Maintaining the Quality of Black-Box Testing

Patrick J. Schroeder
Illinois Institute of Technology

Dr. Bogdan Korel
Illinois Institute of Technology

Techniques to reduce cost and schedule can wreak havoc with product quality, especially in the software testing process. Shortsighted reductions can unwittingly result in the loss of long-term customers. In this article, the authors present an automated approach to reducing the cost of black-box testing while maintaining the quality of the testing process and software product. Experimental studies have shown a significant cost reduction using this automated approach.

Controlling cost, schedule, and quality in a software development project remains a challenging task. This type of control is difficult largely because of our inability to accurately measure attributes of the software development process, especially quality [1]. Measuring the quality of development processes and artifacts then relating them to final software product quality is not a process that is well understood. Under constant pressure to reduce cost and schedule, software engineers often use reduction techniques without fully understanding their impact on processes and final product quality.

Nowhere is this truer than in the software testing process. Software testing can improve software quality, but at a significant cost. It is not unusual for 40 percent to 80 percent of product development cost to be spent finding and fixing software errors [2]. Balancing testing cost and schedule with quality is difficult. Ad hoc reductions in the testing effort may bring short-term savings in cost and schedule. Unfortunately, these savings may be erased by quality problems discovered later in the product life cycle where fixes are expensive and a product's reputation may be badly damaged.

In this article, we present an approach that reduces black-box testing effort while maintaining the *quality* of the testing process. By quality, we refer to fault-detection capability of the test suite, or simply, how many faults a test suite uncovers. The type of software faults are those related to the correctness of program output; other types of software faults associated with other quality attributes such as performance, fault tolerance, usability, etc. are addressed in other testing process areas. Our goal is to create a *reduced* test suite that is smaller in size than the original test suite. The reduced test suite, however, is to maintain the fault detection capability of the larger, original test suite. This ensures that any savings in testing cost and schedule gained by executing the reduced test suite are not lost to expensive field repairs. To accomplish this, we perform additional analysis of the program under test. By using information gathered from the program specification and implementation, we can reduce the size of the original test suite in a controlled fashion that ensures that the original suite's fault detection capability is maintained.

While we believe our approach is applicable in a wide array of testing situations, in this article we focus on our initial efforts of applying our approach to black-box testing and the problem of combinatorial testing in data-driven programs. The following sections define the problem and current approaches; our approach, called Input-Output (IO) Analysis; details of implementing IO analysis; the results of experimental studies using our approach; and the conclusion.

Combinatorial Testing Techniques

In this article, we focus on black-box testing of data-driven programs. Black-box testing ensures that a program meets its specification from a behavioral or functional perspective and is typically performed without knowledge of software internals. Black-box testing can be applied to both control-driven and data-driven programs. In control-driven applications, outputs are determined by sequences of events or processing states. In data-driven applications, manipulation of data inputs and the relationships between data items determine outputs. Data driven applications, which typically have multiple inputs and outputs, include transaction-based systems, order-processing systems, application program interfaces, and many of the form-based applications common on the Web today.

Black-box testing techniques used for data-driven programs include equivalence-class partitioning and boundary value analysis [3]. Using these techniques, testers select test data values for each of the program's input variables. The tester must then consider how to test combinations of the selected test data values. It is important to test different input combinations, otherwise the result could be an unacceptable number of undetected software faults.

The most comprehensive approach to testing program-input combinations is referred to as combinatorial testing. In combinatorial testing, all possible combinations of the test data values selected for the program inputs are tested. (Combinatorial testing should not be confused with exhaustive input testing, which is testing every possible data value, valid and invalid, in every possible combination. This generates astronomically large test suites in all but trivial applications.) From a software quality perspective, combinatorial testing is desirable because it covers a large portion of the program's input space, resulting in fewer faults passed to the end-users of the product.

The challenge in combinatorial testing is managing the size of the test suite. In reality, combinatorial test suites grow very rapidly in size, frequently making the approach impractical. To fit within available resources and schedule, several approaches could be considered to reduce the size of the combinatorial test suite. Orthogonal arrays [4] and experimental design techniques [5, 6] have been suggested as ways of reducing the number of combinatorial tests. These techniques generate tests by combining test data for subsets of the input variables, e.g., tests may be generated for all pair-wise combinations of program inputs. These techniques do reduce the number of combinatorial

tests dramatically, but their impact on the fault-detection capability of the test suite, especially when compared to combinatorial testing, is unknown. Random sampling [7] may also be used to reduce the test suite, but it leads to a reduction in fault-detection capability. In the next section, we present an automated approach that reduces the combinatorial testing effort while maintaining the fault-detection capability of the combinatorial test suite.

Combinatorial Testing Using Input-Output Analysis

Our approach performs additional analysis of the program under test. Information gained from this analysis is used to create a reduced test suite that is smaller than the combinatorial test suite (a subset of the combinatorial test suite). The reduced test suite maintains the fault detection capability of the larger, combinatorial test suite. We refer to this analysis as Input-Output (IO) analysis because it identifies the relationships between a program's inputs and outputs. Combinatorial testing can then focus on those input combinations that affect a program output, rather than considering all possible input combinations.

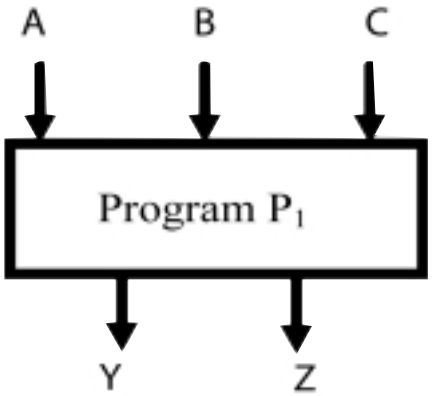


Figure 1: *Program P_1 -Multiple Inputs, Outputs*

For example, consider program P_1 in Figure 1. Program P_1 has three inputs (A, B, and C) and two outputs (Y and Z). To test this program using combinatorial testing, we would first select test data values for each of the program inputs using black-box test design techniques. Assume we select test data values for each of the three input variables, as in Table 1.

Table 1: *Selected Test Data Values for Program P_1*

Input Var.	Test Data Values
A	1, 2
B	North, South, East, West
C	TDC, BDM

Now consider how to test combinations of the input variables. For simplicity, assume there are no constraints among the input variables, and that we will execute combinatorial testing of the selected test data values. The combinatorial test suite generated from the test data values selected for the program inputs is listed in Table 2.

Our goal is to reduce the effort of combinatorial testing without reducing fault-detection capability. Our approach takes advantage of the observation that in data-driven programs not all inputs influence every program output. We find this to be a common occurrence in our study of this class of programs. We use IO analysis to identify relationships between program inputs and outputs, referred to as IO relationships. Based on the IO relationships combinatorial testing is reduced to testing only those input combinations that affect a program output.

For example, suppose we identify the IO relationships for program P_1 as in Figure 2. We find that output Y is a function of inputs A and C, and that output Z is a function of input B, only. Since Y is a function of inputs A and C only, these two inputs will be used to generate combinatorial tests for output Y. Since there are four unique combinations of the test data values selected for inputs A and B, this would result in four tests. Similarly, output variable Z is a function of input B only, so tests for output Z will be generated from the test data values selected for input B, only. Since there are four test data values selected for input B, this results in four additional tests for a total of eight tests. However, notice that the test sets created from the perspective of outputs Y and Z can be merged into a test suite of size four, as in Table 3. The advantage of our approach is apparent; the size of the combinatorial test suite has been reduced from 16 to four tests.

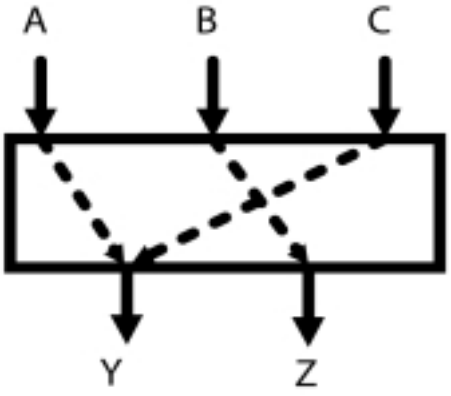


Figure 2: *Program P_1-Input/Output Relations*

Test ID	Input A	Input B	Input C
C_1	1	North	TDC
C_2	1	North	BDM
C_3	1	South	TDC
C_4	1	South	BDM
C_5	1	East	TDC
C_6	1	East	BDM
C_7	1	West	TDC
C_8	1	West	BDM
C_9	2	North	TDC
C_{10}	2	North	BDM
C_{11}	2	South	TDC
C_{12}	2	South	BDM
C_{13}	2	East	TDC
C_{14}	2	East	BDM
C_{15}	2	West	TDC
C_{16}	2	West	BDM

Table 2: *Combinational Test Suite Program P_1*

By only generating tests for the input combinations that influence a program output, a reduced test suite is created. However, has the fault detection capability of the combinatorial test suite been maintained? For data-driven programs, it is easy to show that the fault detection capability is maintained if the IO relationships are correct. For example, if the IO relationships for output Y are correct, any software fault that causes an incorrect value at output Y is detected by entering some combination of the A and C test data values. There are only four unique combinations of the test data values selected for inputs A and C. In the 16-test combinatorial test suite, these four unique combinations of A and C are unnecessarily repeated four times: once for each test data value selected for input B. Clearly, from the perspective of output Y, there are repetitive tests in the combinatorial test suite. Repeating a test does not increase the fault detection capability of the test suite; it only increases its size. Similarly, removing repetitive tests from the test suite does not reduce its fault detection capability; it only reduces the size of the test suite. A similar analysis can be done for output Z.

Table 3: *Reduced Test Suite for Program P_1*

Test ID	Input A	Input B	Input C
C_1	1	North	TDC
C_4	1	South	BDM
C_{13}	2	East	TDC
C_{16}	2	West	BDM

Implementation of Input-Output Analysis

IO analysis is used to determine the relationship between a program's inputs and outputs. The IO relationships are used to reduce a combinatorial test suite by removing tests that are repetitive from the perspective of the program outputs. The fault detection capability of the test suite is maintained because repeating tests will not expose any additional software faults.

To use IO relationships to reduce the number of tests, we must ensure that the IO relationships are correct. If we reduce the number of combinatorial tests using IO relationships that are incorrect, we run the risk of removing tests that could expose a software fault. This would result in the reduced test suite having a lower fault detection capability than the combinatorial test suite. Therefore, it is important that we validate the IO relationships before using them to reduce the number of tests.

We validate IO relationships in much the same way that we validate any other software function. We identify an *expected* set of IO relationships from the program specification, and we compare it to the *actual* IO relationships as implemented in the software. This process is analogous to comparing a test's expected result with the actual result obtained from a program.

The first step in validating the IO relationships is to analyze the software's specification to identify the expected IO relationships. If one is using formal specifications, or rigorous component specifications, the IO relationships may be explicitly stated, or may be easily derived in an automated fashion. For other specification techniques, some additional analysis may be required to develop the expected IO relationships.

The overhead associated with identifying the expected IO relationships should be small, because the relationships are already being identified at several points in the software development process. For instance, software developers must identify IO relationships in the process of implementing the program. One cannot write code to produce a program output without knowing which program inputs are used to create that output. Similarly, software testers create expected results for each of their tests. Again, one cannot determine the expected result of a test without knowing which program inputs are used to create an output. The software development process could be modified to record the expected IO relationships as they are encountered.

The next step in validating the IO relationships is to identify the actual IO relationships implemented in the software. To identify these relationships, static analysis or execution-oriented analysis can be used. Both of these techniques can be automated.

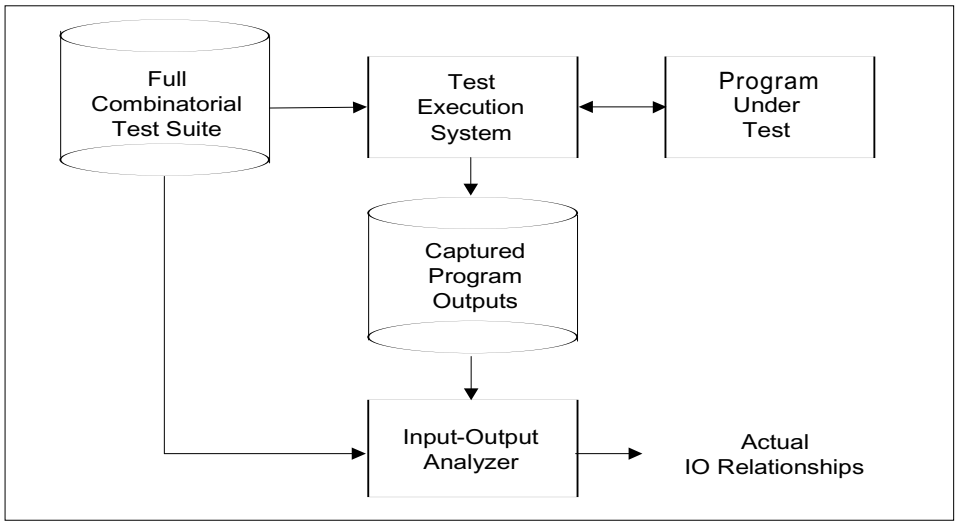
Static analysis is analysis of a program's source code. This analysis, accomplished using a source code specific tool, can produce information on a variety of the program's characteristics. One static technique used to determine relationships between inputs and outputs is referred to as Input-Output Relation Analysis [8]. This analysis uses a program dependence graph to determine which program inputs potentially influence a program output. Other program dependence analysis techniques used in code optimization, static slicing, and white-box testing, e.g. [9, 10, 11], may also be used to determine input-output relationships.

Execution-oriented analysis, as shown in Figure 3, is another technique that may be automated to identify actual IO relationships. This technique is based on program execution [12, 13] and does not require access to the program's source code. This technique does require, however, some type of test execution system, or test harness, to automated data entry and output capture. To determine relationships between inputs and outputs, the test harness executes the program under test many times altering only one input value. By observing changes in program outputs, it is possible to determine which outputs are affected by this input. This process can be repeated for all input variables in turn across a relatively large number of test data values. There is no guarantee that all IO relationships in a program will be detected using execution-oriented analysis. However, execution-oriented analysis can be used to identify all IO relationships exercised by a test suite such as the combinatorial test suite.

An advantage of execution-oriented analysis over static analysis is that execution-oriented analysis does not require the source code for the application. This makes execution-oriented analysis applicable in many testing situations, including testing of commercial off-the-shelf components. Execution-oriented analysis also has low startup costs, especially if test automation is already part of the test process. Another advantage to execution-oriented analysis is that the knowledge base and skill sets required to execute it closely match those of many software testers. Using static analysis to determine IO relationships, on the other hand, requires knowledge of the implementation's programming language and software internals.

The final step in validating the IO relationships is to compare the expected IO relationships identified in the program specifications to the actual IO relationships implemented in the software. This step is necessary if we are to maintain the fault detection capability of the reduced

Figure 3: *Execution-Oriented IO Analysis*



test suite. This is because the actual IO relationships could differ from the expected IO relationships. This difference could be due to different interpretations of the specification, to a fault in the software, or to complexities that arise during implementation that are not accounted for in the specification. To maintain the fault detection capability of the reduced test suite, these differences must be resolved before the IO relationships are used to reduce the combinatorial test suite

Experimental Studies

We have conducted several experimental studies using our approach. The goal of these was to understand the overhead involved with execution-oriented IO analysis and to determine the degree of reduction in testing effort.

We chose three different software applications to conduct the studies. The first application was the Total Return Report produced by a Windows-base personal financial software package. In testing a large system such as this, testing tasks are broken down by function area and assigned to testers. The Total Return Report represents a reasonably sized combinatorial testing task that may be assigned to an individual tester. The next application we studied was the Digital Trunk Configuration (DTC) software. This program provides an easy way to rapidly configure digital telephone trunks. The third application, the Liquidity Spreadsheet, was studied in conjunction with financial analysts who use spreadsheet programs extensively for financial modeling, reporting, and forecasting.

The procedure we used to conduct the experiments included the following steps. First, test data values for each of the application's input variables were selected using equivalence-class partitioning and boundary-value analysis. We then generated a combinatorial test suite from the test data values selected for each application using a test generation tool. The combinatorial test suite was executed automatically in a test execution system, and the outputs were captured. Finally, the inputs and outputs of each application were analyzed to determine the IO relationships. The reduced test suite was generated using these IO relationships. All studies were executed on a 250 megahertz Windowbase PC. Table 4 lists the results of the experimental studies. Sizes are reported as the number of tests.

Our results show a drastic reduction in the number of tests required for these applications. This reduction does not come at the expense of a reduced testing quality. The automated techniques used keep the overhead of IO analysis low, making it a valuable combinatorial testing technique in many situations.

In implementing IO analysis, several costs must be considered. The development effort required to create the tools to perform IO analysis (test data generator, test harness, and IO analyzer) in our case, was minimal; although we do not want to underestimate the effort required to automate program execution is some situations [14, 15]. Creating the test harness had low overhead, but test harness execution times could get quite long for some applications. For programs that use a command line interface, such as the DTC software, IO analysis executes very quickly. For programs with a Graphical User Interface, as used in the Total Return Report study, data entry is complex and relatively slow. The speed of this process could be improved by using a faster computer, or by distributing the work across multiple machines.

This leads us to comment briefly, on how IO analysis may be used in the testing process. We assume a software production process that includes multiple pre-release versions of the software. Each version may incorporate fixes and incremental enhancements. For programs with reasonable IO analysis execution times, it is feasible to run the analysis with every pre-release version of the software. Executing IO analysis on every version of the software ensures that any changes in the IO relationships from one version to the next would be detected. In this scenario, the savings to the test team are largely in the effort to calculate and check the expected result for each test case.

IO analysis executes a combinatorial test suite, but does not require determination of expected results, or comparison of expected results to actual program outputs. When IO analysis is used, the determination and comparison effort is only required of tests in the reduced test suite. In the case of the Liquidity Spreadsheet, the effort to determine and compare 1,058,841 expected results is reduced to the effort required for only 625 expected results. This represents a considerable saving to the test team considering that revalidation of expected results may be necessary with each pre-release version of the software.

For programs where data entry is complex and relatively slow, or where the combinatorial test suite is large, it will not be possible to execute IO analysis on every pre-release version of the software. In these situations, it is possible to execute IO analysis using a small subset of the combinatorial test suite. This subset can be constructed to validate the presence of the expected IO relationships in the software. The subset does not, however, ensure that other, possibly erroneous, IO relationships exist. IO analysis using this subset of tests could be executed on every pre-release version of the software. The reduced test suite created from this analysis will have a fault detection capability close to that of the combinatorial test suite. To ensure that the fault detection capability of combinatorial testing is maintained, IO analysis using a complete combinatorial test suite must be executed at some point. This would likely occur late in the product cycle when the software is stable and few additional software changes are expected.

Conclusion

We have presented an approach to combinatorial testing that reduces the testing effort while maintaining the fault detection capability of the combinatorial test suite. This approach can be used with little overhead for many applications especially where test automation is already in

Table 4 : *Results of Experimental Studies*

Application	Size of Combinatorial Test Suite	Size of Reduced Test Suite	Automated IO Analysis Time
Total Return Report	6,528	264	13.3 Hours
DCT Software	22,500	158	10.5 Minutes
Liquidity Spreadsheet	1,058,841	625	5.25 Hours

use. The experiment's results showed a drastic reduction in the testing effort of the data-driven programs. The low overhead associated with IO analysis and its ability to maintain the fault detection capability of the test suite makes it a valuable alternative to combinatorial testing.

The fault-detection capability of our approach is sensitive to correct identification of IO relationships. Automated IO analysis methods may not guarantee an identification of *essential* relationships in the presence of certain types of software faults. Therefore, it is important that software engineers validate automatically identified IO relationships in order to ensure their correctness. This extra effort may pay off later in reducing the testing effort while maintaining the quality of the reduced test suite. ◆

References

1. DeMarco, T., *Controlling Software Projects: Management, Measurement and Estimation,* Yourdon Press, Englewood Cliffs, N.J., 1982.
2. Kaner, C.; Falk, J.; and Nguyen, H. Q., *Testing Computer Software*, International Thomas Computer Press, New York, 1993.
3. Myers, G., *The Art of Software Testing*, John Wiley & Sons, New York, 1979.
4. Phadke, M. S., Planning Efficient Software Tests, CROSSTALK, Vol. 10, No. 10, pp. 11-15, October 1997.
5. Dalal, S.; Jain, A.; Karunanithi, N.; Leaton, J.; and Lott, C., Model-Based Testing of a Highly Programmable System, Proceedings of the International Symposium on Software Reliability Engineering, pp. 174-178, November 1998.
6. Burr, K., Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage, Proceedings of the International Conference on Software Testing, Analysis, and Review, October 1998.
7. Chen, T.Y. and Yu, Y.T., On the Expected Number of Failures Detected by Subdomain Testing and Random Testing, *IEEE Trans. Software Eng.*, Vol. 22, No. 2, pp. 109-119, February 1996.
8. Korel, B., The Program Dependence Graph in Static Program Testing, *Inform. Processing Let.*, Vol. 24, No. 2, pp. 103-108, January 1987.
9. Ferrante, J.; Ottenstein, K.; and Warren, J., The Program Dependence Graph and Its Use in Optimization, *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 5, pp. 319-349, 1987.
10. Horwitz, S.; Reps, T.; and Binkley, D., Interprocedural Slicing Using Dependence Graphs, *ACM Transaction on Programming Languages and Systems*, Vol. 12, No. 1, pp. 26-60, 1990.
11. Duesterwald, E.; Gupta, R.; and Soffa, M.L., Rigorous Data Flow Testing Through Output Influences, 2nd Irvine Software Symposium, Irvine, Calif., pp. 131-145, Mar. 1992.
12. Ferguson, R. and Korel, B., The Chaining Approach for Software Test Data Generation and Methodology, Vol. 5, No. 1, pp. 63-86, 1996.
13. Schroeder, P.J. and Korel, B., Black-Box Test Reduction Using Input-Output Analysis, Proceedings of the International Symposium on Software Testing and Analysis, pp. 21-24, August 2000.
14. Bach, J., Test Automation Snake Oil, *Windows Technical Journal*, Oct. 1996, pp. 40-44.
15. Fewster, M. and Graham, D., *Software Test Automation: Effective Use of Test Execution Tools*, Addison-Wesley, New York, 1999.

About the Authors

Bogdan Korel, Ph. D., is an associate professor of Computer Science at the Illinois Institute of Technology. His research interests include software engineering and automated software testing and analysis.

Illinois Institute of Technology
Department of Computer Science
10 W. 31st Street
Chicago, IL 60616
Phone: 312-567-5145
FAX: 312-567-5067
Email: korel@iit.edu

Patrick J. Schroeder is currently a Ph.D. candidate at the Illinois Institute of Technology involved in software testing research. Previously, he worked as a software developer and software tester on a variety of industrial projects, including seven years at AT&T Bell Laboratories. His research interests include software engineering, software reliability engineering, and automated software testing and analysis.

Illinois Institute of Technology
Department of Computer Science
10 W. 31st Street
Chicago, IL. 60616
Phone: 312-567-5150
Fax: 312-567-5067
E-mail: schrpat@iit.edu

"Computers are not meant to usurp human roles, but to aid an individual's work."
Donald D. Spencer

"Most industrialized nations today recognize that the computer is the cornerstone of their national defense and their national economics in the future."
C.W. Spangle

"Every educated person should have some understanding of the principles on which computers operate."
J. N. Snyder

"You know the definition of the perfectly designed machine ... The perfectly designed machine is one in which all its working parts wear out simultaneously. I am that machine."
Frederick A. Linderman