The AETG System: An Approach to Testing Based on Combinatorial Design

David M. Cohen, *Member*, *IEEE Computer Society*, Siddhartha R. Dalal, *Member*, *IEEE*
Michael L. Fredman, and Gardner C. Patton

Abstract—This paper describes a new approach to testing that uses combinatorial designs to generate tests that cover the pair-wise, triple, or n-way combinations of a system's test parameters. These are the parameters that determine the system's test scenarios. Examples are system configuration parameters, user inputs and other external events. We implemented this new method in the AETG system. The AETG system uses new combinatorial algorithms to generate test sets that cover all valid n-way parameter combinations. The size of an AETG test set grows logarithmically in the number of test parameters. This allows testers to define test models with dozens of parameters. The AETG system is used in a variety of applications for unit, system, and interoperability testing. It has generated both high-level test plans and detailed test cases. In several applications, it greatly reduced the cost of test plan development.

Index Terms—Testing, combinatorial designs, experimental designs, orthogonal arrays.

1 Introduction

Testing is an important but expensive part of the software development process. Much research has been aimed at reducing its cost. This paper describes a new approach to testing that uses combinatorial designs to generate efficient test sets. We implemented this method at Bellcore in the AETG system, which is used in Bellcore [8], [4], [5], [3], [6] and several of its clients [2] for unit, system, and interoperability testing.

In this new approach, the tester first identifies parameters that define the space of possible test scenarios. For example, the parameters to test adding records to a data base system would describe the records that can be added in a transaction. The tester then uses combinatorial designs to create a test plan that "covers" all pair-wise, triple, or n-way combinations of the test parameters.

The motivation is two fold. First, there are many systems where troublesome faults are caused by the interaction of a few test parameters. A test plan should ideally cover those interactions. The second is that the number of tests required to cover all n-way parameter combinations, for fixed n, grows logarithmically in the number of parameters. Thus, testers can define test models that have dozens of parameters and that still require only a small number of test cases. This gives testers the freedom to define models with enough detail to capture the semantics of the system under test accurately. They don't have to worry that refining a model by adding one more test parameter will cause the number of tests to explode. Models with 80 and more parameters are common.

We did some experiments to test the effectiveness of AETG tests. In one experiment, we found faults in previously tested modules from two releases of a Bellcore system, System A. In another, we measured the code coverage given by AETG tests for several Unix commands and several modules from System A. The pair-wise AETG test sets gave good code coverage for both examples.

The AETG system is used in a variety of applications. This paper describes two sample applications. In one, it designed a high-level test plan for the telephone 800 service software. In the other, it created detailed test cases for an ATM network performance monitoring system. Designing good test plans for these systems by hand would usually require one or two months. The AETG system reduced the time to one or two weeks.

This paper reports on the AETG system. The next section motivates the basic paradigm with an example. Section 3 proves that the number of tests required by the combinatorial design approach grows logarithmically in the number of test parameters. Section 4 gives a heuristic algorithm to generate tests. Section 5 gives an overview of the AETG input language. Sections 6 and 7 describe the experiments we did concerning effectiveness and two sample applications. Section 8 concerns related work.

2 The Basic Combinatorial Design Paradigm

Consider the problem of testing a telephone switch's ability to place telephone calls. Table 1 shows four parameters that define a very simple test model. The *Call Type* parameter tells the type of call. Its values are Local, Long Distance,

- *D.M. Cohen is with the IDA Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715. E-mail: dmcohen@super.org.*
- *S.R. Dalal and G.C. Patton are with Bellcore, 445 South St. Morristown, NJ 07960. E-mail: {sid, gcp}@bellcore.com.*
- *M.L. Fredman is with the Department of Computer Science, Rutgers University, Busch Campus–Hill Center, New Brunswick, NJ 08903. E-mail: fredman@cs.rutgers.edu.*

Please address all product inquiries about the AETG System to S.R. Dalal, Bellcore, 445 South St., Morristown, NJ 07960. The AETG System is covered by United States Patent 5,542,043.
Manuscript received 22 Aug. 1995; revised 2 Feb. 1997.
Recommended for acceptance by R. Hamlet.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 103736.

and International. The *Billing* parameter says who pays for the call. Its values are Caller (for bill to caller), Collect, and 800. The *Access* parameter tells how the calling party's phone is connected to the switch. The options considered in this simple model are Loop, ISDN line, and a PBX trunk. The final parameter, *Status*, tells whether or not the call was successful or failed either because the calling party's phone was busy or the call was blocked in the phone network.

Since each different combination of parameter values determines a different test scenario, and each of the four parameters has three values, the table defines a total of 3^4 = 81 different scenarios. Suppose for argument's sake that 81 tests is too many as each individual test is expensive. Then one alternative would be to select a default value for each parameter and then vary one parameter in each test until all the parameter values are covered. Table 2 shows the resulting test set. It has nine tests instead of the 81 required for exhaustively testing all possible parameter combinations. However, although it covers all individual parameter values, it covers only 30 of the $9 \times 6 = 54$ possible pair-wise interactions between the test parameters.

The test plan shown in Table 3 also has nine test cases but, unlike the default test plan in Table 2, it covers every pair-wise combination of parameter values. This test plan was constructed using a well-known combinatorial design based on the projective plane [12]. Since 2/3 of the calls in this test plan do not complete successfully, this plan does not reflect the system's normal operational profile. In many applications, a significant number of faults are caused by parameter interactions that occur in atypical, yet realistic, situations [4], [22]. A comprehensive test should also cover these interactions. Since the combinatorial design method covers them very efficiently, testers who feel that the test plan should reflect the operational profile can use the combinatorial design method to complement tests derived from the operational profile.

Suppose that the test model had 13 parameters instead of 9 and that each had three values, say 0, 1, and 2. Then there are 3^{13} = 1,594,323 possible parameter combinations. The default method for this configuration requires 27 test cases and covers $(13 \times 12/2) + 26 \times 12 = 390$ of the $(13 \times 12/2) \times 9 = 702$ possible pair-wise parameter combinations. Table 4 [7] shows 15 tests that cover all pair-wise parameter combinations. In general, default testing for n parameters with three values each requires $2 \times n + 1$ tests and covers 5/9 of the pair-wise parameter combinations while pair-wise testing requires $4 \times \log_2 n$ tests and covers all the pair-wise combinations. For example, for $n = 40$, exhaustive testing requires $3^{40} = 1.2 \times 10^{19}$ tests and default testing 81 tests. Pair-wise testing requires only 21 tests.

In the next section, we show that the number of test cases required to test all pair-wise or n-way parameter combinations grows logarithmically in the number of parameters.

3 Logarithmic Growth for n-Way Interaction Testing

We now show that the number of tests for n-way coverage for fixed n grows logarithmically in the number of parameters. For ease of notation, we state the proof for pair-wise coverage, i.e., for $n = 2$. The logarithmic growth follows from the following.

TABLE 1
PARAMETERS FOR PLACING A TELEPHONE CALL

Call Type	*Billing*	*Access*	*Status*
Local	Caller	Loop	Success
Long Distance	Collect	ISDN	Busy
International	800	PBX	Blocked

TABLE 2
DEFAULT TEST CASES FOR PLACING A PHONE CALL

Call Type	*Billing*	*Access*	*Status*
Local	Caller	Loop	Success
Long Distance	Caller	Loop	Success
International	Caller	Loop	Success
Local	Collect	Loop	Success
Local	800	Loop	Success
Local	Caller	ISDN	Success
Local	Caller	PBX	Success
Local	Caller	Loop	Busy
Local	Caller	Loop	Blocked

TABLE 3
PAIR-WISE TEST CASES FOR PLACING A PHONE CALL

Call Type	*Billing*	*Access*	*Status*
Local	Collect	PBX	Busy
Long Distance	800	Loop	Busy
International	Caller	ISDN	Busy
Local	800	ISDN	Blocked
Long Distance	Caller	PBX	Blocked
International	Collect	Loop	Blocked
Local	Caller	Loop	Success
Long Distance	Collect	ISDN	Success
International	800	PBX	Success

TABLE 4
FIFTEEN TEST CASES FOR 13 PARAMETERS WITH THREE VALUES EACH

	P1	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*	*P13*
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	0	0	0	0
3	2	2	2	2	2	2	2	2	2	0	0	0	0
4	0	0	0	1	1	1	2	2	2	1	1	1	0
5	1	1	1	2	2	2	0	0	0	1	1	1	0
6	2	2	2	0	0	0	1	1	1	1	1	1	0
7	0	0	0	2	2	2	1	1	1	2	2	2	0
8	2	2	2	1	1	1	0	0	0	2	2	2	0
9	1	1	1	0	0	0	2	2	2	2	2	2	0
10	0	1	2	0	1	2	0	1	2	0	1	2	1
11	1	2	0	1	2	0	1	2	0	1	2	0	1
12	2	0	1	2	0	1	2	0	1	2	0	1	1
13	0	2	1	0	2	1	0	2	1	0	2	1	2
14	2	1	0	2	1	0	2	1	0	2	1	0	2
15	1	0	2	1	0	2	1	0	2	1	0	2	2

THEOREM. *Given a system with k parameters, each of which has l values, suppose that r test cases have already been chosen and that the number of uncovered pairs is N. Then there is a test case that covers at least N/l^2 new pairs.*

PROOF. Consider the set

U = {(t, p): where t is a test case and p is a pair covered by t}.

Since there are k parameters, each test case t covers $k(k-1)/2$ pairs. Thus, each test case t appears in U with $k(k-1)/2$ different pairs p. Since each parameter has l values, each pair p appears in U with l^{k-2} different test cases.

Now, consider the subset V of (t, p) such that the test case t is not one of the r already selected test cases and the pair p is not covered by any of the selected test cases. We will prove the theorem by counting the cardinality of V in two different ways.

For any (t, p) in V, since the pair p is not covered, no test case that contains p is among the already selected test cases. Thus, p appears in V with l^{k-2} different test cases t, i.e., there are l^{k-2} different test cases t such that (t, p) is in V. Thus, if the number of uncovered pairs is N, the cardinality of V is $N \times l^{k-2}$.

$$Card(V) = N \times l^{k-2}$$

For each unselected test case t, let m_t be the number of new pairs covered by t. Let m_t be 0 if t is one of the r selected test cases. Then t appears in V m_t times. Thus,

$$Card(V) = \sum_t m_t$$

Now let m be the largest of the m_t. To prove the theorem, we must show that m is at least N / l^2. Since the total number of possible test cases is l^k, we have the following inequality:

$$Card(V) = \sum_t m_t \le m \times l^k.$$

Thus, $N \times l^{k-2} \le m \times l^k$, or

$$N / l^2 \le m.$$

We proved the theorem by showing that at each step in generating a test set, there is a test case that covers $1 / l^2$ of the remaining pairs. Now consider a greedy algorithm that at each step chooses a test case that covers the most uncovered pairs. Let N be the number of pairs at the start. Since there are k fields with l values each, $N = k(k-1)/2 \times l^2$. Using the greedy algorithm, after r test cases have been chosen, the number of remaining pairs is

$$N' \le N \times (1 - 1/l^2)^r.$$

Thus, if $r > -\log(N) / \log(1 - 1/l^2)$ then $N' < 1$ and all the pairs have been covered. Using the approximation that $\log(1 - 1/l^2) = -1/l^2$, we get that the greedy algorithm covers all pairs when

$$r > l^2 \times \log(N) \ge l^2 \times (\log(k(k-1)/2) + 2\log(l)).$$

This shows that the number of test cases required by the greedy algorithm grows at most logarithmically in k and quadratically in l. Gargano, Körner, and Vaccaro have shown [11] that for very large values of k, the number of test cases n satisfies

$$n \sim \frac{l}{2} \log_2 k.$$

Their results are nonconstructive and it seems unlikely that the linear growth in l is true for moderate values of k.

4 A HEURISTIC ALGORITHM

The proof of logarithmic growth for the greedy algorithm assumes that at each stage it is possible to find a test case that covers the maximum number of uncovered pairs. Since there can be many possible test cases, it is not always computationally possible to find an optimal test case. We now outline a random greedy algorithm we developed. Again for simplicity of notation we state the algorithm for pairwise coverage.

Assume that we have a system with k test parameters and that the ith parameter has l_i different values. Assume that we have already selected r test cases. We select the $r + 1$ by first generating M different candidate test cases and then choosing one that covers the most new pairs. Each candidate test case is selected by the following greedy algorithm:

1) Choose a parameter f and a value l for f such that that parameter value appears in the greatest number of uncovered pairs.
2) Let $f_1 = f$. Then choose a random order for the remaining parameters. Then, we have an order for all k parameters $f_1, \ldots f_k$.
3) Assume that values have been selected for parameters $f_1, \ldots, f_j$. For $1 \le i \le j$, let the selected value for f_i be called v_i. Then, choose a value v_{j+1} for f_{j+1} as follows. For each possible value v for f_j, find the number of new pairs in the set of pairs $\{f_{j+1} = v$ and $f_i = v_i$ for $1 \le i \le j\}$. Then, let v_{j+1} be one of the values that appeared in the greatest number of new pairs.

   Note that, in this step, each parameter value is considered only once for inclusion in a candidate test case. Also, that when choosing a value for parameter f_{j+1}, the possible values are compared with only the j values already chosen for parameters $f_1, \ldots, f_j$.

We did many experiments with this algorithm. When we set $M = 50$, i.e., when we generated 50 candidate test cases for each new test case, the number of generated test cases grew logarithmically in the number of parameters (when all the parameters had the same number of values). Increasing M did not dramatically reduce the number of generated tests. Since the candidate test cases depend on the random order selected in Step 2, using a different random seed can produce a different test set. One useful optimization is to generate 50 different test sets using 50 different random seeds and then choose the best among them. This sometimes reduces the number of generated tests by 10 to 20 percent. There is also a deterministic construction and an alternative random algorithm [7] that sometimes generate fewer test cases.

5 AETG INPUT LANGUAGE

The basic constructs of the AETG input language are *fields* and *relations*. The fields are the system's test parameters

and the relations define relationships between the test parameters. To define a relation, the tester specifies the fields it contains and a set of *valid* and *invalid* values for each field. A test generated from valid values is a *valid test* and a test generated from valid and invalid values is an *invalid test*. Invalid tests usually abort before completion because of some error condition.

Table 5 shows two relations that refine the test model given in Table 1 in Section 2. The two relations, Relation 1 and Relation 2, have the same four fields: *Call Type*, *Billing*, *Access*, and *Status*. Relation 1 defines $2 \times 3^3 = 54$ different test scenarios, and relation 2 defines $2 \times 3^2 = 18$ different test scenarios. Since *International* isn't a valid value for *Call Type* in relation 1 and *800* isn't a valid value for *Billing* in relation 2, the set of test scenarios defined by Table 5 has the constraint that the pair (*Call Type* = *International*) & (*Billing* = 800) is not valid, i.e., that there are no international calls to 800 numbers.

TABLE 5
TWO RELATIONS FOR PLACING A CALL WITH CONSTRAINT

Relation 1			
Call Type	*Billing*	*Access*	*Status*
Local	Caller	Loop	Success
Long Distance	Collect	ISDN	Busy
	800	PBX	Blocked

Relation 2			
Call Type	*Billing*	*Access*	*Status*
Internat.	Caller	Loop	Success
	Collect	ISDN	Busy
		PBX	Blocked

The tester specifies a *degree* to test for each relation. If the tester specifies pair-wise testing, the AETG system generates tests that cover all valid pair-wise combinations of values of the relation's fields. This means that for any two fields f_1 and f_2 and any values v_1 for f_1 and v_2 for f_2, there is some test in which f_1 has the value v_1 and f_2 has the value v_2. If the tester specifies n-way testing, the AETG system will generate a test set that covers all n-way parameter combinations for each relation.

The AETG system generates tests for a set of relations by combining tests for the individual relations. The algorithm for combining tests insures that for each combined test there is a set of relations such that the projection of the test onto the fields in each relation in the set is a test for that relation. If two relations have no common fields, the combined tests for the two relations are simply concatenations of tests for each individual relation.

The AETG system generates an *invalid* test for each invalid value specified for a field in a relation. A value is an invalid value only within the context of a relation. A value which is invalid for a field in one relation may be a valid value for that field in another relation. To avoid having one invalid value mask another the AETG system uses only one invalid value per test case. It creates the test for an invalid value by taking a valid test for the relation and substituting the invalid value in place of the field's valid value in that test.

The tester can also guarantee inclusion of their favorite test cases by specifying them as *seed tests* or *partial seed tests* for a relation. The seed tests are included in the generated test set without modification. The partial seed tests are seed test cases that have fields that have not been assigned values. The AETG system completes the partial test cases by filling in values for the missing fields.

5.1 Constraints

While constraints can be expressed using multiple relations as shown in Table 5, it may be more efficient to express them explicitly by using *unallowed* tests. An unallowed test for a relation specifies a set of test cases that are not valid for that relation. Table 6 shows a relation with an explicit constraint. The relation, Relation 3, has the same four fields as the two relations in Table 5. It also has the explicit constraint that any test case with (*Call Type* = *International*) & (*Billing* = 800) is not allowed, independent of the values for the *Access* and *Status* fields (the * in Table 6 is a wild card).

Relation 3 defines the same set of possible test scenarios as relations 1 and 2, but the two AETG inputs are not identical. Since relations 1 and 2 have incompatible values for the *Call Type* field, tests generated for one relation are not valid tests for the other. Since each relation requires nine tests for pair-wise coverage, the union of the two test sets has 18 tests. Relation 3 requires only the 10 tests shown in Table 7.

TABLE 6
DEFINITION OF RELATION 3

Relation 3			
Call Type	*Billing*	*Access*	*Status*
Local	Caller	Loop	Success
Long Distance	Collect	ISDN	Busy
International	800	PBX	Blocked

Constraints for relation 3			
International	800	*	*

TABLE 7
TEN TEST CASES FOR RELATION 3

Call Type	*Billing*	*Caller Access*	*Status*
Local	Collect	PBX	Busy
Long Distance	800	Loop	Busy
International	Caller	ISDN	Busy
Local	800	ISDN	Blocked
Long Distance	Caller	PBX	Blocked
International	Collect	Loop	Blocked
Local	Caller	Loop	Success
Long Distance	Collect	ISDN	Success
International	Caller	PBX	Success
Local	800	PBX	Success

Relations 1 and 2 together require more tests than relation 3 because they impose more stringent test requirements. Relation 1 specifies that the pair (*Access* = *ISDN*) & (*Status* = *Busy*) is covered in the context of *Call Type* = (*Local or Long Distance*) and relation 2 specifies that the pair is covered in the context of *Call Type* = *International*. Conse-

quently, the pair is covered twice in the union of the test sets for the two relations, once in each context. However, the pair is covered only once in the test set for relation 3. A relation specifies not only a set of pairs to be covered but also a context for those pairs.

In this example, the tester may not care if the pair (*Access = ISDN*) & (*Status = Busy*) is covered in both contexts. In that case, an alternative semantics would regard the relation as specifying only a set of pairs and not a context. The two specifications would then be equivalent and Table 7 would be a test plan for either specification.

A simple test generation algorithm is to first generate tests for one relation and then use them to account for pairs in the other relation. This algorithm however does not generate a minimal test set. For example, consider first covering Relation 1 and then Relation 2. Relation 1 would still require nine test cases and Relation 2 would require two test cases, one for the pair (*Call Type = International*) & (*Billing = Caller*) and one for the pair (*Call Type = International*) & (*Billing = Collect*). The combined test set would then have 11 test cases. This in one more than the 10 shown in Table 7.

We doubt that testers would prefer as a rule to ignore the context provided by the relation. Testers often use different relations to define different semantic situations. For example, they may have one relation to define requirements to test a line interface card when the line's protocol is Ethernet and another for when the protocol is ATM. The tester would want to insure that flow control worked in both environments.

Since the fields in an AETG relation have only a finite number of values, the user-interface can translate higher level constraints such as $x \neq y$ and $x \leq y$ into the unallowed tests.

5.2 Hierarchy and Hierarchical Testing

A system often has several natural degrees of interaction between its fields. A few fields might be important and the tester may want to test their interactions with each other more intensively then their interactions with the rest of the system. One option is to have two relations. One which contains all the fields and which is tested for pair-wise combinations and another which contains only the most important fields and which is tested for a high-degree of interaction. However, that would be wasteful. A better solution is to use a *subrelation*.

A subrelation is a relation that is used as a part of another relation. The tester can put the most important fields into a subrelation and give it a high degree of interaction testing. The tester can then use the subrelation inside relations that are tested for a lower degree of interaction. When generating tests, the AETG system will first generate tests that cover the subrelation's specified degree of interaction and then use those tests as partial seed test cases when generating tests for the containing relation.

6 Experiments

We did experiments to check the effectiveness of AETG test sets. In one experiment, we tested user interface modules from two releases of a Bellcore system. In another, we measured code coverage of AETG test sets.

In the first experiment, the AETG system tested modules from two releases of a Bellcore system, System A. It tested nine modules from the first release and 13 from the second. The modules were designed to validate the user's input for internal consistency. A validation module usually has from 1,000 to 2,000 lines of C code. In this experiment, the testers created the AETG input from the module's detailed development requirements. Although the modules had already been tested, the experiment found problems caused by defects in the code and by defects in the requirement's documents.

Table 8 shows the results. The column labeled Code shows the number of code defects and the column labeled Requirements shows the number of requirement defects. There are more requirements defects than code defects. The requirements defects are introduced when the system engineers take the high-level user-oriented requirements and write detailed development requirements. This process requires a great deal of effort and knowledge; faults are often introduced during it. Many of these faults are corrected in the code later in the development process. Finding and documenting them is important since the detailed development requirements are used for maintenance.

We also measured the code coverage given by AETG test sets. We used the ATAC [15], [23] coverage tool to measure the block, decision, C-uses and P-uses metrics of AETG tests generated for several Unix commands and several validation modules from System A. The pair-wise AETG test sets gave over 90 percent block coverage for both application domains. For example, a set of 29 pair-wise AETG tests gave 90 percent block coverage for the Unix sort command. We also compared pair-wise testing with random input testing and found that pair-wise testing gave better coverage. For the modules from System A, we also found that code coverage didn't increase when we increased from pair-wise testing to testing all valid input combinations. Table 9 has the results for one module. The Unix coverage experiments are discussed in detail in [5].

Several coverage experiments were done at Nortel by Burr and Young [2]. They also found that the AETG pair-wise test sets gave good coverage in a variety of situations.

Of course, it is easy to construct examples where only one unique combination of test parameters will trigger a fault. However, there is growing evidence that for many real-world systems, a large number of faults are triggered by many parameter combinations. This is an area that merits further study.

TABLE 8
DEFECTS FOUND IN TWO RELEASES OF SYSTEM A

	9 modules from release 1		*13 modules from release 2*	
	Code	*Requirements*	*Code*	*Requirements*
Range	0-5	0-5	0-3	0-10
Average	2	4	1	3

TABLE 9
CODE COVERAGE RESULTS FOR MODULE A

Code coverage results for module A					
Method	*No of tests*	*Block*	*Decision*	*P-uses*	*C-uses*
Pair-wise	200	92	85	49	72
All	436	92	85	49	72
Random	300	67	58	36	55

7 Overview of Applications

The AETG system is used to generate both high-level test plans and detailed test cases. This section gives an example to illustrate each type of application. Other applications are discussed in [4], [5], [6], [3].

7.1 High-Level Test Planning

In this example, the AETG system designed a test plan for the telephone switch software implementing 800 service. Table 10 shows the relation to test calls reaching the switch on a trunk from another switch. The first three fields specify the trunk's type, its high-level protocol, and its signaling protocol. The next two fields specify attributes of the caller's phone line. The last field says if the caller's phone number (ANI) is known to the switch. The three constraints specify that certain trunk types can not use ISDN signaling. Table 10 defines 336 possible valid test scenarios (480 scenarios without the constraint). Pair-wise testing required only 30 tests. Since each test scenario takes a few hours to run, going from 336 tests to 30 means a considerable cost savings. The AETG input for the complete 800 software had two additional relations and required 100 tests in total.

TABLE 10
800 Service Testing—Calls Arriving on Trunks

Relation for Calls arriving on trunks

Trunk			*Phone*		ANI
Type	*Protocol*	*Signalling*	*Phone*	*Class*	
Inter-office	FGC	MF	flat rate	No	No
PBX	FGD	ISDN	measured srv	Yes	Yes
Operator			ISDN phone		
Cellular			business		
Billing			coin		
			multi-party		

Unallowed combinations for calls arriving on trunks

Operator	*	ISDN	*	*	*
Cellular	*	ISDN	*	*	*
Billing	*	ISDN	*	*	*

7.2 Test Case Generation

In this example, the AETG system generated detailed test cases for an ATM network monitoring system. It generated tests for two releases. Creating the AETG input for the first release took one week and modifying it for the second took an hour. This system has several monitors each of which can signal when the number of corrupted ATM cells exceeds a specified threshold during a specified unit of time. The system has commands to turn monitors on and off, to set their thresholds and time units, and to display statistics. To test the system, a tester gives it some configuration commands and then uses an attenuator to corrupt the ATM transmission facilities. The tester then checks if the system displays the correct statistics.

The AETG input had one relation and modeled the configuration commands and the attenuator. The input for the first release had 61 fields; 29 fields with two values, 17 with three values, and 15 with four values. This gives a total of $2^{29} \times 3^{17} \times 4^{15} = 7.4 \times 10^{25}$ different combinations. The input for the second release had 75 fields; 35 fields with two values, 39 with three values, and 1 with four values. This gives a total of $2^{35} \times 3^{39} \times 4 = 5.5 \times 10^{29}$ different combinations. The AETG system generated 41 pair-wise tests for the first release and 28 pair-wise tests for the second. Even though the second release had many more combinations, pair-wise coverage required fewer tests. This illustrates the logarithmic growth properties of the AETG method. Even though the second release had six more fields with two values and 22 more fields with three values, it required fewer tests because it had 14 fewer fields with four values.

This example also illustrates the distinction between the AETG approach and some forms of input testing. Even though the system had a screen interface, the AETG fields modeled the system's commands and not its user interface. This distinction is discussed in greater detail in [5].

8 Related Methods

The combinatorial design paradigm is a "black box" approach to testing; i.e., it generates tests from a model of the system's expected functionality. The test model can be created from the system's functional requirements or from its detailed development specifications. The combinatorial design approach differs from most other black box methods in that its basic test requirement is coverage of all valid n-way test parameters combinations for tester defined values of n.

A method related to our approach is random input testing and partition testing (see, e.g., Duran and Ntafos [9] and Hamlet and Taylor [13]). The AETG approach differs from random testing by allowing the tester to define complex relationships between the test parameters. The tester can use the AETG constructs for relations, constraints and hierarchy to focus testing. The AETG test plans are far from random.

Closely related to our work is the use by Mandl [17], Brownlie, Prowse, and Phadke [1] and Heller [14] of *orthogonal arrays* to generate pair-wise test sets. Orthogonal arrays are combinatorial designs used to design statistical experiments [21], [18]. Because of their use in statistical experimentation, they have a balance requirement that every pair is covered the same number of times. The AETG approach requires only that every pair is covered at least once. It does not specify how many times each pair is covered.

The orthogonal array balance requirement is very severe and preclues logarithmic growth in the number of test parameters. For example, an orthogonal array for 100 parameters each with two values would require 101 test cases. An unbalanced pair-wise test set requires only 10 tests. (The construction uses a combinatorial argument due to Renyi [20].) Many applications have a large number of parameters that have only a few values each. For these applications, the balance requirement causes the number of test cases generated by orthogonal arrays to grow unacceptably large. For example, it would not be practical to test the application described in Section 7.2 using a balanced test set.

Another problem with balanced test sets is the incorporation of constraints that specify that some combinations of values are invalid and must not occur in any test case. How does one efficiently modify a balanced test set to prevent some pair from occurring while insuring that the other pairs still occur the same number of times? In contrast, it is easy to

incorporate constraints into the heuristic algorithm in Section 4. One can either throw away candidate test cases that violate a constraint or one can avoid generating them by not selecting a parameter value if it would violate a constraint.

By eliminating the balance requirement, we reduced the number of required test cases to logarithmic growth in the number of parameters. We also allowed easy specification of constraints. Together these two properties allow testers to have test models with many parameters. Test models with 80 and more parameters are common. Testers are free to add detail to a model by defining new test parameters.

The closest work to the AETG system is the CATS system developed by Sherwood at AT&T [19], [16], [10]. CATS generates test sets that give n-way coverage for a set of relations. However, it does not have the AETG notions of explicit constraints or hierarchy. Instead, it uses multiple relations to express constraints. As shown above, using multiple relations instead of explicit constraints may require more tests than necessary. While we have not done a comparison study of the AETG algorithms verses the CATS algorithms, the published data suggest that the AETG algorithms generate fewer test cases than CATS. For example, Sherwood [19] reports that CATS generated 240 tests for pair-wise coverage of 20 fields with 10 values each. The AETG algorithms generate only 180 tests for this example [7].

9 Summary

The AETG system uses new combinatorial design algorithms to generate test sets that efficiently cover the pair-wise or n-way combinations of a system's test parameters. Examples of such parameters are a system's configuration parameters, the parameters that define its environment, its inputs and internal events.

The basic AETG test requirement is that every pair-wise or n-way combination of parameter values is covered. Unlike the orthogonal array approach, the AETG method does not require that every combination is covered the same number of times. By allowing unbalanced test sets, it greatly reduces the number of tests required to check the specified level of interactions. For example, a balanced test set for a system with 100 binary fields requires 101 tests, while an unbalanced test set requires only 10.

In general, the number of tests required by the AETG method grows logarithmically in the number of test parameters. For example, checking all pair-wise combinations of 13 fields with three values each requires only 15 tests out of a potential 1.5 million test combinations. Consequently, the cost of adding detail in the form of additional parameters is logarithmic. This is in contrast to models such as the finite state model where each new feature adds a multiplicative factor to the number of tests.

Testers can use the AETG constructs to focus testing. The AETG constructs for relations, constraints, and hierarchy allow testers to express knowledge about the system under test. The AETG test cases are far from random. In several experiments with code coverage, the AETG test sets gave significantly better coverage than randomly generated tests.

The AETG system is used in a variety of applications for unit, system, and interoperability testing. It has generated both high-level test plans and detailed test cases. Testers can base the AETG input on detailed development requirements or on a system's high-level functional requirements, such as its user manual. The experience with this new approach indicates that it is widely applicable and generates efficient test sets of good quality.

Acknowledgments

The authors thank Ajay Kajla, George Horruitiner, David Carmen, Kirk Burroughs, Aridaman Jain, Robert Erickson of Bellcore, and Nishit Goel, Kevin Burr, William Young and Steve Yu of Nortel for developing new AETG applications. We thank Ajay Kajla and Jesse Parelius for their work on the code coverage experiments. Finally, we thank Isaac Perelmutter, Adam Irgon, and Jon Kettenring for their support.

References

[1] R. Brownlie, J. Prowse, and M. Phadke, "Robust Testing of AT&T PMX/StarMail Using OATS," *AT&T Technical J.*, vol. 71, no. 3, pp. 41-47, Mar. 1992.

[2] K. Burr and W. Young, "Test Acceleration and Automatic Efficient Testcase Generation," Nortel Technical Report, May 1997.

[3] K. Burroughs, A. Jain, and R.L. Erickson, "Improved Quality of Protocol Testing through Techniques of Experimental Design," Proc. *Supercomm/IEEE Int'l Conf. Comm. '94*, pp. 745-752, 1994.

[4] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, "The Automatic Efficient Tests Generator," *Proc. Fifth Int'l Symp. Software Reliability Eng.*, pp. 303-309, IEEE, 1994.

[5] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software*, vol. 13, no. 5, pp. 83-89, Sept. 1996.

[6] D.M. Cohen, S.R. Dalal, G. Horruitiner, and G.C. Patton, "The AETG System," *Fifth Int'l Conf. Software Testing, Analysis and Review*, Software Quality Eng., Jacksonville, Fla., 1996.

[7] D.M. Cohen and M.L. Fredman, "New Techniques for Designing Qualitatively Independent Systems," *J. of Combinatorial Designs*, to appear.

[8] S.R. Dalal and G.C. Patton, "Automatic Efficient Test Generator (AETG): A Test Generation System for Screen Testing, Protocol Verification, and Feature Interactions Testing," *Internal Bellcore Technical Memorandum*, 1993.

[9] J. Duran and S. Ntafos, "An Evaluation of Random Testing," *IEEE Tran. Software Eng.*, vol. 10, pp. 438-444, July 1984.

[10] W.K. Ehrlich, I.S. Dunietz, B.D. Szablak, C.L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," *Proc. 19th Int'l Conf. Software Eng.*, IEEE, 1997.

[11] L. Gargano, J. Körner, and U. Vaccaro, "Sperner Capacities," *Graphs and Combinatorics*, vol. 9, pp. 31-46, 1993.

[12] M. Hall Jr., *Combinatorial Theory*. New York: Wiley Interscience, 1986.

[13] D. Hamlet, and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.*, vol. 16, pp. 1,402-1,412, Dec. 1990.

[14] E. Heller, "Using Design of Experiment Structures to Generate Software Test Cases," *12th Int'l Conf. Testing Computer Software*, pp. 33-41, June 1995,.

[15] J.R. Horgan and S. London, "ATAC: A Data Flow Coverage Testing Tool for C," *Proc. IEEE Assessment of Quality Software Development Tools*, pp. 2-10, IEEE, 1992.

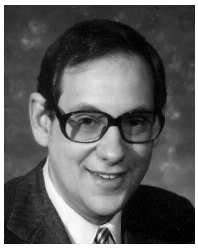
[16] C. Mallows, "Covering Designs in Random Environments," *Festschrift for John Tukey*, 1997, to appear.

[17] R. Mandl, "Orthogonal Latin Squares: An Application of Experimental Design to Compiler Testing," *Comm. ACM*, vol. 28, no. 10, pp. 1,054-1,058, Oct. 1985.

[18] M.S. Phadke, *Quality Eng. Using Robust Design*. Englewood Cliffs, N.J.: Prentice Hall, 1989.

[19] G. Sherwood, "Effective Testing of Factor Combinations," *Third Int'l Conf. Software Testing, Analysis and Review*, Software Quality *Eng.*, Jacksonville, Fla., 1994.

[20] N.J.A. Sloane., "Covering Arrays and Intersecting Codes," *J. Combinatorial Designs*, vol. 1, no. 1, pp. 51-63, 1993.
[21] G. Taguchi, *System of Experimental Design*. Quality Resources, 1987. Translation of *Jikken keikakuho*, Maurzen Co., Tokyo, 1976.
[22] C.H. West, "Protocol Validation—Principles and Applications," *Computer Networks and ISDN Systems*, vol. 24, no. 3, pp. 219-242, May 1992.
[23] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Proc. 17th Int'l Conf. Software Eng.*, pp. 41-50, IEEE, 1995.



David M. Cohen received a BA degree from Harvard University and a PhD degree in mathematics from MIT. He is a member of the research staff of IDA's Center for Computing Sciences in Bowie, Maryland, where he does research in computer science in support of the NSA. From 1981 to 1996, he worked at Bellcore and Bell Telephone Labs, where he most recently did research in software engineering and telecommunications; he has two patents. Cohen has held postdoctoral fellowships from the Institute for Advanced Study in Princeton, New Jersey and the German Alexander von Humboldt Foundation. Cohen is a member of the ACM and the IEEE Computer Society.

Siddhartha R. Dalal received his MBA and PhD degrees from the University of Rochester, and then started his industrial research career at the Bell Labs Math Research Center. He is a chief scientist and a director of the Information Technologies and Internet Applications Laboratory at Bellcore. Besides leading the research at Bellcore on combinatorial designs and the AETG system, he leads research projects in software engineering, risk analysis, mathematics, and statistics. He received the American Statistical Association's 1988-1989 award for an outstanding application paper for work on the Challenger disaster on behalf of a National Research Council committee. He is a member of a National Academy of Sciences panel that produced an NRC report on software engineering. He is a fellow of the American Statistical Association and an associate editor of the *Journal of the American Statistical Association*.

Michael L. Fredman received his BS degree in mathematics from California Institute of Technology and a PhD degree in computer science from Stanford University. He is currently a professor in the Department of Computer Science at Rutgers University Previously he was a professor of mathematics and a professor of computer science at the University of California at San Diego. Fredman's research interests mainly concern algorithms and data structures. He is a member of the editorial board of the *SIAM Journal on Computing* and is a member of the ACM.

Gardner C. Patton received the BA degree in physics from Brown University and a MS in industrial engineering from the New Jersey Institute of Technology. He is a senior scientist in the Software Engineering and Statistical Research group at Bellcore. After joining Bell Laboratories in 1961, he worked on developing software missile guidance systems, telephone applications, and operating systems. For 14 years, he managed a group testing TIRKS, a telephone inventory system containing more than 10,000,000 lines of code. He is currently interested in test measurement, automatic test generation, client/server testing, and intranet testing. He is a member of the ACM.