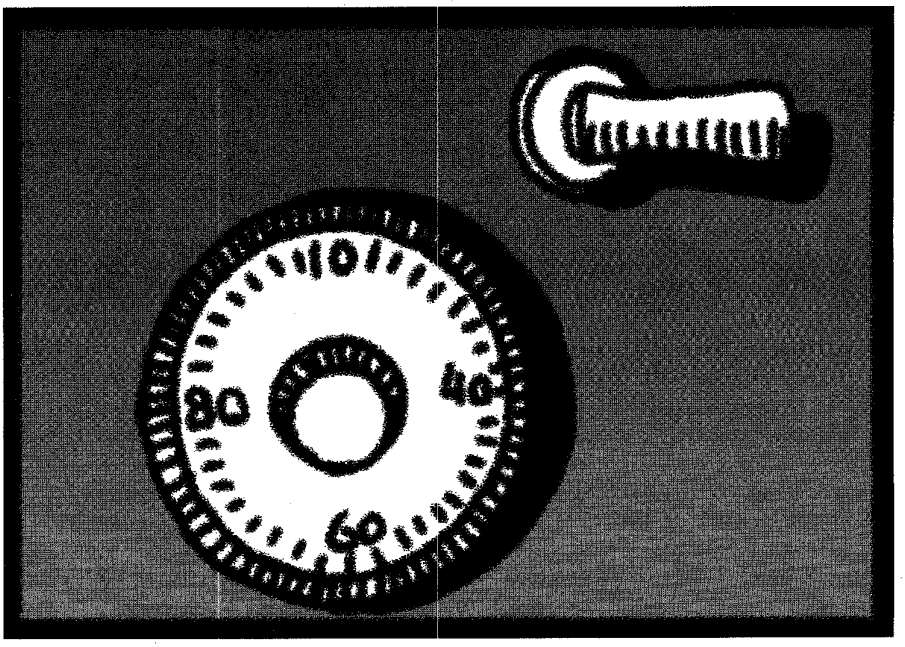
This work will be presented at the 1996 International Symposium on Software Reliability Engineering, White Plains, N.Y., October 30-November 2.

The Combinatorial Design Approach to Automatic Test Generation

The combinatorial design method substantially reduces testing costs. The authors describe an application in which the method reduced test plan development from one month to less than a week. In several experiments, the method demonstrated good code coverage and fault detection ability.



DAVID M. COHEN, SIDDHARTHA R. DALAL, JESSE PARELIUS, AND GARDNER C. PATTON, Bellcore

Designing a system test plan is difficult and expensive. It can easily take several months of hard work. A moderate-size system with 100,000 lines of code can have an astronomical number of possible test scenarios. Testers need a methodology for choosing among them. The ISO 9000 process gives some guidance, specifying that each requirement in the requirements document must be tested. However, testing individual requirements does not guarantee that they will work together to deliver the desired functionality.

The combinatorial design method, a relatively new software testing approach, can reduce the number of tests needed to check the interworking of system functions. Combinatorial designs[1] are mathematical constructions widely used in medical and industrial research to construct efficient statistical experiments.[2] At Bellcore, we have developed a system called AETG (Automatic Efficient Test Generator),[3,4] which uses combinatorial design theory to generate tests. Several Bellcore and other groups are using the AETG system for unit, system, and interoperability testing.

0740-7459/96/$05.00 © 1996 IEEE

TABLE 1
RELATION FOR A VOICE RESPONSE UNIT

Values for voice response unit			
Announcement	**Digits wanted**	**Billing**	**Access type**
None	None	Yes	Line
Interruptible	Fixed	No	Trunk
Noninterruptible	Variable		
Constraint specifying prohibited test cases			
None	None	*	*

TABLE 2
TESTS FOR VOICE RESPONSE UNIT

Tests for voice response unit				
Test	**Announcement**	**Digits wanted**	**Billing**	**Access type**
1	None	Fixed	No	Line
2	None	Variable	Yes	Trunk
3	Interruptible	None	No	Trunk
4	Interruptible	Fixed	Yes	Trunk
5	Interruptible	Variable	Yes	Line
6	Noninterruptible	None	Yes	Line
7	Noninterruptible	Fixed	Yes	Trunk
8	Noninterruptible	Variable	No	Trunk

We performed experiments using AETG to test the effectiveness of the combinatorial design approach. We tested some standard Unix commands and found that the combinatorial design test sets gave good code coverage. In another experiment, we tested modules from two releases of a Bellcore product. The combinatorial design tests uncovered faults in the code and in the requirements that were not detected by the standard test process. In this experiment, the product's developers and requirement writers created the tests, demonstrating that practicing engineers can use the combinatorial design approach to test software effectively.

THE COMBINATORIAL DESIGN PARADIGM

To design a test plan, a tester identifies parameters that determine possible scenarios for the system under test (SUT). Examples of such test parameters are SUT configuration parameters, internal SUT events, user inputs, and other external events. For example, in testing the user interface software for a screen-based application, the test parameters are the fields on the screen. Each different combination of test parameter values gives a different test scenario. Since there are often too many parameter combinations to test all possible scenarios, the tester must use some methodology for selecting a few combinations to test.

In the combinatorial design approach, the tester generates tests that cover all pairwise, triple, or n-way combinations of test parameters specified in formal test requirements. Covering all pairwise combinations means that for any two parameters p_1 and p_2 and any valid values v_1 for p_1 and v_2 for p_2, there is a test in which p_1 has the value v_1 and p_2 has the value v_2.

Using the AETG system, the tester specifies a system's test requirements as a set of relations. Each relation has a set of test parameters and a set of values for each parameter. The set of possible test scenarios specified by the relation is the Cartesian product of the value sets for its test parameters. The tester specifies constraints between test parameters either by using multiple relations or by prohibiting a subset of a relation's tests. For each relation, the tester specifies the degree of interaction (for example, pairwise or triple) that must be covered. AETG then uses combinatorial designs to generate tests covering the specified degree of interaction.

Testers usually generate test sets with either pairwise or triple coverage. An empirical study of user interface software at Bellcore found that most field faults were caused by either incorrect single values or by an interaction of pairs of values. Our code coverage study also indicated that pairwise coverage is sufficient for good code coverage. The seeming effectiveness of test sets with a low order of coverage such as pairwise or triple is a major motivation for the combinatorial design approach.

Table 1 shows a relation with four parameters that defines test scenarios for a voice response unit, a device that plays voice announcements and collects dialed digits. We abstracted this example from a test plan generated for a telephone network service called AIN (Advanced Intelligent Network). The first parameter specifies the type of announcement to play. It has three values: *none*, *interruptible*, and *noninterruptible*. The second parameter specifies that the user should input no digits, a fixed number of digits, or a variable number terminated by the pound key. The third parameter specifies whether the voice response unit is to make a billing record for the transaction. The final parameter indicates whether the user is accessing the unit by a local phone line or a long-distance trunk. To prevent vacuous test scenarios, the relation has a constraint that prohibits any combination using *none* for the announcement parameter and *none* for the digits-wanted parameter.

Testing all combinations of the four parameters in Table 1 that satisfy the constraint requires 32 different possible test scenarios. The eight tests shown in Table 2 cover all pairwise combinations of the parameter values, for a 75 percent reduction. Complex testing situations often give more dramatic reductions. For example, 120 binary parameters require 10 tests for pairwise coverage, instead of 2^{120} for exhaustive testing.

TABLE 3
TEST SPECIFICATION OF TWO MONITOR COMMANDS

Command 1		Command 2		
Circuit pack	Active status	Circuit pack	Time unit	Threshold value
1	On	1	Day	Low
2	Off	2	Timed	High
	No change		Untimed	

TABLE 4
COVERAGE OF PAIRWISE TESTS FOR SORT

Model	Height	Width	Avoids	Cases	Block (%)	Decision (%)	C-uses (%)	P-uses (%)
A	10	13	0	126	92	80	75	72
B	4	23	0	41	95	86	77	75
Avg.					93.5	83	76	73.5

SYSTEM TESTING

The first and perhaps most important step in defining test requirements is to identify the test parameters. In general, the parameters for unit testing are low-level entities, such as screen fields or message components, to which the tester must assign values. The system documentation usually makes clear what the unit testing parameters should be. The way to define test parameters for system testing, however, is less obvious. The key to defining these parameters is to model the system's functionality, not its interface. This reduces the model's complexity and focuses the model on the system aspects important to testing.

Our test plan for a network performance-monitoring system serves as an example. We generated tests for two releases of this equipment. Creating the test requirements for the first release took one week; modifying it for the second release took an hour. Generating a test plan for the second release would normally take well over a month.

We based the test requirements on the system's user manual. The test requirements for the final release had a total of 75 parameters, with 10^{29} possible test combinations. The combinatorial design approach required only 28 tests to cover all pairwise parameter combinations for the 75 test parameters.

This equipment includes several monitors that track the number of corrupted ATM (asynchronous transfer mode) cells in transmission facilities. The system software contains commands to configure the monitors and display the collected data. To test the system, the tester enters a series of configuration commands and uses an attenuator to corrupt the transmission facility's ATM cells. Then the tester enters display commands to check that the system has correctly collected the data. The configuration commands and the attenuator setting specify all of each test's tester-controllable parameters.

Users interface with the system via screens on a data input terminal. The system has several screens for command entry and data display. The screens have several hundred fields. The tester enters a command's name on the first screen and then enters its arguments on the following screens. The command determines which screens follow the initial screen. For example, to give the command that turns a monitor on or off, the tester enters the command's name on the initial screen. The next screen displays each monitor's current on/off status. On this screen, the tester enters either *on* or *off*. The following screen confirms the changes.

Although the tester gives the system commands by typing into screens, the test relation parameters do not model screen fields; rather, they model the commands and the attenuator. Several test parameters determine each command's arguments. A back-end script translates the test parameters into the appropriate keystrokes.

Table 3 shows the test specification of two monitor commands. One command turns the monitor on and off. The other sets the time units and thresholds for the monitor's alarms. The full test specification of the two commands has a copy of these parameters for each monitor. The active status parameter has three values: *on*, *off*, and *no change*. The back-end script translates the test parameter value *on* into an *on* entry in the monitor's field on the status screen and a *yes* entry in the monitor's field on the confirm screen. The value *no change* becomes either a *yes* or a *no* entry in the monitor's field on the status screen and a *no* entry in the monitor's field on the confirm screen. The back-end script similarly translates the other test parameters.

COVERAGE MEASUREMENTS

Several recent studies indicate that good code coverage correlates with effectiveness.[5-7] We measured the coverage of combinatorial design test sets for 10 Unix commands: basename, cb, comm, crypt, sleep, sort, touch, tty, uniq, and wc. We used the ATAC test tool[8] to provide the block, decision, C-uses, and P-uses metrics for the test sets. The pairwise tests gave over 90 percent block coverage.

We performed several experiments on the sort command. We selected the sort command because, with 842 lines, it has substantially more code than the other commands. W.E. Wong and his colleagues[8] also experimented with the sort command. They generated a pool of 1,000 random tests from which they created multiple, distinct test sets with various amounts of block coverage. The maximum block coverage provided by a test set was 80-85 percent. The average number of tests in the 29 test sets with 80-85 percent block coverage was 26.97.

We based the sort command's test

TABLE 5
FAULT DETECTION EXPERIMENTS ON TWO RELEASES OF A BELLCORE PRODUCT

Release	Screens	Code defects	Requirement defects
1994	9	19	30
1995	15	15	29

requirements on the documentation in the user manual. Sort has many flags. Several can occur more than once in a command line. Some flags are incompatible. Others cause the test to bypass much of the code. For example, the -c flag specifies that the sort program should only check its input and not sort it. Any test with this flag will not execute the sort code. After studying the documentation, a tester would normally write a test plan including only a few tests with such flags. Since we wanted to automate the process as much as possible, we initially did not partition the flags to avoid these tests.

There are many ways to model the sort command. We tried several different models. Table 4 shows the coverage data for two of them. Our models differed from each other in two ways. First, we varied the number of parameters, which we call *width*, and the maximum number of values for a parameter, which we call *height*. In general, the number of pairwise tests is at least the product of the number of values for the two parameters with the most values. Consequently, rewriting a model to decrease the maximum number of values per parameter can decrease the number of tests, even if it increases the number of parameters.

The other factor we varied was whether or not values such as the -c flag were valid or invalid parameter values. A parameter's valid values are tested pairwise against the other parameters' valid values. Consequently, they occur in many tests. For example, if a parameter has three valid values, each will occur in about a third of the tests. A parameter's invalid values are not tested pairwise with the other parameter values but are tested only once. If a value is known to cause a visible fault, making it invalid can reduce the number of tests causing that fault. Although having many tests causing an easily checked fault may not greatly increase the testing cost, the fault may mask the effect of the test's valid values. Consequently, testing the invalid value pairwise with the valid values may decrease coverage.

In the first sort model, A in Table 4, we did not partition the sorting flags. Each of this model's four test parameters had the nine flags b, d, f, i, n, r, c, m, and u as values. The model covered 92 percent of the blocks and required 126 tests. It required so many tests because several parameters had many values. Many tests produced error messages saying that the program was trying to write to an illegal output file, such as "sort: can't create OUT2: Permission denied." One of the illegal output files was a directory; the other, a file without write permission. Since these tests probably bypassed much of the code, we wanted to rewrite the test model to reduce their number.

We made two changes to model A, creating an intermediate model not shown in Table 4. First, we made the two illegal output files invalid instead of valid parameter values. As a result, they would be tested individually and would not occur in so many tests. Second, we decreased the maximum number of values per test parameter—in other words, decreased the model's height. We did this by adding additional test parameters and distributing the flags among them. This model had 23 parameters and a height of 4. It required 37 valid tests and two invalid tests (one for each of the two invalid test parameters). It covered 86 percent of the code.

The intermediate model generated several tests that caused the sort program to core dump. Nine tests core dumped with a bus error message, and two with a segmentation fault message. Two input files caused the core dumps. We also had 13 tests that exited with the error message "sort: can check only 1 file." Since 24 of the 39 tests were causing these three faults, we decided to rewrite the model.

In the next model, B in Table 4, we made the two input files that caused the core dumps invalid instead of valid values. We also made the -c option an invalid value for the parameters specifying the sorting options. These changes did not affect the number of fields or the height, so, like the intermediate model, this model also had 23 test parameters and a height of 4.

Model B generated 41 tests: 29 pairwise tests of the valid values and 12 tests of the invalid values. The 29 valid tests gave 90 percent block and 82 percent decision coverage. Including the 12 invalid tests, the 41 tests for model B gave 95 percent block and 86 percent decision coverage. This is better coverage than that of the 126 tests for model A or the 39 tests for the intermediate model. It is also better than the random test coverage described by Wong and his colleagues.

These experiments illustrate an important optimization method for the combinatorial design approach. By making some values invalid instead of valid, we increased the block coverage from 86 percent to 95 percent while increasing the number of tests by only two, from 39 to 41 tests. We identified these invalid values either from the error messages produced by the sort program or from its user documentation.

By basing our optimizations on the program's error messages and user documentation, we took a "black box" approach to testing. If the source code is available, testers can also use "white box" optimization methods.

FAULT DETECTION

We used the combinatorial design method to test user interface modules from two releases of a Bellcore product. The product's developers and requirements writers wrote the test requirements based on the development requirements. Because we used Bellcore's Mynah sys-

tem to run the tests and compare the results automatically, we made little effort to minimize the number of tests. Some of the problems we found were faults in the code, and others were faults in the requirements. Finding faults in the requirements is important because the requirements play a part in maintenance and future development. Table 5 shows the details of our testing.

Because these tests are generated from high-level test requirements, they can be easily modified when the code is updated. In fact, some of the tests for the 1995 release were revisions of tests for the 1994 release.

When we ran our tests, the software was nearing the end of system test, and most defects had already been found. Nevertheless, the combinatorial design test sets found defects the normal test process missed. Several were so severe that the module needed extensive rewriting. One of the developers, who tested his own software, reported that the combinatorial design approach uncovered all the faults (nine code faults in three modules) that he had found during extensive unit testing.

COMPARISON OF METHODS

Because the combinatorial design testing approach is relatively new, the literature reports on only two test generation systems using it: Bellcore's AETG and CATS (Constrained Array Test System), developed by G. Sherwood at Bell Labs.[9] Although published descriptions of CATS applications and results are scant, the AETG combinatorial design algorithms seem to generate fewer test cases than CATS. For example, Sherwood reports that in its largest known run, CATS generated 240 test cases for all pairwise combinations of 20 factors with 10 values each. Mallows[10] reports that he reduced this by hand analysis to 206 tests. The AETG system requires only 180 tests for this configuration.

A related approach[11-13] is the use of orthogonal arrays, a pairwise combinatorial design with the additional property of requiring every pair of values to occur exactly the same number of times. This requirement is severe and makes orthogonal arrays impractical for testing software. For example, for 100 parameters with two values each, the orthogonal array requires at least 101 tests, while 10 test cases are sufficient to cover all pairs.

One developer reported that the combinatorial design approach uncovered all the faults he found during unit testing.

The combinatorial design approach differs from traditional input testing by giving testers greater ability to use their knowledge about the SUT to focus testing. Testers define the relationships between test parameters and decide which interactions must be tested. Using the combinatorial design method, testers often model the system's functional space instead of its input space.

The combinatorial design approach to automatic test generation combines an expressive interaction description language with algorithms that reduce the number of tests needed to cover the specified interactions. Testers can create effective and efficient test plans, often faster than by traditional methods entailing hand optimization. This approach can reduce the testing cost while preserving or increasing test quality. ◆

ACKNOWLEDGMENTS

We thank Chris Bailey, Mike Dobiesz, Peter Duchnowski, Eleanor Kaldon, and Ajay Kajla for their participation in the fault detection experiments. We also thank Michael Friedman, Mary Griffin, Charles Hallowell, Adam Irgon, and Isaac Perelmuter for their help. Finally, we thank Jon Kettenring, without whose support this project would not have begun.

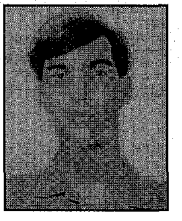
REFERENCES

1. M. Hall, Jr., *Combinatorial Theory*, Wiley Interscience, New York, 1986.
2. G. Taguchi, *Introduction to Quality Engineering*, Kraus Int'l Pubs., White Plains, N.Y., 1986.
3. D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, "The Automatic Efficient Test Generator," *Proc. 5th Int'l Symp. Software Reliability Eng.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 303-309.
4. D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The AETG System: A New Approach to Testing Based on Combinatorial Design," Tech. Report TM-25261, Bell Communications Research, Morristown, N.J., 1995.
5. P. Plwowarski, M. Ohba, and J. Caruse, "Coverage Measurement Experience During Function Test," *Proc. 15th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 287-303.
6. W.E. Wong et al., "Effect of Test Set Minimization on Fault Detection Effectiveness," *Proc. 17th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 41-50.
7. W.E. Wong et al., "Effect of Test Size and Block Coverage on Fault Detection Effectiveness," *Fifth Int'l Symp. Software Reliability Eng.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 230-238.
8. J.R. Horgan and S. London, "ATAC: A Data Flow Coverage Testing Tool for C," *Proc. IEEE Conf. Assessment of Quality Software Development Tools*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 2-10.
9. G. Sherwood, "Effective Testing of Factor Combinations," *Proc. 3rd Int'l Conf. Software Testing, Analysis and Review*, Software Quality Eng., Jacksonville, Fla., 1994.
10. C. Mallows, "Covering Designs in Random Environments," to be published in *Festschrift for J.W. Tukey*, S. Morgenthaler, ed., 1996.
11. R. Mandl, "Orthogonal Latin Squares: An Application of Experimental Design to Compiler Testing," *Comm. ACM*, Oct. 1985, pp. 1054-1058.
12. R. Brownlie, J. Prowse, and M. Phadke, "Robust Testing of AT&T PMX/StarMail Using OATS," *AT&T Technical J.*, Vol. 71, No. 3, 1992, pp. 41-47.
13. E. Heller, "Using Design of Experiment Structures to Generate Software Test Cases," *Proc. 12th Int'l Conf. Testing Computer Software*, ACM, New York, 1995, pp. 33-41.

David M. Cohen is a member of Bellcore's Information Sciences and Technologies Research Lab. His research interests are software engineering, simulation, and protocol engineering. He has also done research in number theory, combinatorics, and automata theory. He holds two patents.

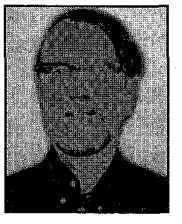
Cohen received a BA from Harvard University and a PhD in mathematics from MIT. He has held postgraduate research fellowships from the Institute for Advanced Study in Princeton, N.J., and the Alexander von Humboldt Foundation of Bonn, Germany. Cohen is a member of ACM and the IEEE Computer Society.

Siddhartha R. Dalal is chief scientist and a director of the Information Sciences and Technologies Research Lab at Bellcore. Besides initiating and leading the research on combinatorial designs and the AETG system, he has led research projects in software engineering, risk analysis, mathematics, and statistics. After receiving an MBA and a PhD from the University of Rochester, Dalal started his industrial research career at the Bell Labs Math Research Center. He received the American Statistical Association's 1988-89 award for an outstanding statistics application paper on his work in the Challenger disaster investigation on behalf of a National Research Council committee. Dalal 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*.



Jesse Parelius is a PhD student in statistics at Rutgers University. His research interests include signal processing and alternative centrality statistics.

Parelius received a BS in applied mathematics from the Massachusetts Institute of Technology.



Gardner C. Patton 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 over 10 million lines of code. He is currently interested in test measurement, automatic test generation, client/server testing, and intranet testing.

Patton received a BA in physics from Brown University and an MS in industrial engineering from the New Jersey Institute of Technology. He is a member of ACM.

Address correspondence about this article to Dalal at Bell Communications Research, 445 South Street, Morristown, NJ 07960; sid@bellcore.com.