SPECIAL ARTICLE

THE CATEGORY-PARTITION METHOD FOR SPECIFYING AND GENERATING FUNCTIONAL TESTS

A method for creating functional test suites has been developed in which a test engineer analyzes the system specification, writes a series of formal test specifications, and then uses a generator tool to produce test descriptions from which test scripts are written. The advantages of this method are that the tester can easily modify the test specification when necessary, and can control the complexity and number of the tests by annotating the tests specification with constraints.

THOMAS J. OSTRAND and MARC J. BALCER

The goal of functional testing of a software system is to find discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification. To achieve this goal requires first, that tests be executed for all of the system's functions, and second, that the tests be designed to maximize the chances of finding errors in the software. Although a particular method or testing group may emphasize one or the other, these two aspects of testing are mutually complementary, and both are necessary for maximally productive testing. It is not enough merely to "cover all the functionality"; the tests must be aimed at the most vulnerable parts of the implementation. For functional testing, this implies testing boundary conditions, special cases, error handlers, and cases where inputs and the system environment interact in potentially dangerous ways. Conversely, it is not enough to write excellent, error-exposing tests for only some of a system's functions, or to write tests aimed only at certain types of errors or certain characteristics of a specification or implementation.

A preliminary version of this paper appeared as "Machine-Aided Production of Software Tests," in the Proceedings of the Fifth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 19–20, 1987.

Functional tests can be derived from the software's specifications, from design information, or from the code itself. All three test sources provide useful information, and none of them should be ignored. Code-based tests relate to the modular structure, logic, control flow, and data flow of the software. They have the particular advantage that a program is a formal object, and it is therefore easy to make precise statements about the adequacy or thoroughness of code-based tests. Design-based tests relate to the programming abstractions, data structures, and algorithms used to construct the software. Specification-based tests relate directly to what the software is supposed to do, and therefore are probably the most intuitively appealing type of functional tests.

A standard approach to generating specification-based functional tests is first to partition the input domain of a function being tested, and then to select test data from each class of the partition. The idea is that all elements within an equivalence class are essentially the same for the purposes of testing. If the testing's main emphasis is to attempt to show the presence of errors, then the assumption is that any element of a class will expose the error as well as any other one. If the testing's main emphasis is to attempt to give confidence in the software's correctness, then the assumption is that correct results for a single element in a class will provide confidence that all elements in the class would be processed correctly.

Various methods of creating a test partition are discussed in the literature [1, 5, 6, 9, 10]. Despite the general agreement on this key technique of functional testing, and its use since the earliest days of computing, the partitioning process lacks a systematic approach. According to Howden [7],

> ... there have traditionally been no firm guidelines for the application of the method [of functional testing] ... Dissatisfaction with the fuzziness of functional testing was partly responsible for the development of systematic approaches to testing which may be inadequate in terms of the errors they are capable of discovering, but which have the advantage that they are well defined.

In this article we describe a systematic, specification-based method that uses partitioning to generate functional tests for complex software systems. Our approach, called the *category-partition method*, includes the use of formal test specifications and is supported by a generator tool that produces test case descriptions from test specifications. The method goes through a series of decompositions, starting with the original functional specification, and continuing through the individual details of each subprogram being tested.

The main characteristics of the category-partition method include the following:

A. The test specification is a concise and uniform representation of the test information for a function.
B. The test specification can be easily modified, if this is necessitated by
   - changes in the functional specification of a command;
   - mistakes in an original test specification;
   - a desire for more or fewer test cases.
C. The test specification gives the tester a logical way to control the volume of tests.
D. The generator tool provides an automated way to produce thorough tests for each function, and to avoid impossible or undesirable combinations of parameters and environments.
E. The method emphasizes both the specification coverage and the error detection aspects of testing.

A STRATEGY FOR TEST CASE GENERATION

Although formal requirements and specification languages [2, 3] are becoming more widely used, many software specifications are still written in natural language. These documents are frequently wordy and unstructured, making them difficult to use directly as a basis for functional testing. The tester must transform such a specification into some more concise and structured intermediate representation, from which test cases and test scripts can then be generated.

Not only do natural language specifications lack structure, but they are frequently incomplete, ambiguous, or self-contradictory. The process of transforming the specification into an intermediate representation can be useful for revealing such problems. Frequently, the tester has to ask the specification writer to clarify the intention of a particular section or sentence. These questions are themselves a form of testing, as they may expose errors in the specification before any code is written or in the implementation before any code is executed.

The first steps of the category-partition method help to accomplish specification coverage and analysis. The later steps provide information used for generating error-detecting tests. The method begins with the decomposition of the functional specification into *functional units* that can be tested independently. A functional unit can be either a top-level user command or a function described in the specifications that is called by other functions of the system. For a very large or complex specification, this decomposition may require several stages, beginning with the identification of major subcomponents that can be executed independently, and leading eventually to the individual functional units. The process is very similar to the high-level decomposition done by software designers. In fact, a carefully done software design decomposition could serve as the basis for the test decomposition, and could save considerable duplication of effort. An advantage of an independent test decomposition, however, is that it provides a distinct and fresh point of view of the specification. Independent decomposition is highly recommended whenever error detection is a high priority goal, since it greatly increases the chances of finding problems in the specification, and also increases the chances that the resulting decomposition will lead to more effective tests for the software.

After the functional units of a subcomponent have been identified, the next decomposition step is to identify the *parameters* and *environment conditions* that affect the function's execution behavior. Parameters are the explicit inputs to a functional unit, supplied by either the user or another program. Environment conditions are characteristics of the system's state at the time of executing a functional unit.

The actual test cases for a functional unit are composed of specific values of the parameters and environment conditions, chosen to maximize the chances of finding errors in the unit's implementation. To select these values, the next step in the decomposition process finds *categories* of information that characterize each parameter and environment condition. A category is a major property or characteristic of a parameter or environment condition. Selecting the categories is done by careful reading of the specification. For each parameter or environment condition, the tester marks phrases in the specification that describe how the functional unit behaves with respect to some characteristic of the parameter or environment condition. Each characteristic that can be identified in this way is recorded as a category. Ambiguous, contradictory, or missing descriptions of a function's behavior are frequently discovered during this process.

Deriving the categories is the final step of analyzing the specification for coverage purposes. The next decomposition step is to partition each category into distinct *choices* that include all the different kinds of values that are possible for the category. Each choice within a category is a set of similar values that can be assumed by the type of information in the category. The choices are the partition classes from which representative elements will be taken to build test cases.

The concepts of category and choice can be illustrated with an example of a sorting program specification. The specification describes the expected input as a variable length array of values of arbitrary type, and the expected output as a permutation of the input, with the values sorted according to some total ordering criterion. The specification also requires that the program produce separate outputs containing the minimum and maximum values found in the input array. Based on this specification, the tester identifies the following categories: the array's size, the type of the elements, the maximum element value, the minimum element value, and the positions in the array of the maximum and minimum values.

While the categories are derived entirely from information in the specification, the choices can be based on the specification, the tester's past experience in selecting effective test cases, and knowledge of likely situations for errors to occur. If the code is available for analysis, the tester can also base the choices on the program's internal structure. In the sorting program example, one possible way to partition the category *array size* into choices is size = 0, size = 1, $2 \leq$ size ≤ 100, and size > 100. These choices are based primarily on experience with likely errors. If the tester happens to know that memory for variable size arrays is allocated in blocks of size 256, then it might be wise to include choices of size < 256, size = 256, and size > 256.

Once the categories and choices have been established, they are written into a formal *test specification* for each functional unit. The test specification consists of a list of the categories, and lists of the choices within each category. The information in a specification is used to produce a set of *test frames* that are the basis for constructing the actual test cases. A test frame consists of a set of choices from the specification, with each category contributing either zero or one choice. An actual test case is built from a test frame by specifying a single element from each of the choices in the frame. In the sorting program example, the *array size* category could contribute any of four choices to a test frame. For each of the first two choices, there is only a single element to specify for the test case. The third choice allows any size between 2 and 100 inclusive to be specified, while the fourth choice allows any array size over 100 to be specified. The tester placed 0 and 1 into separate partition classes of the array size category because he wanted to assure that arrays of those specific sizes became part of some test cases. Placing 2–100 into a single partition class indicates the belief that a test case with any of these sizes is essentially as good as one with any other size in that range.

A test specification as described so far is an *unrestricted specification*, since there are no relations indicated among any of its choices. Each test frame generated from an unrestricted specification contains exactly one choice from each category. The total number of frames generated from an unrestricted specification is equal to the product of the number of choices in each category.

Since the choices in different categories frequently interact with each other in ways that affect the resulting test cases, the choices in a test specification can be annotated with *constraints* that indicate these relations. A typical constraint says that a choice from one category cannot occur together in a test frame with certain choices from other categories. Constraints are used to refine the unrestricted set of test frames to a set considered both technically effective and economically feasible. The final determination of an appropriate set of test cases is a pragmatic decision that depends on the criticality of the software, the project's budget, and perhaps, on the skill levels of the programmers and the testers.

After a test specification has been annotated with constraints, it is processed by a *generator tool* that produces the test frames. Since this detailed combining of information is done by an automated tool, it is relatively easy for the user to make changes to the test specification, and then rerun the tool to produce a revised set of test frames.

The final steps in the test production process are to transform the generated test frames into test cases, and then to combine the cases into test scripts. Information from the parameter categories of a test frame determines the choice of specific inputs for a test case. The environment categories guide the establishment of the system state when the test case is run.

A *test script* is a sequence of related test cases for one or more functional units. The tester must decide whether test cases in a script should be independent, or share common environment settings. Since each case requires that its environment be set up before executing the command, an efficient approach groups test cases that require the same environment into a single script. This technique is most advantageous for test cases that do not modify their environment. Another approach is to use the result of case K in the script to establish the environment for case $K + 1$. With both of these methods of test script construction, if a test case does not perform as expected, it is possible that the remainder of the script is useless. A safer method is to set up each test case's environment individually, just before executing the test.

We now summarize this discussion in a series of steps that constitute the category-partition method. The section entitled **A Test Specification Language and Tool** describes these steps in more detail, and the method is illustrated with an example.

The category-partition method steps are as follows:

A. **Analyze the specification.** The tester identifies individual functional units that can be separately tested. For each unit, the tester identifies:
   - parameters of the functional unit;
   - characteristics of each parameter;
   - objects in the environment whose state could affect the functional unit's operation;
   - characteristics of each environment object.

   The tester then classifies these items into *categories* that have an effect on the behavior of the functional unit.

B. **Partition the categories into choices.** The tester determines the different significant cases that can occur within each parameter/environment category.

C. **Determine constraints among the choices.** The tester decides how the choices interact, how the occurrence of one choice can affect the existence of another, and what special restrictions might affect any choice.

(Step **C** and the following two steps frequently occur repeatedly as the tester refines the test specification to achieve the desired level of functional tests.)

D. **Write and process test specification.** The category, choice, and constraint information is written in a formal Test Specification. The written specification is then processed by a generator that produces a set of test frames for the functional unit.

E. **Evaluate generator output.** The tester examines the test frames produced by the generator, and determines if any changes to the test specification are necessary. Reasons for changing the test specification include the absence of some obviously necessary test situation, the appearance of impossible test combinations, or a judgment that too many test cases have been produced. If the specification must be changed, Step D is repeated.

F. **Transform into test scripts.** When the test specification is stable, the tester converts the test frames produced by the tool into test cases, and organizes the test cases into test scripts.

A TEST SPECIFICATION LANGUAGE AND TOOL

The Test Specification Language (TSL) is the means of implementing the category-partition strategy described in the previous section. The tester specifies *categories* of environments and input values, partitions each category into a set of mutually exclusive *choices*, and describes *constraints* that govern the interactions between occurrences of choices from different categories. This information is written in a formal *test specification* that can be processed by a test generator tool.

We illustrate the use of TSL and the generator tool with a simple **find** command that searches to occurrences of a pattern in a file. The command's natural language specification is shown in Figure 1.

Step A: Analyze the specification.
The **find** command is an individual function that can be separately tested. Its parameters are *pattern* and *file*. The specification mentions four characteristics of the pattern parameter:

- the length of the pattern;
- whether the pattern is enclosed in quotes;
- whether the pattern contains embedded blanks;
- whether the pattern contains embedded quotes.

There are several additional characteristics of the pattern that are *not* explicitly mentioned in the specification, but that a careful tester might wish to include as part of the information from which to construct tests. These include:

- whether a quoted search pattern always has to include a blank character;
- whether several successive quotes can be included in a pattern;
- the possibly ambiguous meaning of a leading quote in the pattern: is it used to quote the entire pattern, or is it used to insert a literal quote character?

The file can be considered both as a parameter of the **find** operation and as an object in the environment. We take the view that properties relating to the *name* of the file are parameter characteristics, while properties that have to do with the *contents* of the file are environment characteristics. In terms of the test cases produced by the category-partition method, this distinction is not important, since the generator tool treats both types of

Command:
find

Syntax:
find <pattern> <file>

Function:
The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

Examples:
`find john myfile`
displays lines in the file **myfile** which contain **john**

`find "john smith" myfile`
displays lines in the file **myfile** which contain **john smith**

`find "john"" smith" myfile`
displays lines in the file **myfile** which contain **john" smith**

FIGURE 1. Natural Language Specification for FIND Command

characteristics the same way. The main reason to emphasize both the parameter and environment characteristics is to remind the tester to think of both when analyzing the specification.

When *file* is considered as a parameter, its significant characteristic is whether or not it names an existing file. When *file* is considered as an object in the environment, its characteristics are:

- number of occurrences of the pattern in the file;
- number of occurrences of the pattern in a line that contains it (in the test specification, we call such a line a *target line*.);
- maximum line length in the file.

These characteristics come directly from the specification. Again, it is possible to consider additional unspecified characteristics of the file that could affect the behavior of **find**. Examples are the file type (text, binary, executable, etc.), whether the pattern overlaps itself in a line of the file, and whether the pattern extends across more than one line.

```
# Unrestricted Test Specification for FIND command

Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted

Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one
        more than one
```

FIGURE 2. Category Partitions for FIND

To keep the size of the example reasonable, we use only the characteristics that are directly mentioned in the specification. We also omit the third file characteristic, maximum line length. We use the five parameter characteristics *pattern size*, *quoting*, *embedded blanks*, *embedded quotes*, and *file name*, and the two environment characteristics *number of occurrences of pattern in file* and *pattern occurrences on target line*. These characteristics are the categories of the test specification that are partitioned in the next step of the method.

Step B: Partition the categories.
In partitioning the categories, it is important to include any situation that might reasonably occur, plus at least some that might not be expected. The latter type of situation will become the basis for error tests for the function. For the **find** function, the categories and choices are shown in Figure 2. It should be noted that the categories are divided into the two broad groups of Parameters and Environments. Lines beginning with # are comments.

The information in Figure 2 could serve as an unrestricted test specification. However, it will generate 1,944 test frames, many of which express situations with contradictory requirements. The frame in Figure 3 is an example. According to this frame, the pattern parameter is supposed to be an empty string that contains several embedded blanks. In addition, there are supposed to be no occurrences of the pattern in the file, but one occurrence on the target line.

```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

FIGURE 3. Test Frame with Contradictory Requirements

Step C: Determine constraints among the choices.
Contradictory test frames are produced because the generator combines choices without any regard to their meaning and how they might interact. To eliminate the contradictory frames, constraints on the use of choices are added to an unrestricted specification. Constraints are indicated with *properties* that can be assigned to certain choices, and tested for by other choices. Each individual choice in a test specification can be annotated with a *property list* and a *selector expression*. The properties assigned to a choice are associated with every test frame that includes the annotated choice. The form of a property list is

```
[property A, B, ...]
```

where A, B, ... are individual property names.

A selector expression is a conjunction of properties that were previously assigned to other choices. An expression tells the test frame generator not to include the annotated choice in a test frame unless the properties in the expression are associated with choices that are already in the frame. A selector expression is assigned to a choice by appending a suffix of the form

`[if A]`

or

`c-if A and B and ...]`

to the choice.

Figure 4 shows the **find** command test specification annotated with property lists and selector expressions. Since we want to distinguish empty patterns from non-empty ones, the two properties `Empty` and `NonEmpty` are assigned to the appropriate choices of the category `Pattern size`.

When the generator encounters a choice with a selector expression, it omits that choice from any partial test frame that does not already have the properties specified in the selector expression. For example, all of the choices in the categories `Embedded blanks` and `Embedded quotes`, as well as the last two in `Quoting` would never be combined with the choice `Pattern size: empty`. In addition, the second and third choices in `Embedded blanks` would only be combined with a tuple that contains the choice `Quoting: pattern is quoted`.

A choice may be annotated with both a selector expression and a property list; for example, the choices `exactly one` and `more than one` in the Environments section. In such a case, both are applied independently; the selector expression is applied to restrict the test frames that will include the choice, and the properties are assigned to any partial frames containing the choice.

The restricted specification of Figure 4 produces 678 test frames, a reduction of two-thirds from the original 1,944. However, 678 is still too many test cases to be economically and practically feasible for a fairly simple operation like **find**.

Steps D and E: Write test specification and evaluate generator output.

If we examine the test frames produced from the restricted specification, we find that many are redundant; they test the same essential situation. They differ only in varying parameters that have no effect on the command's outcome. This occurs frequently when some parameter or environment condition causes an error. For instance, every **find** command for which the named file does not exist will result in the same error, regardless of the form of the pattern.

TSL allows the special annotation `[error]` to be attached to a category choice. `[error]` tests are designed to test a particular feature which will cause an exception or other error state. It is assumed that if the annotated parameter or environment has this particular value, any call of the function using that choice will result in the same error. A choice marked with

```
# Test Specification for find command
# Modified:  property lists and selector expressions added

Parameters:
    Pattern size:
        empty                                [property Empty]
        single character                     [property NonEmpty]
        many character                       [property NonEmpty]
        longer than any line in the file     [property NonEmpty]

    Quoting:
        pattern is quoted                    [property Quoted]
        pattern is not quoted                [if NonEmpty]
        pattern is improperly quoted         [if NonEmpty]

    Embedded blanks:
        no embedded blank                    [if NonEmpty]
        one embedded blank                   [if NonEmpty and Quoted]
        several embedded blanks              [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                   [if NonEmpty]
        one embedded quote                   [if NonEmpty]
        several embedded quotes              [if NonEmpty]

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none                                 [if NonEmpty]
        exactly one                          [if NonEmpty] [property Match]
        more than one                        [if NonEmpty] [property Match]

    Pattern occurrences on target line:
        #  assumes line contains the pattern
        one                                  [if Match]
        more than one                        [if Match]
```

FIGURE 4. Specification with Property Lists and Selector Expressions

[error] is not combined with choices in the other categories to create test frames. The generator creates a test frame that contains only the [error] choice. The test case defined from such a frame must fulfill the statement of the marked [error] choice, but the tester can set the test's other parameters and environment conditions at will. Adding four [error] markings to the restricted specification reduces the number of test frames to 125.

combinations against the pragmatic limits of time and personnel that usually prevail in a software development project.

Step F: Transform into test scripts.
Figure 6 shows one test frame generated from the final specification for **find**, together with the operating system command to establish the file environment, the instance of **find** that performs the test, and a description of the test's expected output.

```
# Test Specification for find command
# Modified: property lists and selector expressions added
# Modified: error and single annotations added

Parameters:
    Pattern size:
        empty                                   [property Empty]
        single character                        [property NonEmpty]
        many character                          [property NonEmpty]
        longer than any line in the file        [error]

    Quoting:
        pattern is quoted                       [property Quoted]
        pattern is not quoted                   [if NonEmpty]
        pattern is improperly quoted            [error]

    Embedded blanks:
        no embedded blank                       [if NonEmpty]
        one embedded blank                      [if NonEmpty and Quoted]
        several embedded blanks                 [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                      [if NonEmpty]
        one embedded quote                      [if NonEmpty]
        several embedded quotes                 [if NonEmpty] [single]

    File name:
        good file name
        no file with this name                  [error]
        omitted                                 [error]

Environments:
    Number of occurrences of pattern in file:
        none                                    [if NonEmpty] [single]
        exactly one                             [if NonEmpty] [property Match]
        more than one                           [if NonEmpty] [property Match]

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one                                     [if Match]
        more than one                           [if Match] [single]
```

FIGURE 5. Final Test Specification for FIND

The number may be further reduced by using [single] markings. This annotation is intended to describe special, unusual, or redundant conditions that do not have to be combined with all possible choices. As with [error] choices, the generator does not combine a [single] choice with any choices from other categories. A single frame is produced that contains only the marked [single] choice. This reduces the total number of frames, but assures that one frame will be created with the marked choice. By marking three choices of the **find** specification as [single], the total number of test frames drops to 40. The final, completely marked specification is shown in Figure 5.

The decision to use a [single] marking is a judgment by the tester that the marked choice can be adequately tested with only one test case. It represents an attempt to balance the desire for complete testing of all

The test frame contains two numbers. The "Test Case" number sequentially identifies the test. The "Key" number refers to the choices made from each of the frame's categories (choice 3 from the first category, choice 1 from the second, and so forth). Once the categories and choices in a test specification are fixed, the key number for a given test frame stays the same even if the property lists and selector expressions change.

In the example shown in Figure 6, the file case_28 has been created earlier and stored in another directory. It is copied into a testing directory to set up the environment for the test. The output of **find** is either a display of matching lines or a message that no matches were found. Test result checking could be done either manually or by directing the command's output to a log file which is later compared to a prepared output file. In cases where the tested function modifies the system

state, additional commands may be necessary to verify that it performed correctly.

Some inconsistencies, redundancies, and ambiguities in the test specifications may only become apparent when the actual test scripts are created. A combination of parameter values and environment conditions that looked right in the specification may actually be impossible to test. One test case may be identical to another. Such situations require that the tester modify the original test specification, usually by adding property lists and selector expressions, to correct the problem.

EXPERIENCE WITH TSL

The TSL is now being used for the design of functional tests for the version and configuration management (VCM) component of an interactive programming support environment. The other components of this system are a text editor and a source-language debugger. The entire system has been designed using object-oriented techniques and is implemented in Ada.

The TSL has been used to create test scripts for the second and third releases of the VCM component. The second release consisted of approximately 35,000 Ada source statements (counted by semicolons), making up over 2,200 Ada procedures in 131 packages. Function testing for this release consisted of testing 91 high-level procedures, each roughly equivalent to one user-level command.

The VCM design and implementation team consists of four members. When functional testing was started, one team member became primarily responsible for testing. This test engineer used the written specification documents, Ada package specifications, and his own experiential knowledge of the system to write the test specifications. One test specification was written for each of the 91 high-level procedures. All of the test specifications were written in approximately three weeks.

The four other team members were then given a one-page overview of the TSL format. These team members individually reviewed the test specifications, noting errors and inconsistencies and suggesting changes. Judging by the quality of the reviews, they found the TSL format to be relatively easy to understand.

Surprisingly, most of the review comments simplified the test specifications. Many environment conditions were eliminated as unnecessary and others were marked `[single]`. In the few instances where functionality had changed since the last revisions of the program specifications, the entire test specification had to be rewritten (and the program specification was quickly revised).

The TSL generator was run on each of the test specifications to produce a set of test frames for each procedure. The entire system required 1,022 test cases.

Writing the actual test scripts was the most time-consuming part of the process. Each procedure's script consisted of some setup commands, the test cases themselves, and cleanup commands. Common data structures were used as appropriate. Although it would have cut down on the total number of commands, it was decided not to use one test case to set the environment for a following test case (e.g., test case A creates a file, test case B reads the file). Wherever possible, the test cases included commands to check the results of the test.

Test Frame:

```
Test Case 28:  (Key = 3.1.3.2.1.2.1.)
   Pattern size : many character
   Quoting : pattern is quoted
   Embedded blanks : several embedded blanks
   Embedded quotes : one embedded quote
   File name : good file name
   Number of occurrences of pattern in file : exactly one
   Pattern occurrences on target line : one
```

Commands to set up the test:

```
copy /testing/sources/case_28 testfile
```

find command to perform the test:

```
find "has "" one quote" testfile
```

Instructions for checking the test:

The following line should be displayed:

```
This line has " one quote on it
```

FIGURE 6. A Test Frame and Test Case

Once the test cases were written, it took approximately two weeks to completely run all of the tests. Many times there were bugs in either the test scripts or the original test specifications themselves, despite the fact that the test specifications had been reviewed prior to the time the test cases were generated and written. This required modifying the scripts, and in a few cases, the test specifications themselves. Thirty-nine *bona fide* bugs in the software were found and corrected using the tests produced with the category-partition method.

Following the development group's testing with the category-partition tests, the system was sent to an independent system testing group for additional testing. This group found 61 additional problems with the VCM software, of which 19 were classified as implementation faults, and the remaining 42 as "documentation errors," "unclear messages," "operating system errors," and "suggestions for improvement."

The complete suite of test scripts has been saved as a regression test base for future releases. All of the test specifications and scripts have been version controlled. To facilitate checking the regression tests, a transcript of the last complete run of the test scripts was saved so that it can be compared mechanically against a transcript of a run of the same scripts against a future release.

RELATION TO OTHER METHODS

The Condition Table Method

Goodenough and Gerhart's [5] test case definition technique is called the *condition table method*. They construct a condition table in which each column represents a combination of conditions that can occur during execution of the program. By examining the program's specifications, they try to identify conditions that have a significant impact on the execution behavior of the program. In the cited article, their main concern was to provide a method that could be used in conjunction with their theory of testing, and they do not discuss any automatization of condition tables.

Some aspects of the Goodenough and Gerhart method are quite similar to the category-partition method. The *conditions* of Goodenough and Gerhart are what we call *categories*. They define the *value set* of a condition as the possible values for the condition; the value set corresponds to what we call *choices* for each category. Goodenough and Gerhart also identify *constraints* between conditions, but use them in a more restricted way than we do. The Goodenough and Gerhart constraints are only used to express interactions between conditions that are consequences of the conditions' definitions. For example, a constraint may say that if condition 1 holds then neither condition 2 nor condition 3 can hold. This type of statement limits the number of combinations that are possible, and hence limits the number of test predicates that must be considered. In the category-partition method, constraints are used both to express definition-based interactions of the above type, and also as a means for the tester to introduce additional restrictions to limit the type and number of test cases that are generated.

Cause-Effect Graphing

Another strategy for transforming a specification is *cause-effect graphing*, originally defined by Elmendorf [4] and illustrated by Myers [8]. This strategy begins with the identification of each individual function or command of the system to be tested. Then for each function, the tester identifies all significant causes that influence the function's behavior, and all effects of the function. The next step is to construct a graph that relates combinations of the causes to the effects they produce. Test cases are defined for each effect by considering all combinations of causes that produce that effect.

Although careful use of the cause-effect method can produce effective tests, the method is difficult to apply in practice. In particular, the cause-effect graph can become very complex when a function has a large number of causes. To keep the complexity under control, the tester must add intermediate nodes to represent logical combinations of several causes. An appropriate choice of intermediate nodes is frequently not obvious. Other problems with the cause-effect graph are the difficulty of verifying its correctness, or of updating it when the specification changes or when the tester realizes that some information has been overlooked. If new nodes are added to the input, or cause, side of the graph, much of its internal structure may have to be redesigned.

By transforming a written specification into a set of cause-effect graphs, the tester is replacing one complex representation with another. Any changes that occur in the specification must be translated into corresponding changes in the graph. Naturally, changes in the specification are a problem for any method, but a simpler intermediate representation can ease the difficulty.

Revealing Subdomains

Weyuker and Ostrand [10] proposed a combined black-box and white-box method to derive a partition of a function's input domain into *revealing subdomains*. A revealing subdomain contains elements that are either all processed correctly or all processed incorrectly. Once such a subdomain has been identified, executing the program on a single one of its elements is sufficient to test the entire subdomain.

The authors note that, theoretically, the existence of a correctly processed input from a subdomain, together with showing that the subdomain is revealing, are equivalent to proving the program's correctness for all inputs in the subdomain. Given the impossibility in general of proving program correctness, one should not expect to be able to produce revealing subdomains at will. To render the theory somewhat more practically applicable, the notion of *revealing subdomain for a specific error E* is introduced. With such a subdomain, if the error E is present and affects some element of the subdomain, then every element of the subdomain is pro-

cessed incorrectly. This definition makes it easier to find revealing subdomains, and guides the tester to concentrate on probable errors and the sets of inputs that they might affect.

The article presents a technique that attempts to construct revealing subdomains by identifying the most likely places for errors to occur. The technique uses information from both the specification and the code. The tester first creates a *problem partition* from the specification, by looking for classes of inputs that should be treated the same way by the program. The next step is to create a *path partition*, whose equivalence classes contain inputs that actually are treated the same way by the program. The partition used for functional testing is then created by intersecting the problem partition and the path partition, creating a set of equivalence classes whose elements both should be and are treated the same way by the program. A test set is built by choosing one element from each of the testing partition's classes.

The revealing subdomain concept emphasizes the importance of choosing test data from both program-dependent and program-independent sources. The main difficulty in its implementation is that there are no formal or systematic guidelines for creating the problem partition; [10] states only that it should be formed "on the basis of common properties implied by the specifications, algorithm, and data structures." Since the category-partition method provides a systematic approach to creating test sets on the basis of the specification, it could conceivably be used to help create the problem partition.

Equivalence Partitioning

Richardson and Clarke's [9] equivalence partitioning method is very similar to the revealing subdomain approach. They partition a function's input domain into a *procedure partition*, which is the intersection of a program-based *path domain* and a specification-based *specification domain*. The path domain is constructed by standard symbolic execution techniques applied to the program. To construct the specification domain, the authors assume that the specification is presented in a formal specification language, to which symbolic execution techniques can be applied.

Test data are selected in two ways, depending on the types of errors the tester is looking for. To find computation errors (incorrect computations), the tester should study the path and specification domains, and use "computationally sensitive data," such as extreme values and special values within those domains. To find domain errors (input data following the wrong path in the program), the tester should use test cases that are boundary points of the procedure domain.

Equivalence partitioning depends crucially on a formal specification to allow the symbolic analysis that creates the specification domain. Assuming such a formal specification exists, the method probably will produce test data that are similar to those produced by the revealing subdomain method. For a system that is specified in natural language, the category-partition technique could be used to determine an appropriate set of specification domains. The informal representations of these domains could be converted into a format similar to that produced by symbolic execution of a formal specification, and the result used to form the procedure partition required by equivalence partitioning.

PLANNED EXTENSIONS

The TSL was developed during the implementation of a major software project, to meet an immediate need for a systematic test development strategy and tool. The criteria were to produce an easily usable and easily understood system as soon as possible. While the present system greatly facilitated testing the VCM software, we are currently considering the following enhancements.

A. **More general ways of specifying and checking properties.** At present, the selector expression on a choice can only be the conjunction of previously assigned properties. A simple change is to allow any Boolean combination of properties. Although in practice the inability to specify any logical combination has not been a problem, this generalization would provide more flexibility in specifying combinations.

B. **Generalize error and single tests.** At present, error and single tests are generated from a single parameter or environment choice, forcing the tester to make an all-or-nothing decision. The ability to use Boolean combinations of choices to specify them would give the tester finer-grained control in designing error or single situations.

C. **Specifying test results.** The present version of TSL provides no means for specifying results of test cases. It is up to the tester to define the expected results for each case when the test scripts are created from test frames. The next version of the tool will allow the tester to include a **Results** section for each function. The user will decompose the possible results for a function into *choices* that are similar to the choices in the decomposition of parameter and environment influences. A Boolean expression attached to each result choice will describe combinations of the parameter and environment choices that cause that result to occur. The information contained in such a result-augmented test specification will be very similar to the information in a cause–effect graph.

   The test generator will then be able to produce the following types of output:

   - Test frames that include a list of expected results.
   - A *result frame* that, for a given result, lists all combinations of parameters and environments that produce that result.

- A list of results that are not produced by any allowable combination of choices.
- A list of choice combinations for which a test frame is produced, but which have no associated result.

D. Automate test script generation. Converting the generator-produced test frames into test scripts is the most time-consuming part of the entire testing process. Much of the conversion could be automated by taking advantage of frequently occurring environment conditions. In using TSL, the test engineer found that after a few test cases were completely written by hand, he could rapidly assemble additional cases by cutting and pasting pieces from the previous ones. A fixed set of commands could be used to establish an environment corresponding to a particular environment category and choice. This idea could be used to build a version of TSL that would automatically supply the appropriate system commands to establish each test case's environment.

SUMMARY

The category-partition method provides the tester with a systematic method for decomposing a functional specification into test specifications for individual functions. The test specification language is a concise, easily modified representation of tests that can be understood by programmers, testers, and managers. A particular benefit is that the tester can control the size of a product's test suite, not by arbitrarily stopping when a given number of tests have been written, but by specifying the interaction of parameters and environments that determine the tests.

TSL and the category-partition method are a promising foundation for further automation of specification-based functional testing, a task that is generally seen as tedious but necessary for the implementation of high quality software.

REFERENCES

1. Adrion, W.R., Branstad, M.A., and Cherniavsky, J.C. Validation, verification, and testing of computer software. *ACM Comp. Surv. 14*, 2 (June 1982), 159–192.
2. Berzins, V., and Gray, M. Analysis and design in MSG 84: Formalizing functional specifications. *IEEE Trans. Softw. Eng. SE-11*, 8 (August 1985), 657–670.
3. Bjorner, D., and Jones, C.B. *Formal Specification and Software Development*. Prentice-Hall International, Englewood Cliffs, N J., 1982.
4. Elmendorf, W.R. Functional analysis using cause-effect graphs. In *Proceedings of SHARE XLIII*. SHARE, New York, 1974, 567–577.
5. 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.
6. Howden, W.H. A survey of dynamic analysis methods In *Tutorial: Program Testing and Validation Techniques*, edited by E.F. Miller and W.H. Howden. IEEE, 1981.
7. Howden, W.H. Errors, design properties, and functional program tests. In *Computer Program Testing Summer School, Sogesta, Italy*, edited by B. Chandrasekaran and S. Radicchi. North-Holland, New York, 1981.
8. Myers, G.J. *Software Reliability: Principles and Practices*. John Wiley, New York, 1976.
9. Richardson, D. and Clarke, L. A partition analysis method to increase program reliability. In *Proceedings of the 5th International Conference Software Engineering* (San Diego, CA, Mar. 9–12). IEEE, 1981, pp. 244–253.
10. Weyuker, E.J., and Ostrand, T.J. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng. SE-6*, 3 (May 1980), 236–246.

CR Categories and Subject Descriptors: D.2.5. [**Testing and Debugging**]; K.6.3 [**Software Management**]

Additional Key Words and Phrases: Automatic test generator, partition, software testing, test specification

Authors' Present Address: Thomas J. Ostrand and Marc J. Balcer, Software Technology Department, Siemens Research and Technology Laboratories, 105 College Road East, Princeton, N.J. 08540.