SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2005; **15**:167–199
Published online 10 March 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.319

Combination testing strategies: a survey

Mats Grindal[1,*,†], Jeff Offutt[2] and Sten F. Andler[1]

[1]*School of Humanities and Informatics, University of Skövde, SE-541 28 Skövde, Sweden*
[2]*Information and Software Engineering Department, George Mason University, Fairfax, VA 22030, U.S.A.*

SUMMARY

Combination strategies are test case selection methods that identify test cases by combining values of the different test object input parameters based on some combinatorial strategy. This survey presents 16 different combination strategies, covering more than 40 papers that focus on one or several combination strategies. 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. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: combination strategies; category partition; orthogonal arrays; AETG; test case selection

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 16 different combination strategies described through more than 40 papers published between 1985 and 2004. This survey is an attempt to collect all this knowledge in one single place and form a basis for comparisons among the combination strategies.

Combination strategies are useful in many settings, not only in software testing. This survey only includes papers that explicitly mention software testing.

Section 2 gives a brief background on testing in general and on the Category Partition (CP) method [1] in particular. The CP method is a convenient place to start because it yields independent

*Correspondence to: Mats Grindal, School of Humanities and Informatics, University of Skövde, SE-541 28 Skövde, Sweden.
†E-mail: mats.grindal@his.se

Received 13 April 2004
Accepted 17 December 2004

values for parameters and spawned much of the work in combination strategies. Section 3 starts 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 the combination strategies have been structured into different groups based on different properties of the combination strategy algorithms. The overview of each algorithm includes an explanation of the algorithm and an example. Associated coverage criteria 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 the collected experiences of using combination strategies. Finally, Section 6 concludes this survey with a summary of the most important results and some future directions of research.

2. BACKGROUND

Testing, loosely considered to be the dynamic execution of test cases, consumes a significant amount of the resources in software projects [2,3]. Thus, it is of great importance to investigate ways of increasing the efficiency and effectiveness of testing [4–8].

Several existing testing methods (e.g. Equivalence Partitioning [2], CP [1], and Domain Testing [3]) 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 behaviour 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 [9] 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 favour 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 [10] later investigated the same question and concluded that the Duran and Ntafos model was unrealistic. One 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 than the information that defines the subdomains. Gutjahr [11] 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 favour partition testing over random testing in practical cases. Reid [6] and Yin *et al.* [12] 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.

1. Identify functional units that may be tested separately.
   For each functional unit, identify parameters and environment
   variables that affect the behaviour 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 CP method.

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 testing methods like Equivalence Partitioning (EP) [2] and Boundary Value Analysis (BVA) [2], parameters of the test problem are identified. Each parameter is then analysed 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 [1] proposed the CP method partially to address this problem.

2.1. Category partition

The CP method [1] 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 CP 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 for each one of the N parameters). Combination strategies can be used to decrease further the number of test cases in the test suite.

2.2. Classification trees

An important refinement to the CP method is that of *classification trees*, introduced by Grochtmann *et al.* [13,14]. Classification trees organize the information defined in Figure 1 into a tree structure. The first-level nodes are the parameters and environment variables (categories); they may be recursively broken into sub-categories. Choices appear as leaves in the tree and combinations are chosen by selecting among the leaves. A visual editor, the Classification-Tree Editor (CTE), is available [15].

The CP method and classification trees suggest an organization for testing. Defining categories, partitioning them into choices, picking values, and deciding which combinations of choices to use as tests has been the subject of a great deal of research in the past decade.

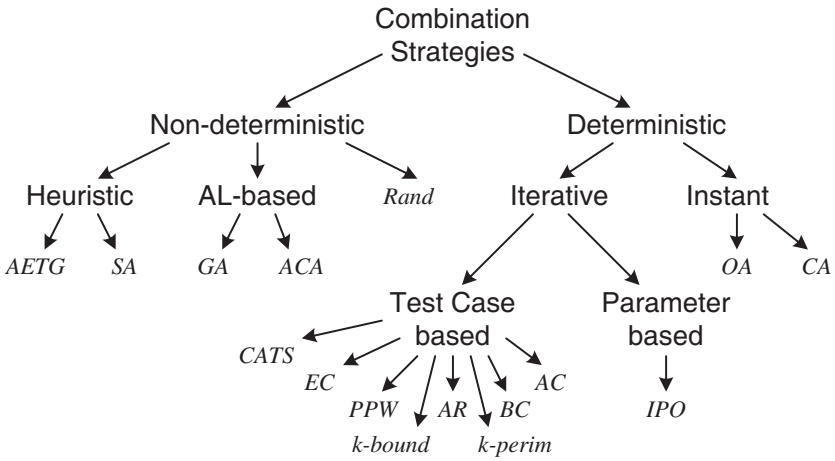

Figure 2. Classification scheme for combination strategies.

3. COMBINATION STRATEGIES: METHODS

Combination strategies form a class of test case selection methods where test cases are identified by choosing 'interesting' values[‡], and then combining those values of test object parameters. The combinations 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 labelled *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 includes two heuristic methods, *Simulated Annealing (SA)* and *Automatic Efficient Test Generator (AETG)*. Finally, two combination strategies are based on *Artificial Life (AL)* techniques, *Genetic Algorithm (GA)* and *Ant Crawl Algorithm (ACA)*.

[‡]The term 'interesting' may seem insufficiently precise and also a little judgmental. However, it appears frequently in the literature, so it 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.

The *deterministic* combination strategies group is further divided into two subgroups, *instant* and *iterative*. 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 iterative combination strategies build the test suite step by step. 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.

The largest group of combination strategies is *test case based*. They share the property that the algorithms generate one test case at a time and add it to the test suite. *Each Choice (EC)*, *Partly Pair-Wise (PPW)*, *Base Choice (BC)*, *All Combinations (AC)*, *Anti-random (AR)*, *CATS*, *k-perim*, and *k-bound* all belong to the test case based category.

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, each-used, does not take into account how interesting values of different parameters are combined, while the more complex coverage criteria, such as pair-wise coverage, are 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 is 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 [16].

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 that 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. However, semantic information may also be used when defining coverage criteria. Cohen *et al.* [17] indicate that normal 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.* [18].

A special case of t-wise coverage called *variable strength* was proposed by Cohen *et al.* [19]. This strategy requires higher coverage among a subset of parameters and a lower coverage criterion across all variables. Assume, for example, a test problem with four parameters A, B, C, D. Variable strength may require 2-wise coverage across all parameters and 3-wise coverage for parameters B, C, D.

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 are also 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 are valid.

A special case of normal values was used by Ammann and Offutt [20] to define base choice coverage. First, a base test case is identified by choosing an 'interesting value' of each parameter, such as the most frequently used value. The assumption is 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 are base values. Further, the test suite must also contain the base test case.

3.2. The test problem

The following sections describe a number of combination strategies. A common example is introduced here, and this example is used to illustrate the various strategies.

Consider a command '`series <start> <stop> [<step>]`', which generates a sequence of numbers that begin at `<start>` and end at `<stop>` with an optional `<step>`. For example, the command '`series 5 15 3`' will generate the sequence '5 8 11 14'.

The first step for the tester is to identify a set of parameters and interesting values to use as input to the combination strategies. Since the intent of this example is to illustrate the strategies, values are selected to be small yet illustrative, and thus may not be complete for testing.

The problem is modelled with four parameters $\langle A, B, C, D\rangle$. A is used for different starting values, B is used for different stopping values, C is used for the sign $(+/-)$ of the step, and D is used for different step values. Table I gives values for each parameter. The abstract test case $\langle 1, 2, 1, 3\rangle$ represents the first value for A, the second value for B, the first for C, and the third for D; thus it represents the command `series 2 15 +3`.

An empty cell, for example $B3$, means that no parameter value is defined for the testing problem. Whenever that value is included in a test case by a combination strategy it represents a 'free choice' and may be exchanged for an arbitrary defined value, in this case $B1$ or $B2$.

The 'abc' value of $A3$ represents an invalid value, that is, characters instead of integers. The 'do not use' value of $D1$ means that the step value should be omitted from the command.

The following sections use only the abstract representation of the test problem, that is, A1, B2, etc.

Table I. Mapping of parameters and parameter values for the series example. An empty cell means no existing value.

	Number		
Parameter	1	2	3
A	2	27	abc
B	8	15	
C	+	−	
D	do not use	1	3

```
Assume test cases t1–ti−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 t1–ti−1

A) Select candidates for ti by
     1) Selecting the variable and the value included in most pairs in UC.
     2) Put the rest of the variables into a random order.
     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 ti be the test case that covers
   the most pairs in UC. Remove those pairs from UC.

Repeat until UC is empty.
```

Figure 3. AETG heuristic algorithm for achieving pair-wise coverage.

3.3. 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.

3.3.1. Heuristic combination strategies

Heuristic *t*-wise (AETG). Burroughs *et al.* [21] and Cohen *et al.* [22] 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.* [17,23] 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 3 shows an algorithm that will generate a test suite to satisfy pair-wise coverage.

There are 36 possible combinations for the example in Section 3.2. Suppose that test cases {(A1, B2, C1, D2), (A2, B2, C1, D3), (A3, B1, C2, D1)} have already been selected. Then *UC* contains

the set of pairs {(A1, B1), (A1, C2), (A1, D1), (A1, D3), (A2, B1), (A2, C2), (A2, D1), (A2, D2), (A3, B2), (A3, C1), (A3, D2), (A3, D3), (B1, C1), (B1, D2), (B1, D3), (B2, C2), (B2, D1), (C1, D1), (C2, D2), (C2, D3)}.

When selecting candidates for the next test case, parameter values are evaluated to find which is included in the most pairs in *UC*. *B*1 and *C*2 each appear in five pairs. Suppose the algorithm selects *C*2, yielding the partial test case (_, _, C2, _). Then a random order of the remaining parameters is established. Assume the algorithm selects *D*, *A*, and *B*. Comparing the possible values of *D*, (_, _, C2, D1) does not cover any pairs in *UC*, while both (_, _, C2, D2) and (_, _, C2, D3) cover one pair. Assume the algorithm selects *D*2. Next is to compare the values of *A*. (A1, _, C2, D2) and (A3, _, C2, D2) each cover one additional pair in *UC* (only pairs including the currently studied parameter are counted). (A2, _, C2, D2), on the other hand, covers two pairs in *UC*. Thus, the algorithm selects (A2, _, C2, D2). Finally, the possible values of *B* are investigated. (A2, B1, C2, D2) covers two combinations in *UC*, while (A2, B2, C2, D2) only covers one. Thus, the algorithm has found a candidate, which in total covers five pairs in *UC*. The steps to identify candidates are repeated *k* times. Note, that there is some variance in the order of the remaining parameters and when two or more parameter values cover the same number of pairs of *UC*; in these cases an arbitrary choice is made. After generating enough candidates, the one that includes most pairs in *UC* is selected as the next test case and the search is restarted for the next test case.

The number of test cases generated by the AETG algorithm for a specific test problem is related to the number of constructed candidates (*k* in the algorithm in Figure 3) 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 dramatically decrease the number of test cases.

The AETG algorithm in Figure 3 generates a test suite that satisfies pair-wise coverage. It is straightforward to modify the algorithm to achieve *t*-wise coverage.

Simulated Annealing. Cohen *et al.* [19] suggest using Simulated Annealing (SA) to generate test suites for *t*-wise coverage. The algorithm was not described completely enough to include in this survey.

3.3.2. Artificial life based combination strategies

Genetic algorithm. A genetic algorithm (GA) was proposed by Shiba *et al.* [24]. The algorithm was inspired by AETG [23], and adds one test case at a time to a test suite until enough test cases have been identified to satisfy the desired coverage criterion. Figure 4 shows the genetic algorithm used to find the next test case.

A fitness function based on the number of new combinations covered by the test case candidate plays a crucial role in this algorithm. The fitness function is used both to identify the σ best individuals for survival to the next iteration (*Elite*) and in the selection mechanism to identify a reproduction base P_{mating}. P_{mating} is created by repeatedly selecting two test cases from P, and letting the fitter one win with some probability. Note that the same test case may be selected and included in P_{mating} several times. The contents of P_{mating} are then manipulated by uniform crossover and mutation. Crossover is achieved by letting two test case candidates from P_{mating} exchange values of each position independently with probability 0.5. Mutation replaces the value of a position in a test case with a randomly chosen value.


```
Input: A partial test set
Output: A test case
Begin
   Create an initial population P consisting of m candidates
   Evaluate P
   While (stopping condition is not met) {
       Identify Elite for survival consisting of σ best individuals from P.

       Apply Selection to individuals in P to create P_mating,
       consisting of (m − σ) individuals.

       Crossover P_mating
       Mutate P_mating

       P = Elite + P_mating

       Evaluate P.
       If (stagnation condition is met) Massively mutate P.
   }
   Return the best test case found.
End
```

Figure 4. Genetic algorithm for achieving t-wise coverage.

Should several iterations not result in any further improvements of the current solution, there is a possibility to 'massively mutate' P to avoid getting stuck at a local optimum. The algorithm terminates when the stopping condition is met, in this case, when M test case candidates have been generated. The best candidate among these is then returned.

To illustrate the GA, consider the mapping of the test problem in Section 3.2 presented in Table II. Assume that test cases {(110100), (010010), (101101), (101010)} have already been selected. Further assume that the population in the GA is of size 4, and that two test cases are selected to survive in each iteration.

Thus, four candidate test cases are generated randomly in the first step of the algorithm. Assume these candidates are {100101, 110000, 010100, 111010}. Each candidate is then evaluated using the fitness function (how many new pairs are covered by that candidate). In order, the fitness of the four candidates are {0, 1, 2, 0}. Thus, the *Elite* is {110000, 010100}. P_{mating} is then constructed in two steps. First a pair of candidates is selected randomly from P. Suppose the first two candidates are 100101 and 110000. A weighted choice favouring the best fit, 110000, is then made between the two, and suppose it is selected. Assume in the second iteration that the first and third candidates are selected. The weighted choice favours the third candidate, but assume for the sake of illustration that the first candidate, 100101, is selected. Thus P_{mating} initially consists of 110000 and 100101. In the crossover, b0 and b4 are selected for crossover, resulting in an updated P_{mating} of 100001 and 110100. In the mutation step, b1 of the first candidate is selected for mutation, resulting in the final P_{mating} of 100011 and 110100. The new content of P is then {100011, 110000, 010100, 110100}, with the fitness {3, 1, 2, 0}. In this manner the algorithm iterates until a stopping criterion is reached and the best test case

Table II. Encoding of the test problem from Section 3.2 into a bit vector.

Parameter	Bits	Value	Significance
A	$b1$, $b0$	00	A1 (2)
		01	A1 (2)
		10	A2 (27)
		11	A3 (abc)
B	$b2$	0	B1 (8)
		1	B2 (15)
C	$b3$	0	C1 (+)
		1	C2 (−)
D	$b5$, $b4$	00	D1 (do not use)
		01	D2 (1)
		10	D3 (3)
		11	D3 (3)

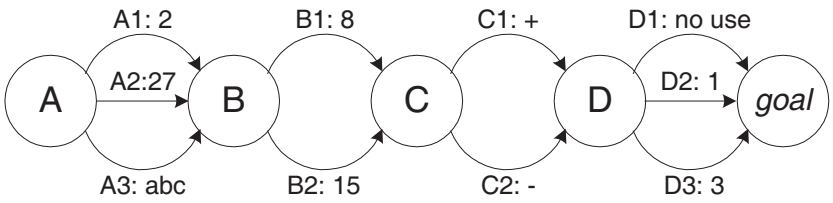

Figure 5. The test problem in Section 3.2 represented as a graph for the ACA.

candidate, the one with the highest fitness, is upgraded to a final test case and the algorithm starts over to find a new test case.

Genetic algorithms can be used to construct test suites to satisfy t-wise coverage for arbitrary t.

Ant Colony Algorithm. The Ant Colony Algorithm (ACA) was proposed by Shiba *et al.* [24]. It is based on an algorithm originally designed to solve the travelling salesman problem, and was inspired by the AETG algorithm. One test case at a time is added to a test suite until enough test cases have been identified to satisfy the desired coverage criterion.

The test case is represented as a graph with nodes representing the different parameters in the test problem and edges representing the different values of each parameter. Figure 5 shows the graph that represents the test problem described in Section 3.2. A test case is represented as a path from the starting node to the goal node.

The main idea of the algorithm is to let ants crawl the graph, guided by the amount of pheromone at each edge in the graph. When an ant has completed a path, the corresponding test case candidate is


```
Input: A partial test set
Output: A test case
Begin
   Compute local heuristics.
   Initialize pheromone.
   While (stopping condition is not met) {
       For (each ant k) {
           Generate a candidate test case S_k
           Evaluate S_k
           Lay pheromone
       }
       Update pheromone
       If (stagnation condition is met) Initialize pheromone.
   }
   Return the best test case found.
End
```

Figure 6. ACA for achieving t-wise coverage.

evaluated and more pheromone is deposited, along the way, proportional to the quality of the candidate. The complete algorithm can be seen in Figure 6. In the early iterations of the algorithm, not enough pheromone has been laid, therefore local heuristics are computed to guide the ants in these iterations.

To illustrate the algorithm, consider the example in Figure 5. Assume that test cases {(A1, B2, C1, D3), (A2, B1, C1, D2), (A1, B2, C2, D3), (A2, B1, C2, D3)} have already been selected, and local heuristics are based on the inverse of how many times each edge is already used in some test case. Thus, the initial pheromone is laid according to the following: {A1:2, A2:2, A3:4, B1:2, B2:2, C1:2, C2:2, D1:4, D2:3, D3:1}. Also assume that only one ant at a time crawls the graph.

Assume that the ant selects the path {A3, B1, C2, D1}, based on the initial pheromone weights. This candidate covers five uncovered pairs, which is a good result so a lot of pheromone is deposited on these edges. The resulting pheromone trail then is {A1:2, A2:2, A3:9, B1:7, B2:2, C1:2, C2:7, D1:9, D2:3, D3:1}. The next ant has a higher probability of crawling along the same path as the first. However, there is always an element of chance. Suppose that the next ant selects path {A3, B2, C1, D1}. This candidate also covers five uncovered pairs, thus a lot of pheromone is again deposited on the trail, resulting in {A1:2, A2:2, A3:14, B1:7, B2:7, C1:7, C2:7, D:14, D2:3, D3:1}. When the total number of candidates generated exceeds a predetermined number, the algorithm stops and returns the best candidate.

ACAs can be used to construct test suites to satisfy t-wise coverage for arbitrary t.

3.3.3. 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 a uniform distribution) is an old idea with unclear origins. In the scope of combination strategies, Mandl [25] 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.

3.4. 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 two subcategories: instant and iterative. Instant combination strategies create the complete test suite all at once. Iterative combination strategies build up the test suite iteratively. The test case based combination strategies add one test case at a time while the parameter-based combination strategies build up the test suite by adding values to the test cases for one parameter at a time.

3.4.1. Iterative combination strategies

Iterative combination strategies are those in which the test suite is built up gradually.

Test case based combination strategies. In the test case based iterative combination strategies, one test case at a time is generated and added to the test suite. Thus, a tester may start the algorithm with an already preselected set of test cases.

Each Choice. The basic idea behind the *Each Choice* (EC) 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 [20], who also suggested that EC can result in an undesirable test suite since the test cases are constructed mechanically without considering any semantic information.

Applying EC to the example described in Section 3.2 would yield the test cases {(A1, B1, C1, D1), (A2, B2, C2, D2), (A3, B1, C1, D3)}, where the values of B and C in the third test case are selected arbitrarily. It is clear from the definition of EC that it satisfies each-used coverage.

Partly Pair-Wise. Burroughs *et al.* [21] 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 the algorithm in Figure 7 is something of a guess.

Applying the Partly Pair-Wise (PPW) Algorithm in Figure 7 to the example in Section 3.2 yields the test suite {(A1, B1, C1, D1), (A1, B2, C2, D2), (A2, B1, C1, D3), (A2, B2, C1, D1), (A3, B1, C1, D1), (A3, B2, C1, D1)}, assuming that A and B are the two most significant parameters.

PPW satisfies variable strength coverage combining pair-wise and each-used coverage.

Base Choice. The *Base Choice* (BC) combination strategy was proposed by Ammann and Offutt [20]. 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.* called 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.


```
Assume a test problem with N parameters, P1, P2, ..., PN.
Let P1 and P2 represent the two most significant parameters.

A) Create all combinations of values of P1 and P2
B) For each parameter P3 ... PN
     Extend the combinations obtained in step A with values of the
     current parameter.
     If combinations remain when all parameter values have been used,
     reuse parameter values until all combinations have been extended.
     If there are not enough combinations for each parameter value to be used,
     reuse existing combinations until all parameter values have been used.
```

Figure 7. An instance of an algorithm implementing the partly pair-wise algorithm.

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 utilized 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.

Applying BC to the example in Section 3.2 requires a base test case to be identified: (A1, B2, C2, D3). The entire test suite can be derived from the base test case: {(A1, B2, C2, D3), (A2, B2, C2, D3), (A3, B2, C2, D3), (A1, B1, C2, D3), (A1, B2, C1, D3), (A1, B2, C2, D1), (A1, B2, C2, D2)}.

Ammann and Offutt state that satisfying BC 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 BC coverage *is* inadequate.

A slight variation of the BC combination strategy was later described by Cohen *et al.* [22]. In their version, called 'default testing', the tester varies the values of one parameter at a time while the other parameters contain *some* default value.

The term 'default testing' has also been used by Burr and Young [26], describing yet another variation of the general base choice theme. In their version all parameters except one contain the default value, and the remaining parameter contains a maximum or a minimum value. This variant does not necessarily satisfy each-used coverage.

All Combinations. 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 [25,20,17].

An AC test suite satisfies N-wise coverage.

Antirandom. Antirandom (AR) testing was proposed by Malaiya [27], 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 at maximum Hamming or Cartesian distance from all the other already selected test cases.

Table III. A second encoding of the test problem from Section 3.2 into a bit vector.

Parameter	Bits	Value	Significance
A	*b*1, *b*0	00	A1 (2)
		01	A1 (2)
		10	A2 (27)
		11	A3 (abc)
B	*b*2	0	B1 (8)
		1	B2 (15)
C	*b*3	0	C1 (+)
		1	C2 (−)
D	*b*5, *b*4	00	D1 (do not use)
		01	D2 (1)
		10	D3 (3)
		11	— (5)

To illustrate the concept of AR and the difference between Hamming and Cartesian distance, consider the test problem described in Section 3.2. The first step is to encode the values of the test object into a binary vector. Parameters *B* and *C* have two variables each, thus one single bit per parameter is sufficient for the encoding. Parameters *A* and *D* have three variables each. For these parameters at least two bits per parameter are needed. Using two bits for a parameter provides four different values. Thus, the tester may choose to add another value or let two values of the two-bit sequence represent the same parameter value. Table III shows an example of a complete mapping, in which one value of parameter *A* is used twice and a new value for parameter *D* has been added to illustrate both concepts.

Suppose the bit vector (001010), corresponding to (A2, B1, C2, D1), has been selected as the starting point for AR in the example. The AR algorithm is now invoked to select the second test case. Either the Hamming distance or the Cartesian distance is used to find a test case as far away as possible from the starting test case. The Hamming distance is the number of bits in which two binary vectors differ. The Cartesian distance is defined as the square root of the Hamming distance. In the example, it is straightforward to see that the second bit vector selected by the AR algorithm is the bit-inverse, (110101). The Hamming distance between the two vectors is 6 and the Cartesian distance is $\sqrt{6}$.

The third bit vector is selected such that the sum of the distances to the first two is maximized. If Hamming distance is used, the sum of the distances to the two selected bit vectors is 6 for any bit vector, i.e. any bit vector may be selected as the third. The reason is that each bit in the third bit vector differs from exactly one of the already selected bit vectors. Thus, the sum of the distances is either $1 + 5$, $2 + 4$, $3 + 3$, $4 + 2$, or $5 + 1$. If, on the other hand, Cartesian distance is used, only bit vectors with exactly half of the bits coming from each already selected vector are candidates to be selected as the third bit vector. The reason is that $\sqrt{3} + \sqrt{3}$ is larger than both $\sqrt{2} + \sqrt{4}$ and $\sqrt{1} + \sqrt{5}$.

When selecting the fourth bit vector, the sum of the distances to all of the three previously selected vectors is considered and so on.

In the original description, the AR testing scheme is used to create an ordering among all the possible test cases rather than attempting to limit the number of test cases. However, just as it is possible in random testing to set a limit on how many test cases should be generated, the same applies to AR test cases. Thus, AR testing can be used to select a subset of all possible test cases, while ensuring that they are as far apart as possible.

In the general case, a test suite generated by AR does not satisfy any of the normal coverage criteria associated with combination strategies.

***k*-boundary (*k*-bound) and *k*-perimeter (*k*-perim).** With a single parameter restricted to values $[L, \ldots, U]$ (ordered from L(ower) to U(pper)), generating boundary values is easy. The tester can select the set $\{L, U\}$ or $\{L, L+1, U-1, U\}$ or $\{L, L+1, L+2, U-2, U-1, U\}$ etc. With multiple domains, two problems arise: (1) the number of points grows rapidly and (2) there are several possible interpretations. Hoffman *et al.* [28] defined the k-boundary and k-perimeter combination strategies, where k is an arbitrary integer, 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 ($L+1$ and $U-1$) and so on.

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 $(U,\ L,\ L,\ L)$; and (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. $\{(L+1,\ L,\ L,\ L), (L+2,\ L,\ L,\ L), \ldots, (U-2,\ L,\ L,\ L), (U-1,\ L,\ L,\ L)\}$.

The example used to illustrate the majority of the other combination strategies is not well suited to illustrate k-bound and k-perim due to the small number of parameter values for some of the parameters. Thus, consider a test problem with two parameters A and B. Suppose A has seven parameter values {A1, A2, A3, A4, A5, A6, A7} and B has four parameter values {B1, B2, B3, B4}. Also assume that the parameter values are ordered from smallest to largest. The following test suites satisfy 100% 1-bound and 2-bound coverage respectively: {(A1, B1), (A1, B4), (A7, B4), (A7, B1)} and {(A2, B2), (A2, B3), (A6, B3), (A6, B2)}. 3-bound and higher is not relevant in this example since parameter B only contains four values.

Adding all the intermediate test cases to the test suite satisfying 100% 1-bound coverage yields 100% 1-perim coverage: {(A1, B1), (A1, B2), (A1, B3), (A1, B4), (A2, B4), (A3, B4), (A4, B4), (A5, B4), (A6, B4), (A7, B4), (A7, B3), (A7, B2), (A7, B1), (A6, B1), (A5, B1), (A4, B1), (A3, B1), (A2, B1)}. In the same manner a test suite for 100% 2-perim coverage can be created: {(A2, B2), (A2, B3), (A3, B3), (A4, B3), (A5, B3), (A6, B3), (A6, B2), (A5, B2), (A4, B2), (A3, B2)}.

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

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 [29] and a version of the algorithm that generates a test suite to satisfy pair-wise coverage is shown in Figure 8.

```
Assume test cases t1–ti−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 t1–ti−1

A) Select ti by finding the combination that covers most pairs in UC.
   If more than one combination 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 8. CATS heuristic algorithm for achieving pair-wise coverage.

There are 36 possible combinations for the example in Section 3.2. Suppose that test cases {(A1, B2, C1, D2), (A2, B2, C1, D3), (A3, B1, C2, D1)} have already been selected. Then Q contains the remaining 33 combinations. UC contains the set of pairs {(A1, B1), (A1, C2), (A1, D1), (A1, D3), (A2, B1), (A2, C2), (A2, D1), (A2, D2), (A3, B2), (A3, C1), (A3, D2), (A3, D3), (B1, C1), (B1, D2), (B1, D3), (B2, C2), (B2, D1), (C1, D1), (C2, D2), (C2, D3)}.

Among the combinations in Q there are two, (A1, B1, C2, D3) and (A2, B1, C2, D2), that contain five pairs still in UC, which is the current maximum. Thus, the algorithm selects (A1, B1, C2, D3) as the next test case and it is removed from Q. The new contents of UC are then {(A1, D1), (A2, B1), (A2, C2), (A2, D1), (A2, D2), (A3, B2), (A3, C1), (A3, D2), (A3, D3), (B1, C1), (B1, D2), (B2, C2), (B2, D1), (C1, D1), (C2, D2)}. In the next iteration, $(A2,\ B1,\ C2,\ D2)$ is the only test case in UC that contains five pairs. Thus, it is selected and the resulting contents of UC are now {(A1, D1), (A2, D1), (A3, B2), (A3, C1), (A3, D2), (A3, D3), (B1, C1), (B2, C2), (B2, D1), (C1, D1)}. Of the remaining 31 combinations in Q, only one still in UC, (A3, B2, C1, D1), contains four pairs. The process repeats until UC is empty.

The resulting set of test cases in this example is {(A1, B2, C1, D2), (A2, B2, C1, D3), (A3, B1, C2, D1), (A1, B1, C2, D3), (A2, B1, C2, D2), (A3, B2, C1, D1), (A1, B1, C1, D1), (A2, B2, C2, D1), (A3, B1, C1, D2), (A3, B1, C1, D3)}.

The CATS algorithm has some similarities with AETG. The main difference is in how the next test case to be included in the final test suite is found. CATS examines the whole list of unused test cases to find one that adds as much new coverage as possible while AETG constructs test case candidates one parameter value at a time based on coverage information. The constructed candidates are then evaluated to find the best possible, which is added to the test suite.

In CATS it is guaranteed that the 'best' test case is always selected while in AETG there are no guarantees. However, for large test problems, AETG is more efficient since only a small set of test cases has to be evaluated in each step.

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


```
Algorithm IPO_H (τ, 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
  π = { pairs between values of p_i and values of p_1 to p_{i-1}}
  if (|τ| ≤ q)
  {
    for 1 ≤ j ≤ |τ|, extend the jth test in τ by adding value v_j and
    remove from π pairs covered by the extended test.
  }
  else
  {
    for 1 ≤ j ≤ 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 ≤ |τ|, 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.

Parameter-based combination strategies. There is only one parameter-based combination strategy, *In Parameter Order (IPO)*. Just as with test case based combination strategies, a tester may start the IPO algorithm with an already preselected set of test cases.

In Parameter Order. For a system with two or more parameters, the IPO combination strategy [30–32] 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, is shown in Figure 9. It extends the existing test cases in the test suite with values of the next parameter. The second algorithm, vertical growth, shown in Figure 10, creates additional test cases such that the test suite satisfies pair-wise coverage for the values of the new parameter.

To illustrate how the IPO algorithm works, consider the test problem described in Section 3.2. The starting point for IPO is a test suite that satisfies pair-wise coverage for a subset of the parameters of the test problem. Consider parameters A and B. A test suite that satisfies pair-wise coverage for these parameters is {(A1, B1), (A1, B2), (A2, B1), (A2, B2), (A3, B1), (A3, B2)}. Then IPO_H is invoked to extend the existing test cases with the values of parameter C.

When invoking IPO_H, τ contains the test suite and π contains all uncovered pairs including the values of C, {(A1, C1), (A1, C2), (A2, C1), (A2, C2), (A3, C1), (A3, C2), (B1, C1), (B1, C2), (B2, C1), (B2, C2)}. The first step is to add the first value of C to the first test case in τ, the second value of C

```
Algorithm IPO_V (τ, π)
{
  Where τ contains the set of already selected test cases
  and π contains the set of still uncovered pairs
  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 as 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.

to the second test case in τ, etc., until each value of C has been used once. Thus, τ = {(A1, B1, C1), (A1, B2, C2), (A2, B1, _), (A2, B2, _), (A3, B1, _), (A3, B2, _)}. Covered pairs are removed from π resulting in {(A2, C1), (A2, C2), (A3, C1), (A3, C2), (B1, C2), (B2, C1)}.

The next step of IPO_H is to extend the remaining test cases in τ with a value of C to cover as many pairs as possible in π. Thus, for the third test case, (A2, B1, C1) covers one pair in τ while (A2, B1, C2) covers two, hence the latter is selected and the covered pairs are removed from π. Repeating these steps for all test cases in τ yields τ = {(A1, B1, C1), (A1, B2, C2), (A2, B1, C2), (A2, B2, C1), (A3, B1, C1), (A3, B2, C2)} and π = {}. Since π is empty, τ satisfies pair-wise coverage also for the values of C, which means that IPO_V does not need to be invoked.

In the next iteration, parameter D is considered. Again, IPO_H is invoked, this time with τ = {(A1, B1, C1), (A1, B2, C2), (A2, B1, C2), (A2, B2, C1), (A3, B1, C1), (A3, B2, C2)} and π = {(A1, D1), (A1, D2), (A1, D3), (A2, D1), (A2, D2), (A2, D3), (A3, D1), (A3, D2), (A3, D3), (B1, D1), (B1, D2), (B1, D3), (B2, D1), (B2, D2), (B2, D3), (C1, D1), (C1, D2), (C1, D3), (C2, D1), (C2, D2), (C2, D3)}.

Adding the three values of D to the three first test cases of τ results in τ = {(A1, B1, C1, D1), (A1, B2, C2, D2), (A2, B1, C2, D3), (A2, B2, C1, _), (A3, B1, C1, _), (A3, B2, C2, _)} and π = {(A1, D3), (A2, D1), (A2, D2), (A3, D1), (A3, D2), (A3, D3), (B1, D2), (B2, D1), (B2, D3), (C1, D2), (C1, D3), (C2, D1)}. Extending the remaining three test cases of τ with values of D such that as many pairs in π are covered and removed, results in τ = {(A1, B1, C1, D1), (A1, B2, C2, D2), (A2, B1, C2, D3), (A2, B2, C1, D1), (A3, B1, C1, D2), (A3, B2, C2, D1)} and π = {(A2, D2), (A3, D1), (A3, D3), (B2, D3), (C1, D3)}.

Since π is non-empty there are still some pairs involving parameter values of D that are not covered by the test suite. Thus, IPO_V is invoked with τ and π as above. IPO_V iterates over the pairs of π. In the first iteration, τ' is empty, which results in a new partial test case being added to τ' such that

1	2	3
2	3	1
3	1	2

Figure 11. A 3 × 3 Latin Square.

1	2	3
2	3	1
3	1	2

1	2	3
3	1	2
2	3	1

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

Figure 12. Two orthogonal 3 × 3 Latin Squares and the resulting combined square.

the current pair of π is covered, (A2, _, _, D2). The next pair of π, (A3, D1), is incompatible with the first test case in τ', thus a new incomplete test case is added, with the result that $\tau' = \{$(A2, _, _, D2), (A3, _, _, D1)$\}$. In the third iteration, τ' is further extended with (A3, _, _, D3). In the fourth iteration, the current pair (B2, D3) can be fitted into the third test case of τ', changing it to (A3, B2, _, D3). Finally, the last pair of π is incompatible with the existing three, resulting in a fourth incomplete test case being added to τ', with the result $\{$(A2, _, _, D2), (A3, _, _, D1), (A3, B2, _, D3), (_, _, C1, D3)$\}$.

In the last step of the IPO_V algorithm the test cases of τ' are added to the already existing test cases of τ, resulting in a test suite that satisfies pair-wise coverage for all four parameters. The undecided parts of the incomplete test cases in the final test suite need to be instantiated at execution time, but in terms of pair-wise coverage any value can be used, so it is up to the tester.

The nature of the IPO algorithms makes it difficult to calculate theoretically the number of test cases in a test suite generated by the algorithm.

An algorithm closely resembling IPO was informally described by Huller [33].

3.4.2. Instant combination strategies

Sometimes a tester wants to include favourite 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 of whether some other test cases have already been preselected.

Orthogonal Arrays. *Orthogonal Arrays* (OAs) is a mathematical concept that has been known for quite some time. The application of OAs to testing was first introduced by Mandl [25] and later more thoroughly described by Williams and Probert [34].

The foundation of OAs is Latin Squares. A *Latin Square* is a $V \times V$ 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 11 contains an example of a 3 × 3 Latin Square with the symbols $\{1, 2, 3\}$.

Table IV. Two orthogonal 3 × 3 Latin Squares augmented with coordinates.

	Columns (c)		
Rows (r)	1	2	3
1	1, 1	2, 2	3, 3
2	2, 3	3, 1	1, 2
3	3, 2	1, 3	2, 1

Table V. Tuples from the two orthogonal 3 × 3 Latin Squares that satisfy pair-wise coverage.

Tuple	Tuple $\langle r, c, z_1, z_2 \rangle$
1	1111
2	1222
3	1333
4	2123
5	2231
6	2312
7	3132
8	3213
9	3321

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 12 shows an example of two orthogonal 3 × 3 Latin Squares and the resulting combined square.

If indexes are added to the rows (r) and the columns (c) of the matrix, each position in the matrix can be described as a tuple $\langle r, c, z_i \rangle$, where z_i represents the values of the $\langle r, c \rangle$ position. Table IV contains the indexed Latin Square from Figure 12 and Table V contains the resulting set of tuples. The set of all tuples constructed by a Latin Square satisfies pair-wise coverage.

The set of test cases in Table V can be used directly for the example described in Section 3.2 by mapping the dimensions of the tuple $\langle r, c, z_1, z_2 \rangle$ onto the parameters A, B, C, and D. Recall that parameters A and D contain three values each, while parameters B and C only contain two values each. Assuming a mapping that preserves the stated order, that is, that maps A onto r, B onto c, C onto z_1, and D onto z_2, test cases 3, 5, 6, 7, 9 all contain coordinate values without corresponding parameter values. This problem is resolved simply by substituting the undefined coordinate value with an arbitrary defined value. Consider, for example, test case 7 in Table V. The value of coordinate z_1 is 3 while only values 1 and 2 are defined in the mapping from 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.

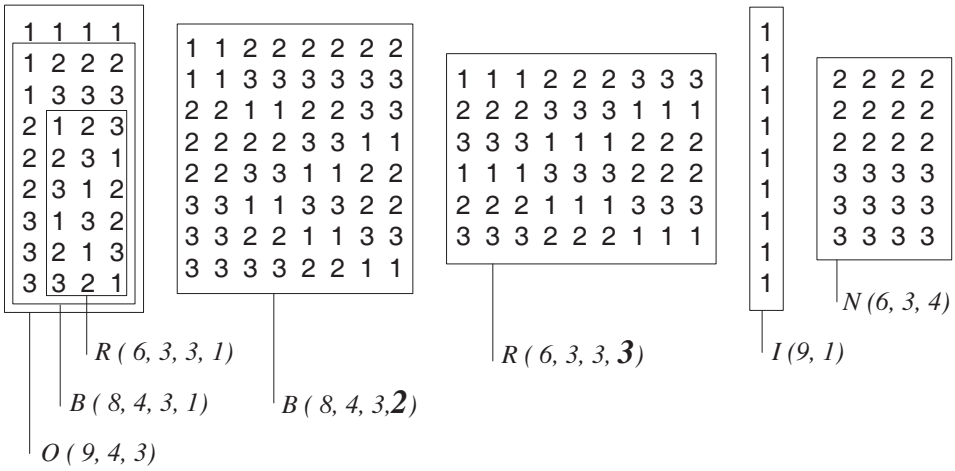

Figure 13. Examples of the five types of building blocks for generation of covering arrays.

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. Williams and Probert [34] 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[§].

Covering Arrays. Covering Arrays (CAs) [35] are an extension of OAs. A property of OAs is that they are *balanced*, which means that each parameter value and every subcombination of values occur the same number of times in the test suite. If only t-wise (for instance pair-wise) coverage is desired the balanced property is unnecessary and will make the algorithm less efficient. In a CA that satisfies t-wise coverage, each t-tuple occurs at least once but not necessarily the same number of times. Another problem with OAs is that for some problem sizes there do not exist enough OAs to represent the entire problem. This problem is also avoided by using CAs.

A CA is denoted by $C(r,\ c,\ k)$, where r is the number of rows (test cases), c is the number of columns (parameters), and k is the maximum number of values of any parameter. A CA is constructed for a test problem from a set of building blocks.

- $O(r,\ c,\ k)$ is an OA based test suite with r rows (test cases) and c columns (parameters), which have at most k values each. In addition, for this OA to be useful, the OA must be constructed using Bose's method [35] and r, c, and k should be related by the formulas $r = k^2$ and $c = k+1$.

[§]URL: http://www.research.att.com/~njas/oadir (page last accessed November 2004).

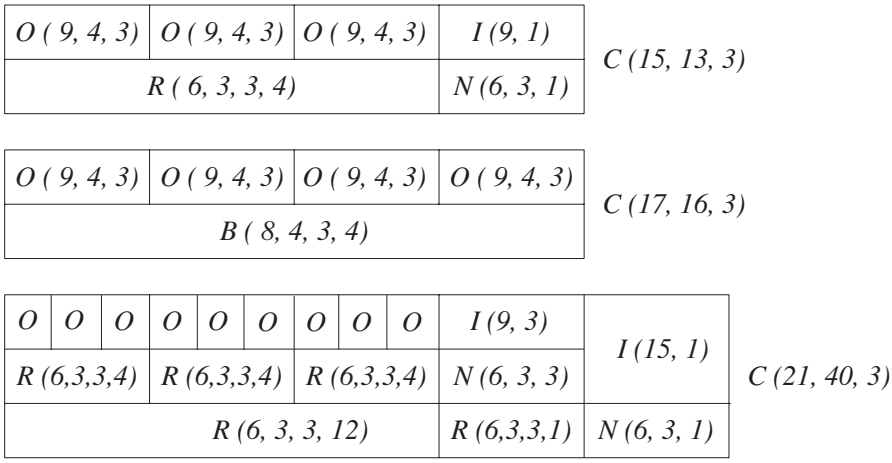

Figure 14. Examples of how different building blocks may be combined to form covering arrays. O is short for $O(9, 4, 3)$.

- $B(r-1,\ c,\ k,\ d) = O(r,\ c,\ k)$ with the first row removed and with the columns used d times each consecutively.
- $R(r-k,\ c-1,\ k,\ d) = O(r,\ c,\ k)$ with the first k rows and the first column removed and with the remaining columns used d times each consecutively.
- $I(r,\ c)$ is a matrix with r rows and c columns completely filled with 'ones'.
- $N(r,\ k,\ c)$ is a matrix with r rows and c columns, which consists of $k-1$ submatrices of size k rows $\times$ c columns filled with 'twos', 'threes', etc. up to k.

Figure 13 gives examples of the five types of building blocks.

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. Figure 14 shows some examples of how building blocks may be combined. The algorithm to identify and assemble suitable building blocks for a given problem size is omitted from this presentation for reasons of size but may be found in Williams' paper [35]. Both the algorithms for finding suitable building blocks and for combining these to fit a specific test problem have been automated.

An extension of the work on CAs was given by Williams and Probert [36]. A similar approach, also building on the concept of building blocks to construct a covering array, was proposed by Kobayashi *et al.* [37].

4. COMPARING COMBINATION STRATEGIES

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

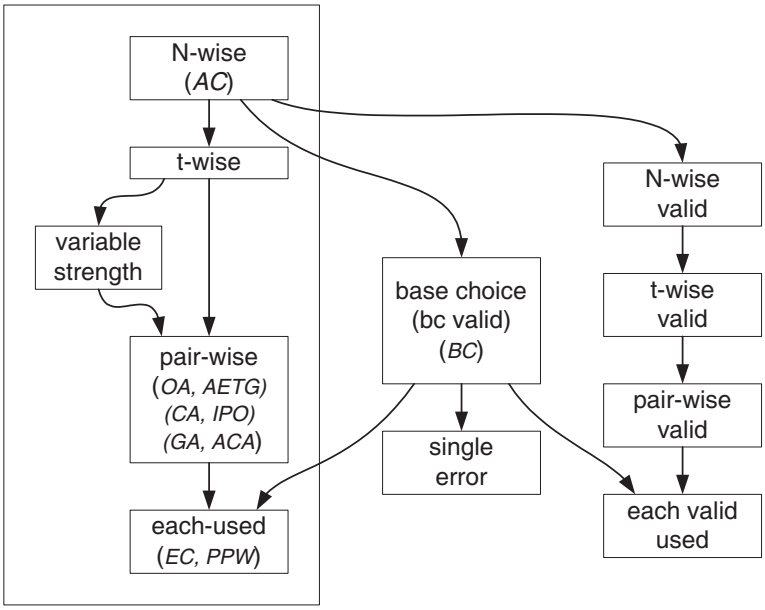

Figure 15. Subsumption hierarchy for the coverage criteria related to combination strategies. Combination strategies that support a certain coverage criterion are indicated by italics.

4.1. Subsumption

A subsumption hierarchy for the combination coverage criteria reported in the literature is shown in Figure 15. The definition of subsumption is from Rapps and Weyuker [38]: coverage criterion X *subsumes* coverage criterion Y if and only if 100% X coverage implies 100% Y coverage (Rapps and Weyuker's original paper used the term 'inclusion' instead of 'subsumption'). A guiding principle for this work has been to generalize the known coverage criteria to obtain a complete picture.

The left, boxed-in, column of coverage criteria in Figure 15 represents criteria that do not use semantic information. N-wise coverage requires that every combination of all parameter values are included in the test suite. The term '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 is 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 counts if the remaining parameters are also 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 as was described in Section 3.1, so it is omitted. At first it may seem that the single error coverage criterion would subsume the each-used coverage criterion. However, a counterexample 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 BC algorithm it follows that all test cases differ from the base test case by the value of only one parameter. Further, 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 VI gives an overview of the sizes of the generated test suites. In some cases it is only possible to give approximate values. The main reason for this is that those algorithms contain some degree of randomness. Thus, the same combination strategy may produce different solutions to the same test problem. Also, only a subset of the described combination strategies is included in the table. There are several reasons. Some strategies, i.e. Rand and AR, 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 greediness, 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 VI contains two example test problems: one with many parameters and few values and one with few parameters and many values.

5. COMBINATION STRATEGIES: EXPERIENCE AND EVALUATIONS

This section gives a brief overview of the collected experience of using combination strategies. First some theoretical results are described, then a brief overview of implemented tools is given. The most important real-life applications of combination strategies are then presented and finally some results from comparing different combination strategies are given.

Table VI. Definite or approximate (∼) test suite sizes for a problem with N parameters and V_i values of the ith parameter. $V_{\max}$ is the largest V_i.

		Example 1	Example 2
Combination strategy	Test suite size	$N = 8, V_i = 4$	$N = 4, V_i = 8$
EC	$V_{\max}$	4	8
BC	$1 + \sum_{i=1}^{N} (V_i - 1)$	25	29
pair-wise (IPO, AETG, OA, CA GA, ACA, SA)	$\sim V_{\max}^2$	∼16	∼64
AC	$\prod_{i=1}^{N} V_i$	65 536	4096

5.1. Applicability of combination strategies: theory

Heller [39] used a realistic example to show that testing all combinations of parameter values is infeasible in practice. Heller concluded that there is a need to identify a subset of combinations of manageable size. The author suggested fractional factorial designs, which are closely related to the OA combination strategies. Traditionally, these have been used in natural science experiments to decrease the number of experiments. The underlying mathematics of the fractional factorial designs allows researchers to establish which factors cause which results. Furthermore, the number of test cases from fractional factorial designs has been shown to be less than all combinations.

Dalal and Mallows [40] provided a theoretical view of combination strategies. They gave 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.

Following the lead of Dalal and Mallows, Kuhn and Reilly [41] gave a strong argument for the use of combination strategies. They investigated 365 error reports from two large real-life projects and discovered that pair-wise coverage was nearly as effective in finding faults as testing all combinations, since most faults were one- or two-factor faults. Their findings were strengthened by Kuhn *et al.* [42] in another study of the relationship between input and output values in a number of real applications.

Dunietz *et al.* [43] examined the correlation between t-wise coverage and achieved code coverage. A number of test suites for a specific test problem were created by random sampling. For every test suite, t-wise coverage with $t = 1, 2, \ldots, n$ was calculated. Then the test suites were executed and block (statement) and path coverage were computed. The result of this study is that test suites satisfying t-wise coverage, where $t \geq 2$, all perform similarly in terms of block (statement) coverage. Thus, the authors provided an argument for using pair-wise coverage instead of all-combination coverage.

Piwowarski *et al.* [44] described how to apply code coverage successfully 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. Burr and

Assume a test problem with three parameters a, b, and c, each with two 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 both in $tc1$ and $tc3$.

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

2-coverage = 8/12 = 67%
2-diversity = 8/9 = 89%

Figure 16. An example of 2-coverage and 2-diversity of a test suite.

Young [26] showed that continually monitoring achieved code coverage works as a tool to improve the input parameter model. Initial experiments showed that *ad hoc* testing resulted in about 50% block and decision coverage. By continually applying code coverage as the input parameter models were refined, decision coverage was increased to 84%.

Dalal *et al.* [45] presented an 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. Test cases, containing test inputs and expected results, are generated by the combination strategy from the test data model. The test inputs are fed to the system under test, which generates actual output. Finally the actual and expected outputs are compared.

The authors claimed 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.

Dalal and Mallows [40] provided two measures of efficiency of a combination strategy test suite. The t-wise coverage (in Dalal and Mallows' 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 16 shows an example of these two metrics.

Williams and Probert [16] formally defined a set of coverage criteria, including t-wise coverage, through the use of a so-called *interaction element*, which may be used in the same way statements and decisions are used for code 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.

In the seminal work on AR testing, Malaiya [27] observed that for some combination strategies, the order in which test cases are generated may help preserve execution. In the case of AR testing, the algorithm always finds the next test case as far away as possible from the already existing test cases. If that order is preserved during execution, and if the execution is prematurely terminated, the executed test cases will always be optimally spread across the input space.

5.2. Tools for combination strategies

Tools have been implemented for some combination strategies, although only one is available commercially. T-GEN [46] is a test case generator implemented for the CP method [1]. T-GEN may also be used to generate test cases based on the output space in a similar manner to the way the input space is handled in CP. Also, T-GEN can generate complete or partial test scripts under certain conditions.

The Orthogonal Arrays Testing System (OATS) tool [47] is based on the OA combination strategy. OATS automatically identifies OAs and generates test suites that satisfy pair-wise coverage of the values of the parameter values.

PairTest [31] is a tool implemented in Java. It implements the IPO combination strategy for pair-wise coverage.

Telcordia (formerly Bellcore) has a test tool called AETG [48]. It implements the heuristic AETG combination strategy. This tool has been used for both research and practice since AETG is also commercially available on the Web[¶].

Daley *et al.* [49] described the 'Roast' framework. This approach differs from the previously described tools by attempting to automate both the generation and execution of test cases. The Roast framework 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 supplying a parameter model of the test problem, the only manual intervention is adding the expected output.

5.3. Applicability of combination strategies: practice

Plenty of examples of the applicability of combination strategies in software testing can be found in the literature. Within the scope of functional testing, Dalal *et al.* [45,50] reported results from using the AETG tool. The tool was used to generate test cases for Bellcore's Intelligent Service Control Point, a rule-based system used to assign work requests to technicians, and a Graphical User Interface (GUI) window in a large application. In a previous study, Cohen *et al.* [22] demonstrated the use of AETG for screen testing, by testing the input fields for consistency and validity across a number of screens. Dalal *et al.* [50] 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 requires a number of test artifacts that must 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.

Burr and Young [26] also used the AETG tool. They tested a third party email product from Nortel that converts email messages from one format to another. Another example of using combination

[¶]http://aetgweb.agreenhouse.com (page last accessed November 2004).

strategies for functional testing is described by Huller [33], who used an IPO-related algorithm to test ground systems for satellite communications.

Closely related to functional testing is the use of the k-bound and k-perim combination strategies within the Roast framework [49] to semi-automatically test Java classes.

Although the examples of using combination strategies for functional testing dominate, there are several examples of the applicability of combination strategies in other areas of testing. Williams and Probert [34] demonstrated how combination strategies can be used to select configurations for configuration testing. The authors pointed out that the cost of 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. Thus, it is important to find ways to identify a small number of configurations to test. In this case OAs were successfully used. A related attempt was carried out by Yilmaz *et al.* [51]. They used CAs as a starting point for fault localization in complex configuration spaces. When a fault of unclear origin occurs, CAs are used to select a set of configurations. The same test cases are then executed on all configurations and the results are analysed, to localize the origin of the fault.

Kropp *et al.* [52] described the Ballista testing methodology, which supports automated robustness testing of off-the-shelf software components. Robustness is defined as the degree to which a software component functions correctly in the presence of exceptional inputs or stressful conditions. Ballista generates a test suite that has all combinations of all parameter values. If the total number of combinations is too large to be feasible, 5000 in the reported experiment, a deterministic pseudo-random sample is used. The application of the Ballista methodology is demonstrated on several implementations of the POSIX operating system C language application programming interface (API).

Berling and Runesson [53] described how to use fractional factorial designs for performance testing. Specifically, they focused on how to find a minimal or near minimal set of test cases to determine the effect on the system performance of different parameter values. An important observation made by Berling and Runesson is that when the result variable has a nominal type (for example, true or false, or pass or fail as in testing), the property of fractional factorial designs cannot be exploited.

5.4. Comparisons of combination strategies

Several papers have reported results from comparing combination strategies. This section contains brief descriptions and conclusions of these studies.

Huller [33] showed that pair-wise configuration testing may save more than 60% in both cost and time compared to quasi-exhaustive testing. A similar conclusion was reached by Brownlie *et al.* [47] who compared the results of using OAs on one version of a PMX/StarMAIL release with the results from conventional testing on a prior release. The authors estimated that 22% more faults would have been found if OAs had been used on the first version, despite the fact that conventional testing required twice as much time to perform.

Several studies have compared the performance of different combination strategies. By far the most popular property used to compare combination strategies is number of test cases generated for a specific test problem. In particular, this is interesting for the non-deterministic and greedy combination strategies, AETG, SA, GA, and ACA, since the size of the test suite may not be determined algebraically. A number of papers have compared a subset of the combination strategies satisfying 2-wise or 3-wise coverage: IPO and AETG [30], OAs and AETG [18], CAs and IPO [35], and AETG,

IPO, SA, GA, and ACA [19,24]. Interesting to note is that in all these comparisons, these combination strategies perform similarly with respect to the number of test cases. SA seems to be generating the least number of test cases most of the time, but the differences are usually small.

Since the performance with respect to generated test cases does not largely favour one particular combination strategy, several of the authors have also compared the strategies with respect to the execution time for the combination strategy to generate its test cases. Lei and Tai [31] showed that the time complexity of IPO is superior to the time complexity of AETG. IPO has a time complexity of $O(v^3N^2\log(N))$ and AETG has a time complexity of $O(v^4N^2\log(N))$, where N is the number of parameters, each of which has v values. Further, Williams [35] reported that CAs outperform IPO by almost three orders of magnitude for the largest test problems in their study, in terms of time taken to generate the test suites. Finally, Shiba *et al.* [24] showed some execution times, but the executions have been made on different target machines so the results are a bit inconclusive. However, SA, SGA and ACA seem to perform similarly, more than an order of magnitude faster than AETG.

Grindal *et al.* [18] took another approach to compare combination strategies, the number of faults found. They showed that BC performs as well, in terms of detecting faults, as AETG and OAs, despite fewer test cases for a number of test objects seeded with 131 faults. However, a similar fault detection ratio does not necessarily mean that the combination strategies detect the same faults. The authors showed that BC, OAs and AETG target different types of faults.

Yet another way to evaluate combination strategies is on the basis of achieved code coverage of the generated test suites. This approach was taken by Cohen *et al.* [23]. They reported on test suites generated by AETG for 2-wise coverage that reached over 90% block coverage. Burr and Young [26] obtained similar results for AETG in another study. AETG reached 93% block coverage with 47 test cases, compared with 85% block coverage for a restricted version of BC using 72 test cases.

Yin *et al.* [12] also used code coverage to compare AR with randomly selected test cases. The AR combination strategy consistently performed well by reaching high coverage with few test cases.

6. CONCLUSIONS AND FUTURE DIRECTIONS

This survey is an attempt to collect and describe in a comprehensive manner some of the properties of combination strategies for test generation. In particular, it gives an overview of the coverage criteria normally associated with combination strategies and it shows how these coverage criteria are interrelated in a subsumption hierarchy. This survey also relates the surveyed combination strategies based on the approximate sizes of the test suites generated by the combination strategies. It reports on the usefulness of combination strategies and it also suggests a classification scheme for the existing combination strategies based on properties of their algorithms.

Despite the many papers covering combination strategies, several important issues still remain largely unexplored. One such issue is the identification of suitable parameters and parameter values. At first glance, identifying the parameters of a test problem seems like an easy task. Almost all software components have some input parameters, which could be used directly. This is also the case in most of the papers on combination strategies [52]. However, Yin *et al.* [12] pointed out that in choosing a set of parameters the problem space should be divided into sub-domains that conceptually can be seen as consisting of orthogonal dimensions. These dimensions do not necessarily map one-to-one onto the actual input parameters of the implementation. Along the same lines of thinking, Cohen *et al.* [23] stated that in choosing the parameters, one should model the system's functionality, not its interface.

It is an open question as to how the effectiveness and efficiency of the testing is affected by the choice of parameters and their values.

A second issue relating to the use of combination strategies not adequately investigated is constraints among the parameter values. It is not unusual in a test problem that certain subcombinations of parameter values are not allowed. For some of the combination strategies, for example AETG [17], it is possible to extend the algorithms to avoid selecting combinations that are not allowed. It is easy to see that this approach is not feasible for the instant combination strategies, for example OAs [34]. Another approach is to rewrite the test problem into several conflict-free subproblems [17,34,49]. A third approach is to change one or more of the values involved in an invalid combination [20]. These and possible other suggestions to handle constraints in the test problem have not been adequately investigated. Important questions to be answered include which constraint handling methods work for which combination strategies and how the size of the test suite is affected by the constraint handling method.

A third issue that needs further investigation is how to select an appropriate combination strategy for a particular test problem. As shown in Section 5.4, several comparisons have been made. Some general advice on the choice of combination strategies has been given. For instance, Ammann and Offutt [20] stated that the EC combination strategy may result in less effective test suites since test cases are assembled without considering any semantic information; thus the test cases actually selected by the combination strategy may have little to do with how the application is actually used in operation.

Beizer [3] explained that test cases that contain multiple invalid parameter values are normally not effective since most applications will find one of the values and then revert to some error handling routine, ignoring the values of any other parameters. Thus, a fault in the error handling routine for the second invalid value may be masked. Grindal *et al.* [18] observed that OA and AETG produced this behaviour and contributed this to the use of pair-wise strategies over the complete input parameter space. They suggested the combined use of a pair-wise strategy and BC, which ensures that each invalid value of any parameter will be included in at least one test case, where all other parameters are valid, that is, free of masking. Another approach for the same problem was proposed by Cohen *et al.* [22]. They suggested that the pair-wise combination strategies should be used only for the valid values of each parameter. For invalid values, some other combination strategy ensuring freedom from masking should be used. Although, undoubtedly useful for the tester, such information only provides partial advice to the tester when deciding upon which combination strategy to use.

Also, some experimental results give advice to the practicing tester. One such piece of advice is that all combination strategies that satisfy pair-wise coverage generate test suites of similar size. Despite this advice, the question of which combination strategies to select is still largely unanswered. In order to make combination strategies widely accessible to testers, future research needs to investigate all of these issues more thoroughly.

ACKNOWLEDGEMENTS

First and foremost, the authors would like to express their gratitude to the anonymous reviewers. Their constructive feedback really improved the presentation of this paper. In addition, Dr Martin Woodward and Dr Christine Cheng made significant contributions proof reading this manuscript. Further, the authors would like to thank the 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. This research is funded in part by KK-stiftelsen and Enea Test AB.

REFERENCES

1. Ostrand TJ, Balcer MJ. The Category-Partition method for specifying and generating functional tests. *Communications of the ACM* 1988; **31**(6):676–686.
2. Myers GJ. *The Art of Software Testing*. Wiley: New York, 1979.
3. Beizer B. *Software Testing Techniques*. Van Nostrand Reinhold: New York, 1990.
4. Myers GJ. A controlled experiment in program testing and code walkthroughs/inspection. *Communications of the ACM* 1978; **21**(9):760–768.
5. Basili VR, Selby RW. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering* 1987; **13**(12):1278–1296.
6. Reid S. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. *Proceedings of the 4th International Software Metrics Symposium (METRICS'97)*, Albuquerque, NM, 5–7 November 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 64–73.
7. Wood M, Roper M, Brooks A, Miller J. Comparing and combining software defect detection techniques: A replicated study. *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)* (*Lecture Notes in Computer Science*, vol. 1013). Springer: Berlin, 1997; 262–277.
8. So SS, Cha SD, Shimeall TJ, Kwon YR. An empirical evaluation of six methods to detect faults in software. *Software Testing, Verification and Reliability* 2002; **12**(3):155–171.
9. Duran JW, Ntafos SC. An evaluation of random testing. *IEEE Transactions on Software Engineering* 1984; **10**(4):438–444.
10. Hamlet D, Taylor R. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering* 1990; **16**(12):1402–1411.
11. Gutjahr WJ. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering* 1999; **25**(5):661–674.
12. Yin H, Lebne-Dengel Z, Malaiya YK. Automatic test generation using checkpoint encoding and antirandom testing. *Technical Report CS-97-116*, Colorado State University, 1997.
13. Grochtmann M, Grimm K. Classification trees for partition testing. *Software Testing, Verification and Reliability* 1993; **3**(2):63–82.
14. Grochtmann M, Grimm K, Wegener J. Tool-supported test case design for black-box testing by means of the classification-tree editor. *Proceedings of the 1st European International Conference on Software Testing, Analysis & Review (EuroSTAR 1993)*, London, U.K., October 1993. Qualtech Conferences, 1993; 169–176.
15. Lehmann E, Wegener J. Test case design by means of the CTE XL. *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*, Copenhagen, Denmark, December 2000. Qualtech Conferences, 2000; 1–10.
16. Williams AW, Probert RL. A measure for component interaction test coverage. *Proceedings of the ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, Beirut, Lebanon, June 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 304–311.
17. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7):437–444.
18. Grindal M, Lindström B, Offutt AJ, Andler SF. An evaluation of combination strategies for test case selection. *Technical Report HS-IDA-TR-03-001*, Department of Computer Science, University of Skövde, 2003.
19. Cohen MB, Gibbons PB, Mugridge WB, Colburn CJ. Constructing test cases for interaction testing. *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, May 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 38–48.
20. Ammann PE, Offutt AJ. Using formal methods to derive test frames in category-partition testing. *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS'94)*, Gaithersburg, MD, June 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 69–80.
21. Burroughs K, Jain A, Erickson RL. Improved quality of protocol testing through techniques of experimental design. *Proceedings of the IEEE International Conference on Communications (Supercomm/ICC'94)*, New Orleans, LA, 1–5 May 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 745–752.
22. Cohen DM, Dalal SR, Kajla A, Patton GC. The automatic efficient test generator (AETG) system. *Proceedings of Fifth International Symposium on Software Reliability Engineering (ISSRE'94)*, Monterey, CA, 6–9 November 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 303–309.
23. Cohen DM, Dalal SR, Parelius J, Patton GC. The combinatorial design approach to automatic test generation. *IEEE Software* 1996; **13**(5):83–89.
24. Shiba T, Tsuchiya T, Kikuno T. Using artificial life techniques to generate test cases for combinatorial testing. *Proceedings of 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, Hong Kong, China, 28–30 September 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 72–77.
25. Mandl R. Orthogonal Latin Squares: An application of experiment design to compiler testing. *Communications of the ACM* 1985; **28**(10):1054–1058.

26. Burr K, Young W. Combinatorial test techniques: Table-based automation, test generation and code coverage. *Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR'98)*, San Diego, CA, 26–28 October 1998. Software Quality Engineering, 1998.
27. Malaiya YK. Antirandom testing: Getting the most out of black-box testing. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'95)*, Toulouse, France, October 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 86–95.
28. Hoffman DM, Strooper PA, White L. Boundary values and automated component testing. *Software Testing, Verification and Reliability* 1999; **9**(1):3–26.
29. Sherwood G. Effective testing of factor combinations. *Proceedings of the Third International Conference on Software Testing, Analysis, and Review (STAR94)*, Washington, DC, *Software Quality Engineering*, May 1994. Software Quality Engineering, 1994.
30. Lei Y, Tai KC. In-parameter-order: A test generation strategy for pair-wise testing. *Proceedings of the Third IEEE High Assurance Systems Engineering Symposium*, November 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 254–261.
31. Lei Y, Tai KC. A test generation strategy for pairwise testing. *Technical Report TR-2001-03*, Department of Computer Science, North Carolina State University, Raleigh, 2001.
32. Tai KC, Lei Y. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 2002; **28**(1):109–111.
33. Huller J. Reducing time to market with combinatorial design method testing. *Proceedings of 10th Annual International Council on Systems Engineering (INCOSE'00)*, Minneapolis, MN, 16–20 July 2000. INCOSE, 2000.
34. Williams AW, Probert RL. A practical strategy for testing pair-wise coverage of network interfaces. *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE'96)*, White Plains, New York, 30 October–2 November 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 246–254.
35. Williams AW. Determination of test configurations for pair-wise interaction coverage. *Proceedings of the 13th International Conference on the Testing of Communicating Systems (TestCom 2000)*, Ottawa, Canada, August 2000. Kluwer Academic: Norwell, MA, 2000; 59–74.
36. Williams AW, Probert RL. Formulation of the interaction test coverage problem as an integer problem. *Proceedings of the 14th International Conference on the Testing of Communicating Systems (TestCom 2002)*, Berlin, Germany, March 2002. European Telecommunications Standards Institute, ETSI, 2002; 283–298.
37. Kobayashi N, Tsuchiya T, Kikuno T. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters* 2002; **81**(2):85–91.
38. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 1985; **11**(4):367–375.
39. Heller E. Using design of experiment structures to generate software test cases. *Proceedings of the 12th International Conference on Testing Computer Software*, New York, 1995. ACM Press: New York, 1995; 33–41.
40. Dalal S, Mallows CL. Factor-covering designs for testing software. *Technometrics* 1998; **50**(3):234–243.
41. Kuhn DR, Reilly MJ. An investigation of the applicability of design of experiments to software testing. *Proceedings of the 27th NASA/IEEE Software Engineering Workshop*, NASA Goddard Space Flight Center, MD, 4–6 December 2002. NASA/IEEE Computer Society Press: Los Alamitos, CA, 2002; 91–95.
42. Kuhn DR, Wallace DR, Gallo AM Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 2004; **30**(6):418–421.
43. Dunietz IS, Ehrlich WK, Szablak BD, Mallows CL, Iannino A. Applying design of experiments to software testing. *Proceedings of 19th International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997. ACM Press: New York, 1997; 205–215.
44. Piwowarski P, Ohba M, Caruso J. Coverage measurement experience during function test. *Proceedings of 15th International Conference on Software Engineering (ICSE'93)*, Baltimore, MD, May 1993. IEEE Computer Society Press: Los Alamitos, CA, 1993; 287–301.
45. Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM, Patton GC, Horowitz BM. Model-based testing in practice. *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, 16–22 May 1999. ACM Press: New York, 1999; 285–294.
46. Toczki J, Kocsis F, Gyimóthy T, Dányi G, Kókai G. Sys/3-a software development tool. *Proceedings of the Third International Workshop on Compiler Compilers (CC'90)*, Schwerin, Germany, 22–24 October 1990 (*Lecture Notes in Computer Science*, vol. 477). Springer: Berlin, 1990; 193–207.
47. Brownlie R, Prowse J, Phadke MS. Robust testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal* 1992; **71**(3):41–47.
48. Dalal SR, Jain A, Patton GC, Rathi M, Seymour P. AETG web: A web based service for automatic efficient test generation from functional requirements. *Proceedings of 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 21–23 October 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 84–85.

49. Daley N, Hoffman D, Strooper P. A framework for table driven testing of Java classes. *Software—Practice and Experience* 2002; **32**(5):465–493.
50. Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM. Model-based testing of a highly programmable system. *Proceedings of the 9th International Symposium in Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, 4–7 November 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 174–178.
51. Yilmaz C, Cohen MB, Porter A. Covering arrays for efficient fault characterization in complex configuration spaces. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, MA. *ACM Software Engineering Notes* 2004; **29**(4):45–54.
52. Kropp NP, Koopman PJ, Siewiorek DP. Automated robustness testing of off-the-shelf software components. *Proceedings of FTCS'98: Fault Tolerant Computing Symposium*, Munich, Germany, 23–25 June 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 230–239.
53. Berling T, Runesson P. Efficient evaluation of multi-factor dependent system performance using fractional factorial design. *IEEE Transactions on Software Engineering* 2003; **29**(9):769–781.