis. The tester provides information about the parameters, and for each parameter the different domains, or equivalence classes. The first strategy, k-boundary, is the Cartesian product of the boundaries of the different parameters. The second strategy, k-perimeter, is the boundary plus "the points in between." To illustrate the two strategies, assume a two parameter problem p_1 and p_2, with the integer domains [0, 1, 2, 3, 4] and [5, 6, 7, 8, 9]. The 1-boundary test suite contains the four test cases $[(0,5),\ (0,9),\ (4,5),\ (4,9)]$. The 2-boundary test suite uses the values closest to the extreme points resulting in the test suite $[(1,6),\ (1,8),\ (3,6),\ (3,8)]$. The 1-perimeter test suite contains the following 16 test cases: $[(0,5),\ (0,6),\ (0,7),\ (0,8),\ (0,9),\ (4,5),\ (4,6),\ (4,7),\ (4,8),\ (4,9),\ (1,5),\ (2,5),\ (3,5),\ (1,9),\ (2,9),\ (3,9),]$. These 16 tests form the complete perimeter.

k-perimeter significantly reduces the number of test cases for test problems with few parameters and many values per parameter, whereas k-boundary is more efficient when the opposite occurs, when there are many parameters with few values. In cases where parameter values depend on each other, Roast contains facilities to represent one large test problem with dependencies as several smaller test problems without dependencies.

5.3 Orthogonal and Covering Arrays

The six papers in this group considered aspects of choosing combinations by organizing the values into arrays.

5.3.1 Mandl

Mandl [1] may have been the first researcher who attempted to solve a testing problem with combination strategies. His focus was on orthogonal arrays. He states that the size of a test suite generated by OA is v^2. Underlying this statement is the assumption that all parameters contain exactly v interesting values. Mandl reports that OA was intended to design test cases used by the Ada Compiler Validation Capability (ACVC) test suite. No results from testing were reported.

5.3.2 Brownlie, Prowse and Phadke

Brownlie, Prowse, and Phadke [4] reported results from a case study. The Orthogonal Arrays Testing System (OATS) tool, which is based on the OA combination strategy, was used to test a PMX/StarMAIL release. OATS automatically identifies orthogonal arrays and generates test suites that are based on descriptions of the parameters and interesting values of the test object.

In the case study OATS was applied both to the PMX/StarMAIL hardware and software configurations to the functionality of the system. 354 test cases were generated by OATS and an additional 68 test cases were created by hand to cover special conditions and to add tests that applied to specific configurations.

A previous release of PMX/StarMAIL was tested "conventionally." The results from this test were used as points of comparison when evaluating the results obtained by OATS and the manually created test cases. The authors estimated that 22 percent more faults would have been found if OATS had been used despite the fact that conventional testing have required twice as much time to perform. Further, they report that 1.5 years after the tested system was put into production, an additional three faults have been found that were missed in the case study. All of these faults were outside the scope of the planned testing.

The conclusion drawn by the authors is that OA should be used whenever there are many independent hardware, software, or functional parameters to test.

5.3.3 Williams and Probert

Williams and Probert [10] illustrated the applicability of OA to test network elements of a telephony system. The OA combination strategy was used to select system configurations for testing rather than individual test cases. The authors pointed out that the cost for setting up a certain configuration is often substantial. Further, each configuration requires complete test suites to be executed, which adds to the cost of testing the configuration. For this reason, it is important to find ways to identify a small number of configurations to test. One of the main aims of the paper was to present a self-contained guide to apply the theory of OA. Thus, most of the paper is spent on describing OA and the method to construct orthogonal Latin Squares of desired sizes and how to handle various practical situations that are not handled by the basic OA combination strategy.

5.3.4 Williams

Williams [22] continues to explore the applicability of pair-wise coverage to configuration testing. In this paper, Williams introduced Covering Arrays (CA) as an alternative to Orthogonal Arrays. A comparative analysis of the number of test cases generated by CA and IPO is reported. CA and IPO behaved similarly, often to the advantage of CA, with only minor differences in number of test cases for the 16 different test problem sizes. However, CA outperforms IPO by almost three orders of magnitude for the largest test problems in terms of time taken to generate the test suites.

5.3.5 Williams and Probert

Williams and Probert [25] stated that the use of an interaction test coverage metric allows testers to both evaluate already existing test suites and to generate new test suites that satisfy a predetermined level of interaction. Based on this, Williams and Probert suggest the use of an *interaction element*, which may be used in the same way statements and decisions are used for code coverage. An interaction element is a specific (sub)combination of parameter values. By reasoning about types of interaction elements, Williams and Probert formally defined a set of coverage criteria, including t-wise coverage.

Williams and Probert showed that the general problem of finding a minimal set of test cases that satisfies t-wise coverage can be NP-complete.

5.3.6 Williams and Probert

Williams and Probert [26] continue the work from Williams previous paper [22] in finding an algorithm that will generate a minimal test suite that satisfies pair-wise coverage. Previously, they provided a fast approximate algorithm by using different building blocks. This time they show how the interaction test coverage problem may be formulated as a $[0, 1]$ integer programming problem. Using general solution mechanisms for such problems, it is possible to find a minimal solution. These problems have previously been shown to be NP-complete, which indicates that finding an optimal solution for the interaction test coverage problem may also be NP-complete. The uncertainty comes from the fact that not all $[0, 1]$ integer programming problems can be converted to an interaction test coverage problem.

5.4 AETG

A test tool called Automatic Efficient Test Generator (AETG) was created at Bellcore (now Telcordia) and a number of papers were published that used it in practical industrial settings.

5.4.1 Burroughs, Jain and Erickson

Burroughs, Jain, and Erickson [7] presented the first use of the Telcordia (Formerly Bellcore) test tool called Automatic Efficient Test Generator (AETG). They illustrated the use of AETG with two examples in which the test suite generated by AETG is compared with corresponding test suites generated by AC and PPW on the basis of number of test cases and pair-wise coverage (breadth of coverage in their terms). AETG produced similar results in terms of test cases as PPW and an order of magnitude lower number of test cases than AC. By definition AETG and AC reach 100% pair-wise coverage while PPW only reaches $30 - 35\%$. The authors concluded that AETG is an effective and efficient way to create test cases.

5.4.2 Cohen, Dalal, Kajla and Patton

Cohen, Dalal, Parelius and Patton [43] described how to use AETG in the context of screen testing, that is, testing the input fields for consistency and validity across a number of screens. From previous investigations they knew that most of the faults in the screens are caused by the value of one single parameter or the interaction of two parameters. Thus, although AETG can generate a test suite for t-wise coverage for an arbitrary t, the authors predicted that pair-wise coverage will be what testers will need.

The authors theoretically compared the underlying algorithm of AETG, without revealing it, with OA. They showed that by removing the requirement of balance in OA, AETG is able to reduce the number of test

cases needed to for pair-wise coverage. Further, the authors pointed to several other advantages of AETG over OA, for example, that the algorithm is always able to find a solution to a test problem.

The authors describe the general features of AETG and conclude by reporting the results of a small experiment in which AETG was used to test ten screens of an inventory database system. No comparative analysis of the results was given.

5.4.3 Cohen, Dalal, Parelius, Fredman and Patton

Cohen, Dalal, Parelius and Patton [9] presented the first publicly available descriptions of the algorithms implemented in the Automatic Efficient Test Generator (AETG). The same information with more details was later presented by Cohen, Dalal, Fredman, Patton [11].

Results of two experiments were described. In the first experiment, AETG was used to test ten Unix commands and the code coverage was measured. The authors report that pair-wise tests resulted in over 90% block coverage. Specifically, Cohen et al. experimented with the Unix command "sort." Different ways of modeling the parameters and values were tried and the results were compared with the results from a randomly generated test suite, and from "all" combinations. The AETG generated test suites consistently produced better coverage than the randomly produced test suites.

Further, Cohen et al. used the AETG system to test two versions of a Telcordia product. A total of 75 parameters were identified, and 28 test cases were created to satisfy pair-wise coverage. Even though the AETG tests were executed at the end of the normal product development, several new faults, both in the specification and in the implementation, were detected. The main conclusion drawn by the authors was that combination strategies in general and the AETG system in specific are both effective and efficient at functional testing.

5.4.4 Burr and Young

Burr and Young [14] report on using AETG to test Nortel software. They tested a third party email product that converts email messages from one format to another. The most important step according to the authors was the modeling of the input space, in their case the possible ways to compose email messages. An augmented BNF that describes the syntax and format of the e-mail messages was used. Thus the authors were faced with the challenge of converting these BNF rules into AETG constructs. The AETG model was iteratively constructed and the final version was used to create a suite of test vectors, i.e., test case inputs. The next step was to convert these abstract test vectors into real test cases. A Perl script was written that converted the AETG representation of the test case inputs to real email messages.

The code coverage of the tests was monitored for two purposes. First, they wanted to verify how complete the specification was. Second, they wanted to measure the code coverage of AETG tests. Initial experiments showed that ad hoc testing resulted in about 50% block and decision coverage. By continually applying code coverage as the AETG models were refined, decision coverage was increased to 84%. The power of AETG combined with automatic translation into real test cases made it easy to find test cases that contributed to the coverage. Usually, the code needs to be analyzed to find test cases that will increase coverage. Due to the simplicity of generating test suites, many "what if" scenarios could be tested without increasing the number of test cases. This turned the traditionally cumbersome analysis of the code into a trial-and-error type of process.

To compare the AETG results with an alternate test generation technique, a restricted version of BC was used to create a test suite from the same input model. Of the reachable code, AETG reached 93% block

coverage with 47 test cases, compared to 85% block coverage for the restricted version of BC using 72 test cases.

5.4.5 Dalal, Jain, Karunanithi, Leaton and Lott

Dalal, Jain, Karunanithi, Leaton, and Lott [17] reported results from two experiments in which principles of model-based testing were used to automatically generate test cases for Bellcore's Intelligent Service Control Point. The authors reported two significant advantages over manual testing. The first is coping with change. They found that changes to the product or the test criteria have less impact on the test organization if test cases can be regenerated. The second is that a systematic choice of test cases, in their research created by AETG, can increase the quality of the product by detecting more faults than with manual testing.

In addition to selecting the test cases, the work also included the implementation of a test scaffolding to automatically execute the test cases and compare actual and expected results. The implementation of the scaffolding was the most labor intensive part of this work.

In total, more than 5000 test cases were generated and executed, of which about 400 failed. The authors reported four lessons learned. First, the model of the test data is fundamental and requires considerable domain expertise. Second, model-based testing is in fact a development project since it consists of much testware that needs to be developed and maintained. Third, change must be handled and unnecessary manual intervention should be avoided. Fourth, technology transfer requires careful planning, that is, changing a work process or a state-of-practice is a project in itself.

5.4.6 Dalal, Jain, Karunanithi, Leaton, Lott, Patton and Horowitz

Dalal, Jain, Karunanithi, Leaton, Lott, Patton, and Horowitz [20] presented the architecture of a generic test-generation system based on combination strategies. A test data model is derived from the requirements and then used to generate test input tuples. These are in turn refined into real test inputs and expected outputs. The test inputs are fed to the system under test, which generates actual output. Finally the actual and expected outputs are compared.

The authors claim that the model-based testing depends on three key technologies: the notation used for the data model, the test generation algorithm, and the tools for generating the test execution infrastructure including expected output. The notation and algorithms can be used in multiple projects but not the tools.

The authors also present the results from four case studies in which their model-based test architecture has been used. Two of these case studies were reported elsewhere [17]. The third case study concerned testing a rule-based system used to assign work requests to technicians. Five separate models were used to generate tests for the system. Two types of tests were generated and run. The first type concerned the initial assignment of jobs at the start of the day. The second type of tests focused on the dynamic assignment of tests during the course of the day. The latter was much harder, since these tests depend on the current state of the database, which dynamically changes. This, in turn, means that checking actual output is very difficult since all data is related. In this case study the checking of the data was performed manually to a large extent. Thirteen tests were run, resulting in block coverage of 84%. Although relatively good, the authors suggest that block coverage may not be significant for an object-oriented system that implements a rule processing mechanism. The tests revealed four failures. The authors claim that the relatively small amount of faults found was because checking the results was complicated.

The fourth case study consisted of testing a single GUI window in a large application. Due to the complexity of the window the creation of the input model turned out to be the most difficult task. 159 test

cases were generated and executed automatically since the expected results were easy to describe and check. Six failures were revealed and reported, three of which were uniquely discovered by the automatic testing approach.

The main lessons learned were that the model of the test data is fundamental and iteration is often required to find a suitable model. The authors also saw benefit in designing the test scaffolding with intermediate results stored in files, which can be manually changed. When executing thousands of tests automatically, the tester must be able to start the execution at an arbitrary point in the test suite without having to worry about the state of the test object. Finally, the authors give advice on how to change a process and how to introduce new tools.

5.4.7 Dalal, Jain, Patton, Rathi and Seymore

Dalal, Jain, Patton, Rathi, and Seymour [19] described a publicly available web interface that allows testers to use AETG. The tester inputs the functional model of the test object either through a set of web GUI widgets or through a text-based notation. AETG can produce up to three test suites. First is a valid test suite, which contains valid combinations of parameter values. Second is an invalid test suite, which contains test cases that violate constraints in parameter value combinations, as defined by the user. Third is an illegal test suite, which contains test cases with values explicitly declared to be illegal by the tester.

5.5 Base Choice

One paper was published that introduced the idea of base choice.

5.5.1 Ammann and Offutt

The seminal work on BC by Ammann and Offutt [6] theoretically compared the number of test cases needed to fulfill base-choice coverage with the number of test cases needed to fulfill each-used coverage and N-wise coverage. They concluded that BC is relatively inexpensive and yet effective.

Ammann and Offutt also demonstrated the feasibility of BC with a small experiment. MiStix is a simple file handling system with ten operations for creating, deleting, copying, and moving files and directories. The implementation is about 900 lines of C code. A complete Z specification had been developed for MiStix for a previous study. A complete BC test suite was generated from the Z specification by hand. The paper reported extensive information about the identification of parameters and interesting values of each parameter. In particular, the paper illustrated how Z preconditions and postconditions can be used to identify parameters and interesting values more or less mechanically.

The final BC test suite contained 72 test cases for the ten operations. Five of the 72 test cases were duplicates and some of the test cases were superseded by others in the sense that they were preconditions of other test cases. The test suite detected 7 out of 10 known faults.

5.6 In Parameter Order

Two papers appeared on the subject of choosing values based on the order of parameters.

5.6.1 Lei and Tai

Lei and Tai describes an algorithm called In-Parameter-Order (IPO) for generating a test suite that satisfies pair-wise coverage [18]. An extended version of the same paper was presented later [24] and an extract was also published [27]. The papers describe the framework of the IPO strategy, an optimal algorithm for the vertical growth of the test suite, and two alternative algorithms for the horizontal growth of the test suite. The algorithms have been implemented in Java in a tool called PairTest, which provides a graphical user interface to make the tool easy to use.

The IPO combination strategy is compared with the AETG combination strategy in the paper. The two combination strategies are shown to perform similarly with respect to number of test cases for six different test problem sizes. Tai and Lei also show that the time complexity of IPO is superior to the time complexity of AETG. IPO has a time complexity of $O(v^3N^2log(N))$ and AETG has a time complexity of $O(v^4N^2log(N))$, where N is the number of parameters, each of which has v values.

5.6.2 Huller

Based on a real example of testing ground systems for satellite communications, Huller [23] showed that pair-wise configuration testing may save more than 60% in both cost and time compared to quasi-exhaustive testing. The set of configurations that satisfied pair-wise coverage was developed by hand via an algorithm that resembles IPO, although no reference is given.

5.7 Antirandom Testing

An approach for choosing values based on the opposite of random selection has been discussed in two papers.

5.7.1 Malaiya

Malaiya [44] presented seminal work on antirandom testing. Tests are added in sequence and each value is represented as a sequence of bits. Antirandom means that each new test case added resides as far from all previous test cases in the sequence as possible, using Hamming or Cartesian distance measured on the bit representation.

The paper presented two algorithms to create an antirandom sequence of test cases. The first is a naive method that requires an exhaustive search. The second exploits properties of an antirandom sequence of test cases to create the final result more quickly. Further, the paper illustrates how checkpoint encoding can be used to identify a mapping between the parameters of the test problem and the corresponding test cases. A number of considerations and observations for this mapping are listed.

For the selected checkpoints, the intent of antirandom testing is to generate all possible combinations. By executing the test cases in the sequence they are generated, that is, as far apart from each other as possible, the hypothesis is that antirandom testing is likely to detect the presence of faults earlier than random testing. No data to support this hypothesis was presented in the paper.

5.7.2 Yin, Lebne-Dengel and Malayia

Yin, Lebne-Dengel, and Malayia [13] used three benchmark programs to compare antirandom testing with random testing. Test cases were generated one by one according to the different evaluated combination strategies. Each new test case was executed and the increase in branch, loop, relational, and total coverage

was monitored. The antirandom combination strategy consistently performed well by reaching high coverage with few test cases. The authors concluded that the results suggest that antirandom testing might be a worthwhile alternative to other black-box test methods. However, the investigated benchmarks are small and might not be representative. Also, the encoding used for the tests can have a strong affect on the results, so encoding rules need to be further investigated.

5.8 Fractional Factorial Designs

Selecting test values has a strong relationship to scientific experiments, and the experimental method of factorial design was suggested to help select values.

5.8.1 Heller

Heller [8] uses a realistic example to show that testing all combinations of the parameter values is infeasible in practice. The paper concluded that there we need to identify a subset of combinations of manageable size. Heller suggests using fractional factorial designs. These designs have traditionally been used in physical experiments to decrease the number of experiments. The underlying mathematics of the fractional factorial designs allows researchers to establish which factors cause which results. Heller does not describe specifically how is use fractional factorial designs in software testing. The paper explains how to translate a test problem into a representation for factorial designs. It is also shown that the number of test cases from fractional factorial designs is less than all combinations.

5.8.2 Berling and Runesson

Berling and Runesson [30] describe how to use fractional factorial designs used for performance testing. Specifically, they focus on how to determine the effect of different parameters on the performance of a system with a minimal or near minimal set of test cases. Berling and Runesson state that when the result variable has a nominal type, (for example, true or false) the property of fractional factorial designs cannot be exploited.

5.9 Missing Papers

The following two papers have proven hard to acquire. According to the papers referencing these papers, both of them contain information about attempts to automate the generation of test cases according to some combination strategy algorithm. The papers are included in this survey for completeness.

5.9.1 Biyani and Santhanam

This paper by Biyani and Santhanam [46] contains information about a tool called TOFU. It is referenced by Kropp et al. [15].

5.9.2 Sherwood

This paper by Sherwood [42] contains among other things a description of a tool called CATS implementing an early heuristic algorithm for pair-wise coverage. The paper is referenced by Dunietz et al. [12] and by Cohen et al. [9, 11, 16].

6 Conclusions

This survey demonstrates the wide variety of existing combination strategies and reports on their usages. It also suggests a classification scheme for the existing combination strategies based on properties of their algorithms. The survey gives an overview of the coverage criteria normally associated with combination strategies and shows how these coverage criteria are interrelated in a subsumption hierarchy. Finally, this survey also relates the surveyed combination strategies based on the approximate sizes of the test suites generated by the combination strategies.

7 Acknowledgments

The authors would like to thank the whole Distributed Real-Time Systems group at the department of Computer Science, University of Skövde. In particular we owe our gratitude to Birgitta Lindström for general support and fruitful discussions about coverage criteria and subsumption.

References

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

[2] T.J. Ostrand and M.J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31:676–686, June 1988.

[3] J. Toczki, F. Kocsis, T. Gyimóthy, G. Dányi, and G. Kókai. Sys/3-a software development tool. In *Proceedings of the third international workshop of compiler compilers (CC'90), Schwerin, Germany, October 22-24, 1990*, pages 193–207. Springer Verlag, Lecture Notes in Computer Science (LNCS 477), 1990.

[4] R. Brownlie, J. Prowse, and M.S. Phadke. Robust Testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal*, pages 41–47, May/June 1992.

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

[6] P.E. Ammann and A.J. 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.

[7] K. Burroughs, A. Jain, and R.L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Proceedings of the IEEE International Conference on Communications (Supercomm/ICC'94), May 1-5, New Orleans, Louisiana, USA*, pages 745–752. IEEE, May 1994.

[8] E. Heller. Using design of experiment structures to generate software test cases. In *Proceedings of the 12th International Conference on Testing Computer Software, New York, USA, 1995*, pages 33–41. ACM, 1995.

[9] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, pages 83–88, September 1996.

[10] A.W. Williams and R.L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE96), White Plains, New York, USA, Oct 30 - Nov 2, 1996*, Nov 1996.

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

[12] 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.

[13] H. Yin, Z. Lebne-Dengel, and Y.K. Malayia. Automatic Test Generation using Checkpoint Encoding and Antirandom Testing. Technical Report CS-97-116, Colorado State University, 1997.

[14] K. Burr and W. 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.

[15] N.P. Kropp, P.J. Koopman, and D.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.

[16] S.R. Dalal and C.L. Mallows. Factor-Covering Designs for Testing Software. *Technometrics*, 50(3):234–243, August 1998.

[17] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, and C.M. Lott. Model-based testing of a highly programmable system. In *Proceedings of 9th International Symposium in Software Engineering (ISSRE'98) 1998, Paderborn, Germany, 4-7 November 1998*, pages 174–178. IEEE Computer Society Press, 1998.

[18] Y. Lei and K.C. Tai. In-parameter-order: A test generation strategy for pair-wise testing. In *Proceedings of the third IEEE High Assurance Systems Engineering Symposium*, pages 254–261. IEEE, November 1998.

[19] S.R. Dalal, A. Jain, G.C. Patton, M. Rathi, and P. Seymour. AETG web: A web based service for automatic efficient test generation from functional requirements. In *Proceedings of 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, USA, October 21-23, 1999*, pages 84–85. IEEE Computer Society, 1999.

[20] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. In *Proceedings of 21st International Conference on Software Engineering (ICSE'99) 1999, Los Angeles, CA, USA, 16-22 May 1999*, pages 285–294. ACM Press, 1999.

[21] D.M. Hoffman, P.A. Strooper, and L. White. Boundary Values and Automated Component Testing. *Journal of Software Testing, Verification, and Reliability*, 9(1):3–26, 1999.

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

[23] J. Huller. Reducing time to market with combinatorial design method testing. In *Proceedings of 10th Annual International Council on Systems Engineering (INCOSE'00) 2000, Minneapolis, MN, USA, 16-20 July 2000*, 2000.

[24] Y. Lei and K.C. Tai. A Test Generation Strategy for Pairwise Testing. Technical Report TR-2001-03, Department of Computer Science, North Carolina State University, Raleigh, 2001.

[25] A.W. Williams and R.L. Probert. A measure for component interaction test coverage. In *Proceedings of the ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001), Beirut, Lebanon, June 2001*, pages 304–311, June 2001.

[26] A.W. Williams and R.L. Probert. Formulation of the interaction test coverage problem as an integer problem. In *Proceedings of the 14th International Conference on the Testing of Communicating Systems (TestCom 2002), Berlin, Germany, March 2002*, pages 283–298, March 2002.

[27] K.C. Tai and Y. Lei. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, January 2002.

[28] N. Daley, D. Hoffman, and P. Strooper. A framework for table driven testing of Java classes. *Software - Practice and Experience (SP&E)*, 32:465–493, 2002.

[29] D.R. Kuhn and M.J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the 27th NASA/IEE Software Engineering Workshop, NASA Goodard Space Flight Center, MD, USA, 4-6 December, 2002*. NASA/IEEE, December 2002.

[30] T. Berling and P. Runesson. Efficient Evaluation of Multi-Factor Dependent System Performance using Fractional Factorial Design. *IEEE Transactions on Software Engineering*, 29(9):769–781, October 2003.

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

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

[33] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

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

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

[36] S. 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.

[37] 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.

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

[39] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, pages 438–444, July 1984.

[40] D. Hamlet and R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[41] W.J. Gutjahr. Partition Testing vs. Random Testing: The Influence of Uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, September/October 1999.

[42] G. Sherwood. Effective testing of factor combinations. In *Proceedings of the third International Conference on Software Testing, Analysis, and Review (STAR94), Washington DC*. Software Quality Eng., May 1994.

[43] 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.

[44] Y.K. Malaiya. Antirandom testing: Getting the most out of black-box testing. In *Proceedings of the International Symposium On Software Reliability Engineering, (ISSRE'95), Toulouse, France, Oct, 1995*, pages 86–95, October 1995.

[45] S. Rapps and E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11:367–375, april 1985.

[46] R. Biyani and P. Santhanam. TOFU: Test Optimizer of Functional Usage. Technical Report Software Engineering Technical Brief 2(1), IBM T.J. Watson Research Center, 1997.