Automatic Generation of Test Scripts from Formal Test Specifications

Marc J. Balcer, William M. Hasling, and Thomas J. Ostrand

Siemens Corporate Research
755 College Road East
Princeton, New Jersey 08540

ABSTRACT: *TSL is a language for writing formal test specifications of the functions of a software system. The test specifications are compiled into executable test scripts that establish test environments, assign values to input variables, perform necessary setup and cleanup operations, run the test cases, and check the correctness of test results. TSL is a working system that has been used to test commercial software in a production environment.*

KEYWORDS: software testing, functional testing, automatic test generation, test specification, category-partition method.

INTRODUCTION

In [11], we introduced the category-partition method for analyzing a software product's functional specification. We also described a test specification language based on the method, and a system that transformed the test specifications into textual descriptions of test cases. Neither the method, the language, nor the system in [11] provided any means for specifying test case results. The method and language only dealt with the tested function's parameters and environments, and the system generated descriptions only of potential test case inputs and setups. Furthermore, since the original system produced only textual descriptions of test cases, additional manual work was needed to transform the tool's output into executable code.

We have now created a considerably enhanced test specification model that substantially automates the testing process. With this model, formal test specifications written in the Test Specification Language (TSL) are compiled into complete, executable test cases and test scripts. For each function of a software system, a test specification describes the function's inputs, the environment conditions (external conditions) that affect its behavior, the outputs that it produces, and the changes it makes to the environment. The test specification also describes which outputs and environment changes are expected to occur as a result of different combinations of inputs and environment conditions. Given this information, the TSL processor builds self-contained, executable test scripts. The scripts assign values to input variables, establish required external conditions, execute test cases and necessary cleanup operations, and compare the program's produced results to the expected results.

TSL also provides the capability of writing additional setup statements to create any needed data objects or test preconditions, cleanup statements to prepare for additional tests, and any result checking code that goes beyond direct comparison of outputs against expected results.

A key objective of TSL is to minimize the amount of tedious work for the human tester, both in the original script creation and in future test maintenance. This objective is realized in several ways:

- The test specification contains a maximal amount of descriptive information about the software, permitting TSL to generate complete test scripts directly from the test specification, without manual intervention by the human tester.
- Information that is common to several functions of a system (such as parameters, environment conditions, and result descriptions) need only be specified once, and can be shared by several test specifications. This facility simplifies the original creation of a test specification, and reduces error possibilities when a specification is modified.
- Part of a specification can be modified without destroying the usefulness of the unchanged part. The revised specification yields test scripts that contain both new test cases and the still relevant original cases.

FUNCTIONAL TESTING AND THE SPECIFICATION OF TESTS

In functional testing (as opposed to structural or internal testing), test cases are selected based on functional or external properties of the tested software and its input and output domains. Functional tests based on inputs are supposed to contain representative values from each set of inputs that can have a potentially different effect on the program's behavior. Tests based on outputs are supposed to produce results that include a member from each different type of output produced by the program.

It is frequently not obvious what constitutes "inputs that can have a potentially different effect" or "each different type of output." However, a common characteristic of functional test derivation methods is the construction of a partition of the program's input domain, with each class of the partition containing inputs that are presumed to be similar either in their effect on the program's behavior, or in the type of output they produce. There has recently been some controversy over the value of this type of testing. In particular, Hamlet and Taylor [7] claim that tests based on input partitions frequently are barely better for error detection than a random choice of tests over the entire input domain. We do not intend to take sides in this controversy, since we believe that both partition-based and random methods are useful and have their own benefits. In particular, part of the appeal of partition-based testing is that a well-defined partition of the input space, together with the corresponding required outputs, is itself a concise description of the program's functional behavior.

Note that a well-defined partition should include some input classes that consist solely of *error inputs.* These are inputs for which the program's requirements either do not specify an expected output, or else explicitly state that such inputs are errors. Executing tests from each class of such a partition exhibits the program's behavior over the entire range of its inputs, both expected and unexpected.

The category-partition method [11] is a systematic way of analyzing a system's functional specification to produce a well-defined partition of each function's input domain and a description of each function's expected results.

The tester starts applying the method by identifying each individual parameter and environment condition that affects the function's behavior, and then determines the significant properties of each one. These properties are called *categories*; their significance depends both on the input and on the function. Examples are *length of an input string* (to a string-processing program), *the number of jobs waiting to be assigned a partition* (to a memory allocator), and *the priority of an individual job* (to a process scheduler). The next step is to decompose each category into a set of mutually exclusive *choices.* The choices represent the set of possible values for the parameter or environment condition to assume in a test case.

Results are characterized by specifying outputs that are produced, and describing changes caused by the function to the system environment.

TSL provides a format for writing concise descriptions of the categories, choices, and expected results for a function, and *rules* that describe the combinations of inputs that can cause each result to occur. The TSL processor compiles the descriptions and rules into a set of test cases for the function.

The TSL system provides a number of benefits that are independent of the particular method of generating tests. The test specification can be viewed as a type of formal functional specification of the program, since it describes the relationship between inputs and outputs. As such, it can serve as an additional check of the program's behavior against informal requirements or specifications, or against another formal specification.

Since the TSL specifications are based entirely on the program's expected functional behavior, they can be written as soon as the software's specification is available, well before design or coding begin. One advantage of writing test specifications early is that the tester frequently discovers errors and inconsistencies in the functional specifications. The earlier these are unearthed, the better. Another obvious advantage is that the test cases will be ready for execution when the code is ready, which in turn implies that implementation schedule slips have a less serious impact on the testing schedule.

It is frequently noted that properly written test cases include both required inputs and expected results. Many testers, however, avoid writing down the results before they run tests. While TSL does not force the tester to describe results in advance, it provides a simple and convenient way to express them, and to write code for checking that a test's output satisfies its expected result.

TSL aids in the purely managerial and administrative aspects of testing software. By using TSL to write test cases, a project gets a complete and standardized record of the tests and their relation to the functions being tested. When all testers on a project use TSL, everyone can read and understand all of the test descriptions and test scripts. The format of test results is also standardized, making it easy for everyone to interpret the output of a test script.

Test maintenance and modification are exceptionally easy with TSL. Since the test specifications are organized according to software functions, a change to the functionality can be immediately traced to the corresponding place in its TSL specification, and the test information updated accordingly. Furthermore, many types of changes to the software require only a small change to bring the TSL specification in line with the new software. For example, if a new parameter is added to a system command, it is only necessary to add a description of the new parameter to the command's TSL specification, specify how test results are affected by the new parameter, and possibly modify a template for test cases to indicate how the new parameter is used in the command. The existing parameter descriptions remain unchanged. A new script produced from the modified test specification will automatically include all relevant combinations of the new parameter with the existing ones. The tester is relieved of the tedious chore of modifying a long series of existing test cases.

THE TEST SPECIFICATION LANGUAGE

A TSL test specification consists of a number of different sections that describe various aspects of the software and its environment; this information is combined and assembled by TSL into an executable test script. The separate sections make the description of a test script very economical, since each parameter and environment category need only be described once, but can be used in many different test cases. The separate sections also make it easy for a tester to modify or add to the information that controls the script. The general forms of the sections of a specification are shown informally in Figure 1.

A test specification contains one TEST section, and may contain any number of PARAMETER, ENVIRONMENT, and RESULT sections. In the Figure, the following notational conventions are used: Upper case strings are keywords. Pointed brackets (< >) denote a generic item that is replaced by an instance in the actual specification. (In a complete formal grammar, these would be non-terminals.) Square brackets ([ ]) surround an optional item.

The TEST section is a template for the code of the actual test cases that are produced; it is required to be in every test specification. The `<testname>` is a required identifier for the TEST section. `<description-string>` is an optional "comment" that can be used to describe the contents or purpose of the test. These comments are printed out when the tests are run. The actual code template is preceded by the keyword FORM, and can contain variables representing `<param-name>`s or `<environment-name>`s. When the FORM is instantiated, the variables are replaced by specific parameter or environonment values. Optional setup and cleanup code for any test cases that are produced, and special code to verify results that are expected to occur for every execution of the test case may also be included. (Verification code for individual results is included in the Results section.)

In each PARAMETER section, the `<param-name>` is the name of an input category of the tested function. `<choice-1>`, ... , `<choice-n>` are the names of choices of the category. Under each choice, the tester can optionally specify a list of specific values. If specific values are included, TSL constructs test cases with them. If specific values are omitted, TSL uses the name of the choice as the default value of the parameter. The categories in a PARAMETER section frequently are, in fact, parameters of an actual function or command of the tested system. However, they can also be characteristics of the tested command or function that the tester believes are important in defining an input partition.

Each ENVIRONMENT section similarly identifies environment condition categories and choices. Environment categories describe features of the system that require more initialization than merely assigning a value

```
TEST <test-name>
      [<description-string>]
      [SETUP {<string>}]
      FORM {<string>}
      [CLEANUP {<string>}]


PARAMETER <param-name>
      [<description-string>]
      [<setup-cleanup>]
            * <choice-1>
                  [<value-list>]
                  [<setup-cleanup>]
            . . .
            * <choice-n>
                  [<value-list>]
                  [<setup-cleanup>]


ENVIRONMENT <environment-name>
      [<description-string>]
      [<setup-cleanup>]
            * <choice-1>
                  [<setup-cleanup>]
            . . .
            * <choice-n>
                  [<setup-cleanup>]


RESULT <result-name>
      [<description-string>]
      [<setup-cleanup>]
      [VERIFY <verification-code>]
      IF <result-expression-1>
            . . .
      IF <result-expression-n>
```

FIGURE 1. General Form of Test Specification Sections

to a variable. Typical environment conditions are the status and contents of a file, the condition of the run-time stack, the amount of unallocated memory available, and the number of processes currently in existence. Because of these more general characteristics, the environment part of a test specification frequently contains setup information that must be executed before any test case that uses the environment.

In each RESULT section, the tester identifies an expected result that should be produced by the test cases, writes code to verify the result, and specifies the combinations of parameters and environment conditions that can produce the result. The `<verification-code>` for a result can be either actual executable code that runs after each test case and checks the post-test state of the system, or it can merely print certain values for the tester to examine visually. Setup and cleanup code can also be included in the specification of a particular result. Such code is executed, respectively, before and after each test case that produces the result.

The output of an individual test case always includes a description of the items (parameters, environments, and results) that make up the test. By default this description is the name of the item. However, the tester has the option of including in any PARAMETER, ENVIRONMENT, and RESULT section a `<description-string>` that is displayed instead of the default. These strings can give the tester reading the output of a script a more accurate description of the characteristics of the test case.

Each `IF <result-expression>` clause in a RESULT section contains a Boolean expression that characterizes when the result is expected to occur. In general, a result can be produced by many different combinations of parameter and environment values. Each combination of values that satisfies the `<result-expression>` in a RESULT section becomes the set of inputs of an individual test case for that result. The values are substituted for the parameter and environment variables in the TEST FORM to build the part of the test script that performs the actual test. If a given combination of parameter-environment values satisfies the `<result-expression>` for more than one result, then only one test case with that input combination is produced, but each of the results is verified after the test is executed. (The `SEPARATE` directive, discussed below, allows exceptions to the one-test-case rule.)

Two special directives can appear in a RESULT section. The directive `SINGLE` can be appended to any `IF` clause; it tells TSL to generate only one test case satisfying that clause. An example of using `SINGLE` is given below. The directive `SEPARATE` can be attached to a result-name; it causes TSL to generate separate test cases for each input combination that produces the result. `SEPARATE` is used for results that can only be verified destructively (i.e., by changing some aspect of the system's state), thus making it impossible to verify additional results of a test.

In general, the expressions specified in an `IF` clause do not constrain all possible parameter or environment categories. If a category is not mentioned in an `IF` clause, it means that the result is expected to occur regardless of the value of that category. TSL generates as many different test cases for that result as are necessary to include all the values that satisfy the clause.

When a test specification does not have a RESULT section, there are no restrictions on the test cases. TSL then generates tests that include all possible combinations of parameters and environment conditions.

The generation of test cases is illustrated briefly with the schematic test specification of Figure 2. Since category P has 3 choices and Q has 2, there are 6 possible test input combinations. However, the `IF` clauses generate four different input combinations that produce the three results, yielding the four test cases shown in Figure 3.

To show a more complete example of a test specification and how it is transformed into a test script, we use the example of an editor CHANGE command described by Myers [10, pp. 218-226]. Figure 4 is the English specification of the command, and Figures 5 and 6 show the corresponding TSL specification. The operating sys-

```
PARAMETER P
        * P1
        * P2
        * P3

PARAMETER Q
        * Q1
        * Q2

RESULT A
        IF (P = P1)
RESULT B
        IF (P ≠ P2) AND (Q = Q1)
RESULT C
        IF (Q = Q1)
```

FIGURE 2. Schematic Test Specification

Test	Input P	Input Q	Expected Result(s)
1	P1	Q1	A, B, C
2	P1	Q2	A
3	P2	Q1	C
4	P3	Q1	B, C

FIGURE 3. Test Cases Derived from Figure 2

Description:

The CHANGE command is used to modify a character string in the "current line" of the file being edited

Inputs:

The syntax of the command is:

C /string1/string2

String1 is the character string to be replaced. It can be from 1 to 30 characters long, and can contain any character except "/". String2 is the string that is to replace string1. It can be from 0 to 30 characters long and can contain any character except "/". If string2 is omitted (has 0 length), string1 is simply deleted from the current line.

At least one blank must follow the command name "C".

Outputs:

The changed line is printed on the terminal if the command is successful. If string1 does not exist in the current line, then "NOT FOUND" is printed. If the command syntax is incorrect, the message "INVALID SYNTAX" is printed.

System Transformations:

If the syntax is valid and string1 exists in the current line, then string1 is removed and string2 inserted in its place. If the syntax is invalid, or if string1 does not exist in the current line, then the line is not changed.

FIGURE 4. English specification for CHANGE command

tem commands are Unix. Although this example does not demonstrate all the features of TSL, it shows enough of the language to give a feel for its use.

In the example, *string1* and *string2* are the explicit parameters of the command. The tester has defined two additional parameter categories, *first-non-blank* and *internal-separator*. The only environment condition is the status of the current line, with respect to containing the search string *string1*. The SETUP code for the two choices for *current-line* builds a line with the desired property (either containing or not containing string1), and puts the line into a file called *testfile*. Note the use of the TSL variable *$string1* to refer to the possible values of the parameter.

This specification for CHANGE will generate a total of 38 test cases. Thirty cases come from each of the first two RESULT sections, and since their clauses have identical expressions, these two results are verified together after the execution of each of the thirty tests. The thirty different tests are derived by combining 5 values for string1 (all except length0) and 6 values for string2 (all listed values, since `string2` is not constrained in the expression). One of the test cases that produces these two results is shown in Figure 7. In the script for the test case, line 01 comes from the `SETUP` code of the environment `contains-string1`, lines 02–04 are the code from the `TEST FORM`, line 05 is the `description-string` from the result `replacement-done`, lines 06–10 are the `VERIFICATION` code for `replacement-done`, and line 11 is the `description-string` from the result `line-typed`. The latter result has no `VERIFICATION` code specified.

The `RESULT not-found` section generates 5 test cases, from the 5 values of `string1`. The tester decided that a single value for `string2` was sufficient to test this result.

In the `RESULT syntax-error` section, each `IF` clause generates only one test case because the directive `SINGLE` is attached to each. `SINGLE` tells TSL to generate only one test case that satisfies the clause. This feature is commonly used in testing error situations, where most of the components of the input are irrelevant to the outcome of the function. The particular value used by TSL for any of the unconstrained categories is usually the first value listed under that category. However, if an unconstrained category appears in another `IF` clause that produces the same result, then TSL uses a choice different from the one specified in the other `IF` clause. The reason is to assure that each test case contains only one distinct input value that is supposed to produce the result. (Myers [10] calls this technique *error-sensitizing*). Thus, the

```
IF first-non-blank = is-not-slash [SINGLE]
```

clause would generate a test case whose choices for each category are:

Category	Choice	Value
first-non-blank	is-not-slash	"z"
internal-separator	is-slash	"/"
string1	length1	"a"
string2	length1	"a"
current-line	contains-string1	

THE EVOLUTION OF TSL

In the original version of TSL, only parameter and environment conditions and relationships among them were specified. Specific values were not included, and no re-

```
TEST change command
      FORM {edit testfile
            C $first-non-blank $string1 $internal-separator $string2
            SaveExit}

PARAMETER first-non-blank
      * is-slash
            VALUE "/"
      * is-not-slash
            VALUE "z"

PARAMETER internal-separator
      * is-slash
            VALUE "/"
      * is-not-slash
            VALUE "x"

PARAMETER string1
      * length0
            VALUE ""
      * length1
            VALUE "a"
      * length2_29
            VALUE "ab"
            VALUE "abcd"
            VALUE "abcd987"
      * length30
            VALUE "abcdefghijklmnopqrstuvwxyz0123"

PARAMETER string2
      * length0
            VALUE ""
      * length1
            VALUE "a"
      * length2_29
            VALUE "ab"
            VALUE "abcd"
            VALUE "abcd987"
      * length30
            VALUE "abcdefghijklmnopqrstuvwxyz0123"

ENVIRONMENT current line
      * contains string1
            SETUP {echo "LINE-PREFIX $string1 LINE SUFFIX" > testfile}
      * does not-contain-string1
            SETUP {echo "LINE-PREFIX LINE-SUFFIX" > testfile}
```

FIGURE 5. Test Specification for CHANGE command (Parameters and Environments)

sults were specified. The system generated a file of permitted combinations of choices for each tested function, and the tester had to manually write an actual test case corresponding to each generated combination.

The current version of TSL improves upon the original in three significant ways. First, rather than producing only descriptions of the test cases, TSL now produces actual test cases, consisting of executable code that assigns values to inputs, sets up environment conditions, and carries out the action that is being tested. Second, TSL constructs test scripts that contain a sequence of individual test cases, together with setup and

```
RESULT replacement-done
      DESCRIPTION {String1 found in current-line, replacement done.}
      VERIFY {
            echo "LINE-PREFIX" $string2 "LINE-SUFFIX" > replaced_testfile
            if ('cmp -s testfile replaced_testfile')
                  then echo "correct-replacement"
                  else echo "incorrect-replacement"
            endif  }
      IF first-non-blank = is-slash AND internal-separator = is-slash
      AND string1 ≠ length0 AND current-line = contains-string1

RESULT line-typed
      DESCRIPTION {String1 found, type changed line.}
      IF first-non-blank = is-slash AND internal-separator = is-slash
      AND string1 ≠ length0 AND current-line = contains-string1

RESULT not-found
      DESCRIPTION {String1 not found in current-line, no replacement done.}
      VERIFY {
            echo "LINE-PREFIX" $string1 "LINE-SUFFIX" > nochange_testfile
            if ('cmp -s testfile nochange_testfile')
                  then echo "correct: no replacement done"
                  else echo "incorrect: current-line was changed"
            endif  }
      IF first-non-blank = is-slash AND internal-separator = is-slash
      AND string1 ≠ length0 AND current-line = does-not-contain-string1
      AND string2 = length1

RESULT syntax-error
      DESCRIPTION {Syntax error in command, do not attempt to execute.}
      IF first-non-blank = is-not-slash [SINGLE]
      IF internal-separator = is-not-slash [SINGLE]
      IF string1 = length0 [SINGLE]
```

FIGURE 6. Test Specification for CHANGE command (Results)

cleanup code that can apply either to the entire script, to a particular result, to a single test case, or to specific parameters and environments for a test case. Third, results can be specified in TSL, and verification code is automatically inserted into test scripts.

Use of TSL is not restricted to testing of programs written in any particular source language, or run under any particular operating system. Potential applications include any type of system for which a functional specification exists, and where parameters and environment conditions can be identified. The category-partition method can be applied at any level of software testing, including unit, module, integration, and system testing. TSL has been used to test the commands of a software management system and software for a process control system. Other applications that have been considered include generating test programs for a compiler, and generating test scripts for an interactive debugger.

RELATED METHODS AND TOOLS

The two main emphases of the TSL work to this point have been the category-partition method of analyzing a specification, and the integration into a formal test specification of all the information needed to produce a complete, executable, and self-verifying test script. Our work thus includes both a method for designing tests as well as a formalism and tool for carrying out the method.

The category-partition method creates functional tests by forming combinations of the many different potential inputs and environment conditions that can influence the way a function behaves. Various other similar functionality-based test derivation methods have been described in the literature, including *cause-effect graphing* [5, 10], *condition tables* [6], and *functional testing* [9]. Two other test derivation methods, *revealing subdomains* [13], and *equivalence partitioning* [12], use both the specification and the program code to derive input

Selected Parameters and Environments

Category	Choice	Value
first-non-blank	is-slash	" "
internal-separator	is-slash	" "
string1	length2-29	"abcd"
string2	length1	"a"
current-line	contains-string1	

Expected Results

```
replacement-done
line-typed
```

Script for Test Case

```
echo  'LINE-PREFIX' 'abcd'  'LINE-SUFFIX'  > testfile    -- Build test file
edit testfile -- Enter editor
C   /abcd/a          -- Issue CHANGE command
SaveExit              Save file  Exit editor
echo "Verification: String1 found in current-line. replacement done."
echo 'LINE-PREFIX'  a   'LINE-SUFFIX'  > replaced_testfile
                                         -- Build checking file
if ( cmp -s testfile replaced_testfile )   -- Verify replacement done
        then echo  correct-replacement
        else echo 'incorrect replacement
endif
echo  Verification: String1 found, type changed line.
```

FIGURE 7. Test Case for CHANGE Command

test classes. Descriptions of these methods and their relation to the category-partition method are given in [11].

The idea of writing a machine-processable formal test specification has been considered by several authors.

Duncan [3] suggested a context-free grammar that could generate sets of test inputs for a program in ever-increasing levels of complexity. These grammars, however, only produce input strings; it is up to the human tester to derive the corresponding outputs, create checking code, and integrate inputs, outputs, and code into an executable script.

Bauer and Finger [1] implemented a method of deriving a regular grammar from a formalized functional specification of the tested system's behavior, and a tool that generates test cases from the grammar. The generated test cases are complete; that is, they consist of both input sequences, and the corresponding expected system responses and outputs. Part of the system is an Automatic Test Executor, which transforms the generated test sequences into executable test input for the system under test, supplies the input to the system, and verifies the system's response. The technique can be used for systems that are formally specified using an augmented finite state machine model, such as process control and transaction processing systems.

Duncan and Hutchison [4] described, but did not implement, a method that uses attribute-augmented context-free grammars to describe the form of inputs and outputs of a program. The grammar functions both as a formal description of the expected behavior of the program, and as a test case generator. The attributes provide context-sensitive information that relates inputs and outputs, and allows a generator to produce semantically correct test cases. In addition, the attributes can be used to control the type and size of test cases that are produced.

Homer and Schooler [8] describe independent testing of the intermediate phases of a compiler through the use of a *test generator generator* (TGG) that transforms an attribute grammar into a program that generates test cases. Since their interest was specifically compiler testing, they were satisfied to produce outputs that consisted of internal representations of the output of the compiler phases. Although Homer and Schooler mention that TGG "can be used to produce compilable, executable, and even self-checking test programs", they do not make use of these capabilities, and their paper does not describe how TGG builds complete, executable scripts.

A more specialized approach to test case generation was taken by Bird and Munoz [2]. They discuss techniques that can be used to build special-purpose test script generators, and characterize the common properties that such generators share. They apply the techniques to building generators for testing several different types of program, including PL/1 compilers, a graphics display manager, and sort/merge routines. Bird and Munoz emphasize that although concepts of test case generation are similar for different types of software, each test case generator is a specific tool that must be coded separately. They have not built a general-purpose program or system for analyzing a test specification, and in fact, the concept of a test specification does not appear in their work.

REFERENCES

1. Bauer, J.A. and Finger, A.B. Test plan generation using formal grammars. In *Proceedings of the Fourth International Conference on Software Engineering*. (September 1979, Munich). IEEE Computer Society, 425-432.
2. Bird, D.L. and Munoz, C.U. Automatic generation of random self-checking test cases. *IBM Systems Journal* 22, 3 (March 1983), 229-245.
3. Duncan, A.G. Test grammars: A method for generating program test data. In *Digest for the Workshop on Software Testing and Test Documentation*. (Dec. 1978, Ft. Lauderdale, Florida). IEEE, 270-283.
4. Duncan, A.G. and Hutchison, J.S. Using attributed grammars to test designs and implementations. In *Proceedings of the Fifth International Conference on Software Engineering*. (March 1981, San Diego, CA). IEEE, 170-177.
5. Elmendorf, W.R. Functional analysis using cause-effect graphs. In *Proceedings of SHARE XLIII* (1974, New York). Share.
6. Goodenough, J.B. and Gerhart, S.L. Toward a theory of test data selection. *IEEE Trans. Softw. Eng. SE-2*, 2 (June 1975), 156-173.
7. Hamlet, D. and Taylor, R. Partition testing does not inspire confidence. In *Proceedings of the Second Workshop on Testing, Verification, and Analysis* (July 1988, Banff, Alberta). IEEE, 1988.
8. Homer, W. and Schooler, R. Independent testing of compiler phases using a test case generator. *Software—Practice and Experience 19*, 1 (January 1989), 53-62.
9. Howden, W.E. Functional Program Testing. Technical Report DM-146-IR, University of Victoria, Victoria, B.C., Canada, August 1978.
10. Myers, G.J. *Software Reliability: Principles and Practices*. John Wiley, New York, 1976.
11. Ostrand, T.J. and Balcer, M.J. The category-partition method for specifying and generating functional tests. *Communications of the ACM 31*, 6 (June 1988), 676-686.
12. Richardson, D. and Clarke, L. A partition analysis method to increase program reliability. In *Proceedings of the Fifth International Conference on Software Engineering*. (March 1981, San Diego, CA). IEEE, 244-253.
13. Weyuker, E.J. and Ostrand, T.J. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering SE-6*, 3 (May 1980), 236-246.