Chapter 13

Functional Testing

A functional specification is a description of intended program[1] behavior, distinct from the program itself. Whatever form the functional specification takes — whether formal or informal — it is the most important source of information for designing tests. The set of activities for deriving test case specifications from program specifications is called functional testing.

Functional testing, or more precisely, functional test case design, attempts to answer the question "What test cases shall I use to exercise my program?" considering only the specification of a program and not its design or implementation structure. Being based on program specifications and not on the internals of the code, functional testing is also called specification-based or black-box testing.

Functional testing is typically the base-line technique for designing test cases, for a number of reasons. Functional test case design can (and should) begin as part of the requirements specification process, and continue through each level of design and interface specification; it is the only test design technique with such wide and early applicability. Moreover, functional testing is effective in finding some classes of fault that typically elude so-called "white-box" or "glass-box" techniques of structural or fault-based testing. Functional testing techniques can be applied to any description of program behavior, from an informal partial description to a formal specification and at any level of granularity, from module to system testing. Finally, functional test cases are typically less expensive to design and execute than white-box tests.

[1]In this chapter we use the term "program" generically for the artifact under test, whether that artifact is a complete application or an individual unit together with a test harness. This is consistent with usage in the testing research literature.

Required Background

- Chapters 14 and 15:
  The material on control and data flow graphs is required to understand section 13.7, but it is not necessary to comprehend the rest of the chapter.

- Chapter 27:
  The definition of pre- and post-conditions can be helpful in understanding section 13.8, but it is not necessary to comprehend the rest of the chapter.

13.1 Overview

In testing and analysis aimed at verification[2] — that is, at finding any discrepancies between what a program does and what it is intended to do — one must obviously refer to requirements as expressed by users and specified by software engineers. A functional specification, i.e., a description of the expected behavior of the program, is the primary source of information for test case specification.

Δ Black-box testing

Functional testing, also known as black-box or specification-based testing, denotes techniques that derive test cases from functional specifications. Usually functional testing techniques produce test case specifications that identify classes of test cases and be be instantiated to produce individual test cases.

A particular functional testing technique may be effective only for some kinds of software or may require a given specification style. For example, a combinatorial approach may work well for functional units characterized by a large number of relatively independent inputs, but may be less effective for functional units characterized by complex interrelations among inputs. Functional testing techniques designed for a given specification notation, e.g., finite state machines or grammars, are not easily applicable to other specification styles.

The core of functional test case design is partitioning the possible behaviors of the program into a finite number of classes that can reasonably expected to consistently be correct or incorrect. In practice, the test case designer often must also complete the job of formalizing the specification far enough to serve as the basis for identifying classes of behaviors. An important side effect of test design is highlighting weaknesses and incompleteness of program specifications.

Deriving functional test cases is an analytical process which decomposes specifications into test cases. The myriad of aspects that must be taken into

[2]Here we focus on software verification as opposed to validation (see Chapter 2). The problems of validating the software and its specifications, i.e., checking the program behavior and its specifications with respect to the users' expectations, is treated in Chapter 12.

Functional vs. Structural Testing

Test cases and test suites can be derived from several sources of information, including specifications (functional testing), detailed design and source code (structural testing), and hypothesized defects (fault-based testing). Functional test case design is an indispensable base of a good test suite, complemented but never replaced by by structural and fault-based testing, because there are classes of faults that only functional testing effectively detects. Omission of a feature, for example, is unlikely to be revealed by techniques which refer only to the code structure.

Consider a program that is supposed to accept files in either plain ASCII text, or HTML, or PDF formats and generate standard PostScript. Suppose the programmer overlooks the PDF functionality, so the program accepts only plain text and HTML files. Intuitively, a functional testing criterion would require at least one test case for each item in the specification, regardless of the implementation, i.e., it would require the program to be exercised with at least one ASCII, one HTML, and one PDF file, thus easily revealing the failure due to the missing code. In contrast, criterion based solely on the code would not require the program to be exercised with a PDF file, since all of the code can be exercised without attempting to use that feature. Similarly, fault-based techniques, based on potential faults in design or coding, would not have any reason to indicate a PDF file as a potential input even if “missing case” were included in the catalog of potential faults.

A functional specification often addresses semantically rich domains, and we can use domain information in addition to the cases explicitly enumerated in the program specification. For example, while a program may manipulate a string of up to nine alphanumeric characters, the program specification may reveal that these characters represent a postal code, which immediately suggests test cases based on postal codes of various localities. Suppose the program logic distinguishes only two cases, depending on whether they are found in a table of U.S. zip codes. A structural testing criterion would require testing of valid and invalid U.S. zip codes, but only consideration of the specification and richer knowledge of the domain would suggest test cases that reveal missing logic for distinguishing between U.S.-bound mail with invalid U.S. zip codes and mail bound to other countries.

Functional testing can be applied at any level of granularity where some form of specification is available, from overall system testing to individual units, although the level of granularity and the type of software influence the choice of the specification styles and notations, and consequently the functional testing techniques that can be used.

In contrast, structural and fault-based testing techniques are invariably tied to program structures at some particular level of granularity, and do not scale much beyond that level. The most common structural testing techniques are tied to fine-grain program structures (statements, classes, etc.) and are applicable only at the level of modules or small collections of modules (small subsystems, components, or libraries).

account during functional test case specification makes the process error prone. Even expert test designers can miss important test cases. A methodology for functional test design systematically helps by decomposing the functional test design activity into elementary steps that cope with single aspect of the process. In this way, it is possible to master the complexity of the process and separate human intensive activities from activities that can be automated. Systematic processes amplify but do not substitute for skills and experience of the test designers.

In a few cases, functional testing can be fully automated. This is possible for example when specifications are given in terms of some formal model, e.g., a grammar or an extended state machine specification. In these (exceptional) cases, the creative work is performed during specification and design of the software. The test designer's job is then limited to the choice of the test selection criteria, which defines the strategy for generating test case specifications. In most cases, however, functional test design is a human intensive activity. For example, when test designers must work from informal specifications written in natural language, much of the work is in structuring the specification adequately for identifying test cases.

13.2 Random versus Partition Testing Strategies

With few exceptions, the number of potential test cases for a given program is unimaginably huge — so large that for all practical purposes it can be considered infinite. For example, even a simple function whose input arguments are two 32-bit integers has $2^{64} \approx 10^{54}$ legal inputs. In contrast to input spaces, budgets and schedules are finite, so any practical method for testing must select an infinitesimally small portion of the complete input space.

Some test cases are better than others, in the sense that some reveal faults and others do not.[3] Of course, we cannot know in advance which test cases reveal faults. At a minimum, though, we can observe that running the same test case again is less likely to reveal a fault than running a different test case, and we may reasonably hypothesize that a test case that is very different from the test cases that precede it is more valuable than a test case that is very similar (in some sense yet to be defined) to others.

As an extreme example, suppose we are allowed to select only three test cases for a program that breaks a text buffer into lines of 60 characters each. Suppose the first test case is a buffer containing 40 characters, and the second is a buffer containing 30 characters. As a final test case, we can choose a buffer containing 16 characters or a buffer containing 100 characters. Although we cannot prove that the 100 character buffer is the better test case (and it might not be; the fact that 16 is a power of 2 might have some unforeseen significance), we are naturally suspicious of a set of tests which is strongly biased toward lengths less than 60.

[3]Note that the relative value of different test cases would be quite different if our goal were to measure dependability, rather than finding faults so that they can be repaired.

Testing Terms

While the informal meanings of words like "test" may be adequate for everyday conversation, in this context we must try to use terms in a more precise and consistent manner. Unfortunately, the terms we will need are not always used consistently in the literature, despite the existence of an IEEE standard that defines several of them. The terms we will use are defined below.

Independently testable feature (ITF): An *ITF* is a functionality that can be tested independently of other functionalities of the software under test. It need not correspond to a unit or subsystem of the software. For example, a file sorting utility may be capable of merging two sorted files, and it may be possible to test the sorting and merging functionalities separately, even though both features are implemented by much of the same source code. (The nearest IEEE standard term is "test item.")

As functional testing can be applied at many different granularities, from unit testing through integration and system testing, so ITFs may range from the functionality of an individual Java class or C function up to features of a integrated system composed of many complete programs. The granularity of an ITF depends on the exposed interface at whichever granularity is being tested. For example, individual methods of a class are part of the interface of the class, and a set of related methods (or even a single method) might be an ITF for unit testing, but for system testing the ITFs would be features visible through a user interface or application programming interface.

Test case: A *test case* is a set of inputs, execution conditions, and expected results. The term "input" is used in a very broad sense, which may include all kinds of stimuli that contribute to determining program behavior. For example, an interrupt is as much an input as is a file. (This usage follows the IEEE standard.)

Test case specification: The distinction between a test case specification and a test case is similar to the distinction between a program and a program specification. Many different test cases may satisfy a single test case specification. A simple test specification for a sorting method might require an input sequence that is already in sorted order. A test case satisfying that specification might be sorting the particular vector *("alpha," "beta," "delta.")* (This usage follows the IEEE standard.)

Test suite: A *test suite* is a set of test cases. Typically, a method for functional testing is concerned with creating a test suite. A test suite for a program, a system, or an individual unit may be made up of several test suites for individual ITFs. (This usage follows the IEEE standard.)

Test: We use the term *test* to refer to the activity of executing test cases and evaluating their result. When we refer to "a test," we mean execution of a single test case, except where context makes it clear that the reference is to execution of a whole test suite. (The IEEE standard allows this and other definitions.)

Accidental bias may be avoided by choosing test cases from a random distribution. Random sampling is often an inexpensive way to produce a large number of test cases. If we assume absolutely no knowledge on which to place a higher value on one test case than another, then random sampling maximizes value by maximizing the number of test cases that can be created (without bias) for a given budget. Even if we do possess some knowledge suggesting that some cases are more valuable than others, the efficiency of random sampling may in some cases outweigh its inability to use any knowledge we may have.

Consider again the line-break program, and suppose that our budget is one day of testing effort rather than some arbitrary number of test cases. If the cost of random selection and actual execution of test cases is small enough, then we may prefer to run a large number of random test cases rather than expending more effort on each of a smaller number of test cases. We may in a few hours construct programs that generate buffers with various contents and lengths up to a few thousand characters, as well as an automated procedure for checking the program output. Letting it run unattended overnight, we may execute a few million test cases. If the program does not correctly handle a buffer containing a sequence of more than 60 non-blank characters (a single "word" that does not fit on a line), we are likely to encounter this case by sheer luck if we execute enough random tests, even without having explicitly considered this case.

Even a few million test cases is an infinitesimal fraction of the complete input space of most programs. Large numbers of random tests are unlikely to find failures at single points (singularities) in the input space. Consider, for example, a simple procedure for returning the two roots of a quadratic equation $ax^2 + bx + c = 0$ and suppose we choose test inputs (values of the coefficients a, b, and c) from a uniform distribution ranging from -10.0 to 10.0. While uniform random sampling would certainly cover cases in which $b^2 - 4ac > 0$ (where the equation has no real roots), it would be very unlikely to test the case in which $a = 0$ and $b = 0$, in which case a naive implementation of the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

will divide by zero (see Figure 13.1).

Of course, it is unlikely that anyone would test *only* with random values. Regardless of the overall testing strategy, most test designers will also try some "special" values. The test designer's intuition comports with the observation that random sampling is an ineffective way to find singularities in a large input space. The observation about singularities can be generalized to any characteristic of input data that defines an infinitesimally small portion of the complete input data space. If again we have just three real-valued inputs a, b, and c, there is an infinite number of choices for which $b = c$, but random sampling is unlikely to generate any of them because they are an infinitesimal part of the complete input data space.

```java
  class Roots {
    double root_one;
    double root_two;
    int num_roots;
    /**
     * Find roots:
     *   Given a quadratic equation ax^2 + bx + c
     *   produce the two values of x at which the
     *   formula evaluates to 0.
     */
    public roots(double a, double b, double c) {
        double q;
        double r;
        // Apply the quadratic formula, from any high school
        // algebra textbook:
        // Roots = -b +- sqrt(b^2 - 4ac) / 2a
        q = b*b - 4*a*c;
        if (q > 0 && a != 0) {
            // If b^2 > 4ac, the formula has
            // two distinct roots
            num_roots = 2;
            r = (double) Math.sqrt(q) ;
            root_one =  ((0-b) + r)/(2*a);
            root_two =  ((0-b) - r)/(2*a);
        } else if (q==0) {
            // The equation has exactly one root
            // ((BUG HERE))
            num_roots = 1;
            root_one = (0-b)/(2*a);
            root_two = root_one;
        } else {
            // The equation has no roots if b^2 < 4ac
            num_roots = 0;
            root_one = -1;
            root_two = -1;
        }
    }
    public int num_roots() { return num_roots; }
    public double first_root()  { return root_one; }
    public double second_root() { return root_two; }
}
```

Figure 13.1: The Java class “roots,” which finds roots of a quadratic equation. The case analysis in the implementation is incomplete: It does not properly handle the case in which $b^2 - 4ac = 0$ and $a = 0$. We cannot anticipate all such faults, but experience teaches that boundary values identifiable in a specification are disproportionately valuable. Uniform random generation of even large numbers of test cases is ineffective at finding the fault in this program, but selection of a few “special values” based on the specification quickly uncovers it.

The observation about special values and random samples is by no means limited to numbers. Consider again, for example, breaking a text buffer into lines. Since line breaks are permitted at blanks, we would consider blanks a "special" value for this problem. While random sampling from the character set is likely to produce a buffer containing a sequence of at least 60 non-blank characters, it is much less likely to produce a sequence of 60 blanks.

The reader may justifiably object that a reasonable test designer would not create text buffer test cases by sampling uniformly from the set of all characters, but would instead classify characters depending on their treatment, lumping alphabetic characters into one class and white space characters into another. In other words, a test designer will *partition* the input space into classes, and will then generate test data in a manner that is likely to choose data from each partition.[4] Test designers seldom use pure random sampling; usually they exploit some knowledge of application semantics to choose samples that are more likely to include "special" or trouble-prone regions of the input space.

Δ Partition testing

Δ Specification-based testing

Δ Functional testing

A testing method that divides the infinite set of possible test cases into a finite set of classes, with the purpose of drawing one or more test cases from each class, is called a *partition testing* method. When partitions are chosen according to information in the specification, rather than the design or implementation, it is called *specification-based partition testing*, or more briefly, *functional testing*. Note that not all testing of product functionality is "functional testing." Rather, the term is used specifically to refer to systematic testing based on a functional specification. It excludes ad hoc and random testing, as well as testing based on the structure of a design or implementation.

Partition testing typically increases the cost of each test case, since in addition to generation of a set of classes, creation of test cases from each class may be more expensive than generating random test data. In consequence, partition testing usually produces fewer test cases than random testing for the same expenditure of time and money. Partitioning can therefore be advantageous only if the average value (fault-detection effectiveness) is greater.

If we were able to group together test cases with such perfect knowledge that the outcome of test cases in each class were uniform (either all successes, or all failures), then partition testing would be at its theoretical best. In general we cannot do that, nor even quantify the uniformity of classes of test cases. Partitioning by any means, including specification-based partition testing, is always based on experience and judgment that leads one to believe that certain classes of test case are "more alike" than others, in the sense that failure-prone test cases are likely to be concentrated in some classes. When we appealed above to the test designer's intuition that one should try boundary cases and special values, we were actually appealing to a combination of experience (many failures occur at boundary and special cases) and knowl-

[4]We are using the term "partition" in a common but rather sloppy sense. A true partition would separate the input space into disjoint classes, the union of which is the entire space. Partition testing separates the input space into classes whose union is the entire space, but the classes may not be disjoint.

edge that identifiable cases in the specification often correspond to classes of input that require different treatment by an implementation.

Given a fixed budget, the optimum may not lie in only partition testing or only random testing, but in some mix that makes use of available knowledge. For example, consider again the simple numeric problem with three inputs, a, b, and c. We might consider a few special cases of each input, individually and in combination, and we might consider also a few potentially-significant relationships (e.g., $a = b$). If no faults are revealed by these few test cases, there is little point in producing further arbitrary partitions — one might then turn to random generation of a large number of test cases.

13.3 A Systematic Approach

Deriving test cases from functional specifications is a complex analytical process that partitions the input space described by the program specification. Brute force generation of test cases, i.e., direct generation of test cases from program specifications, seldom produces acceptable results: test cases are generated without particular criteria and determining the adequacy of the generated test cases is almost impossible. Brute force generation of test cases relies on test designers' expertise and is a process that is difficult to monitor and repeat. A systematic approach simplifies the overall process by dividing the process into elementary steps, thus decoupling different activities, dividing brain intensive from automatable steps, suggesting criteria to identify adequate sets of test cases, and providing an effective means of monitoring the testing activity.

Although suitable functional testing techniques can be found for any granularity level, a particular functional testing technique may be effective only for some kinds of software or may require a given specification style. For example, a combinatorial approach may work well for functional units characterized by a large number of relatively independent inputs, but may be less effective for functional units characterized by complex interrelations among inputs. Functional testing techniques designed for a given specification notation, e.g., finite state machines or grammars, are not easily applicable to other specification styles. Nonetheless we can identify a general pattern of activities that captures the essential steps in a variety of different functional test design techniques. By describing particular functional testing techniques as instantiations of this general pattern, relations among the techniques may become clearer, and the test designer may gain some insight into adapting and extending these techniques to the characteristics of other applications and situations.

Figure 13.2 identifies the general steps of systematic approaches. The steps may be difficult or trivial depending on the application domain and the available program specifications. Some steps may be omitted depending on the application domain, the available specifications and the test designers' expertise. Instances of the process can be obtained by suitably instantiating different steps. Although most techniques are presented and applied as stand alone methods, it is also possible to mix and match steps from different techniques, or to apply different methods for different parts of the system to be tested.

Identify Independently Testable Features Functional specifications can be large and complex. Usually, complex specifications describe systems that can be decomposed into distinct features. For example, the specification of a web site may include features for searching the site database, registering users' profiles, getting and storing information provided by the users in different forms, etc. The specification of each of these features may comprise several functionalities. For example, the search feature may include functionalities for editing a search pattern, searching the data base with a given pattern, and so on. Although it is possible to design test cases that exercise several functionalities at once, the design of different tests for different functionalities can simplify the test generation problem, allowing each functionality to be examined separately. Moreover, it eases locating faults that cause the revealed failures. It is thus recommended to devise separate test cases for each functionality of the system, whenever possible.

The preliminary step of functional testing consists in partitioning the specifications into features that can be tested separately. This can be an easy step for well designed, modular specifications, but informal specifications of large systems may be difficult to decompose into independently testable features. Some degree of formality, at least to the point of careful definition and use of terms, is usually required.

Identification of functional features that can be tested separately is different from module decomposition. In both cases we apply the divide and conquer principle, but in the former case, we partition specifications according to the functional behavior as perceived by the users of the software under test,[5] while in the latter, we identify logical units that can be implemented separately. For example, a web site may require a *sort* function, as a service routine, that does not correspond to an external functionality. The sort function may be a functional feature at module testing, when the program under test is the sort function itself, but is not a functional feature at system test, while deriving test cases from the specifications of the whole web site. On the other hand, the registration of a new user profile can be identified as one of the functional features at system level testing, even if such functionality is implemented with several modules (unit at the design level) of the system. Thus, identifying functional features does not correspond to identifying single modules at the design level, but rather to suitably slicing the specifications to be able to attack their complexity incrementally, aiming at deriving useful test cases for the whole system under test.

[5]Here the word user indicates who uses the specified service. It can be the user of the system, when dealing with specification at system level; but it can be another module of the system, when dealing with specifications at unit level.

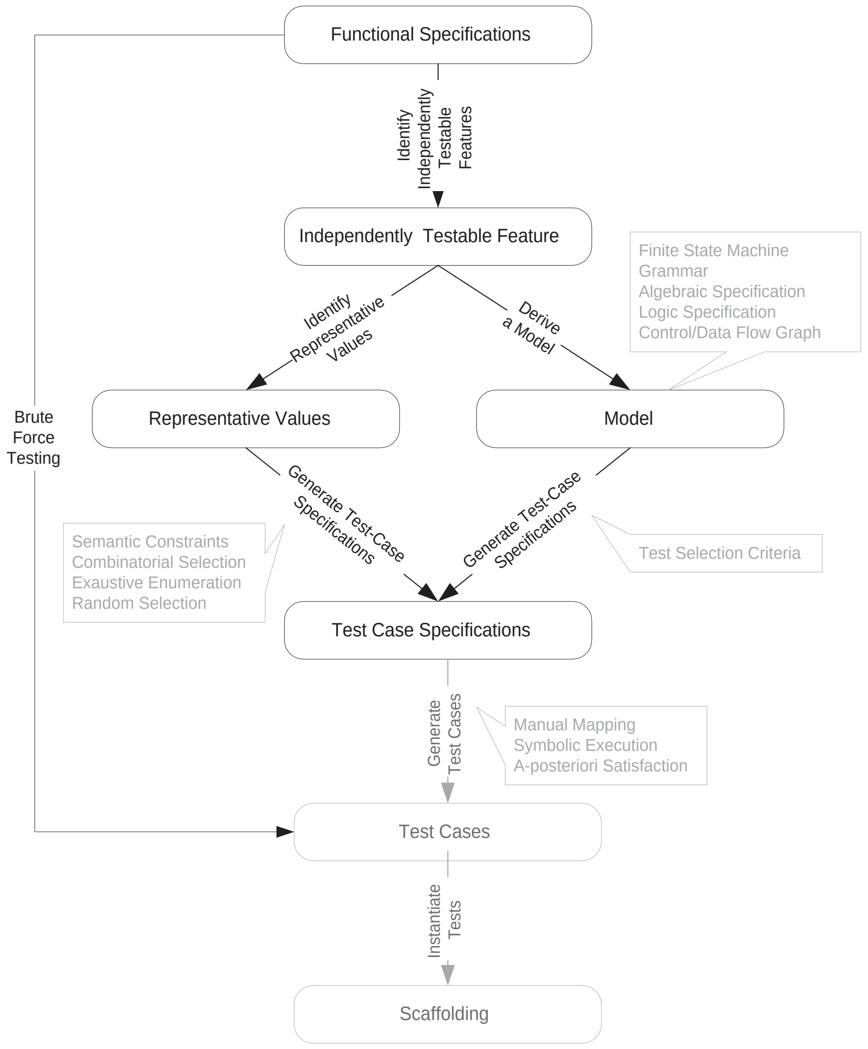


Figure 13.2: The main steps of a systematic approach to functional program testing.

Independently testable features are described by identifying all the inputs that form their execution environments. Inputs may be given in different forms depending on the notation used to express the specifications. In some cases they may be easily identifiable. For example, they can be the input alphabet of a finite state machine specifying the behavior of the system. In other cases, they may be hidden in the specification. This is often the case of informal specifications, where some inputs may be given explicitly as parameters of the functional unit, but other inputs may be left implicit in the description. For example, a description of how a new user registers at a web site may explicitly indicate the data that constitutes the user profile to be inserted as parameters of the functional unit, but may leave implicit the collection of elements (e.g., database) in which the new profile must be inserted.

Trying to identify inputs may help in distinguishing different functions. For example, trying to identify the inputs of a graphical tool may lead to a clearer distinction between the graphical interface per se and the associated calbacks to the application. With respect to the web-based user registration function, the data to be inserted in the database are part of the execution environment of the functional unit that performs the insertion of the user profile, while the combination of fields that can be use to construct such data is part of the execution environment of the functional unit that takes care of the management of the specific graphical interface.

Identify Representative Classes of Values or Derive a Model The execution environment of the feature under test determines the form of the final test cases, which are given as combinations of values for the inputs to the unit. The next step of a testing process consists of identifying which values of each input can be chosen to form test cases. Representative values can be identified directly from informal specifications expressed in natural language. Alternativey, representative values may be selected indirectly through a model, which can either be produced only for the sake of testing or be available as part of the specification. In both cases, the aim of this step is to identify the values for each input in isolation, either explicitly through enumeration, or implicitly trough a suitable model, but not to select suitable combinations of such values, i.e., test case specifications. In this way, we separate the problem of identifying the representative values for each input, from the problem of combining them to obtain meaningful test cases, thus splitting a complex step into two simpler steps.

Most methods that can be applied to informal specifications rely on explicit enumeration of representative values by the test designer. In this case, it is very important to consider all possible cases and take advantage of the information provided by the specification. We may identify different categories of expected values, as well as boundary and exceptional or erroneous values. For example, when considering operations on a non-empty lists of elements, we may distinguish the cases of the empty list (an error value) and a singleton element (a boundary value) as special cases. Usually this step determines characteristics of values (e.g., any list with a single element) rather than actual values.

Implicit enumeration requires the construction of a (partial) model of the specifications. Such a model may be already available as part of a specification or design model, but more often it must be constructed by the test designer, in consultation with other designers. For example, a specification given as a finite state machine implicitly identifies different values for the inputs by means of the transitions triggered by the different values. In some cases, we can construct a partial model as a mean for identifying different values for the inputs. For example, we may derive a grammar from a specification and thus identify different values according to the legal sequences of productions of the given grammar.

Directly enumerating representative values may appear simpler and less expensive than producing a suitable model from which values may be derived. However, a formal model may also be valuable in subsequent steps of test case design, including selection of combinations of values. Also, a formal model may make it easier to select a larger or smaller number of test cases, balancing cost and thoroughness, and may be less costly to modify and reuse as the system under test evolves. Whether to invest effort in producing a model is ultimately a management decision that depends on the application domain, the skills of test designers, and the availability of suitable tools.

Generate Test Case Specifications Test specifications are obtained by suitably combining values for all inputs of the functional unit under test. If representative values were explicitly enumerated in the previous step, then test case specifications will be elements of the Cartesian product of values selected for each input. If a formal model was produced, then test case specifications will be specific behaviors or combinations of parameters of the model, and single test case specification could be satisfied by many different concrete inputs. Either way, brute force enumeration of all combinations is unlikely to be satisfactory.

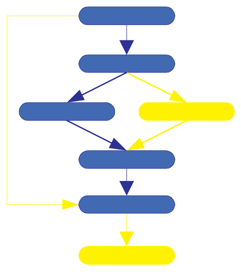
The number of combinations in the Cartesian product of independently selected values grows as the product of the sizes of the individual sets. For a simple functional unit with 5 inputs each characterized by 6 values, the size of the Cartesian product is $6^5 = 7,776$ test case specifications, which may be an impractical number for test cases for a simple functional unit. Moreover, if (as is usual) the characteristics are not completely orthogonal, many of these combinations may not even be feasible.

Consider the input of a function that searches for occurrences of a complex pattern in a web database. Its input may be characterized by the length of the pattern and the presence of special characters in the pattern, among other aspects. Interesting values for the length of the pattern may be zero, one, or many. Interesting values for the presence of special characters may be zero, one, or many. However, the combination of value “zero” for the length of the pattern and value “many” for the number of special characters in the pattern is clearly impossible.

The test case specifications represented by the Cartesian product of all possible inputs must be restricted by ruling out illegal combinations and selecting a practical subset of the legal combinations. Illegal combinations are usually eliminated by constraining the set of combinations. For example, in the case of the complex pattern presented above, we can constrain the choice of one or more special characters to a positive length of the pattern, thus ruling out the illegal cases of patterns of length zero containing special characters.

Selection of a practical subset of legal combination can be done by adding information that reflects the hazard of the different combinations as perceived by the test designer or by following combinatorial considerations. In the former case, for example, we can identify exceptional values and limit the combinations that contain such values. In the pattern example, we may consider only one test for patterns of length zero, thus eliminating many combinations that can be derived for patterns of length zero. Combinatorial considerations reduce the set of test cases by limiting the number of combination of values of different inputs to a subset of the inputs. For example, we can generate only tests that exhaustively cover all combinations of values for inputs considered pair by pair.

Depending on the technique used to reduce the space represented by the Cartesian product, we may be able to estimate the number of test cases generated with the approach and modify the selected subset of test cases according to budget considerations. Subsets of combinations of values, i.e., potential special cases, can be often derived from models of behavior by applying suitable test selection criteria that identify subsets of interesting behaviors among all behaviors represented by a model, for example by constraining the iterations on simple elements of the model itself. In many cases, test selection criteria can be applied automatically.



Generate Test Cases and Instantiate Tests The test generation process is completed by turning test case specifications into test cases and instantiating them. Test case specifications can be turned into test cases by selecting one or more test cases for each item of the test case specification.

13.4 Category-Partition Testing

Category-partition testing is a method for generating functional tests from informal specifications. The main steps covered by the core part of the category-partition method are:

A. Decompose the specification into independently testable features: Test designers identify features to be tested separately, and identify parameters and any other elements of the execution environment the unit depends on. Environment dependencies are treated identically to explicit parameters. For each parameter and environment element, test designers identify the elementary *parameter characteristics*, which in the category-partition method are usually called *categories*.

Δ Parameter characteristic
Δ Category

B. Identify Relevant Values: Test designers select a set of representative classes of values for each parameter characteristic. Values are selected in isolation, independent of other parameter characteristics. In the category-partition method, classes of values are called *choices,* and this activity is called *partitioning the categories into choices*.

Δ Classes of Values
Δ Choice

C. Generate Test Case Specifications: Test designers indicate invalid combinations of values and restrict valid combinations of values by imposing semantic constraints on the identified values. Semantic constraints restrict the values that can be combined and identify values that need not be tested in different combinations, e.g., exceptional or invalid values.

Categories, choices, and constraints can be provided to a tool to automatically generate a set of test case specifications. Automating trivial and repetitive activities such as these makes better use of human resources and reduces errors due to distraction. Just as important, it is possible to determine the number of test cases that will be generated (by calculation, or by actually generating them) before investing any human effort in test execution. If the number of derivable test cases exceeds the budget for test execution and evaluation, test designers can reduce the number of test cases by imposing additional semantic constraints. Controlling the number of test cases *before* test execution begins is preferable to ad hoc approaches in which one may at first create very thorough test suites and then test less and less thoroughly as deadlines approach.

We illustrate the category-partition method using a specification of a feature from the direct sales web site of Chipmunk Electronic Ventures. Customers are allowed to select and price custom configurations of Chipmunk computers. A *configuration* is a set of selected options for a particular model of computer. Some combinations of model and options are not valid (e.g., digital LCD monitor with analog video card), so configurations are tested for validity before they are priced. The *check-configuration* function (Table 13.3) is given a model number and a set of components, and returns the boolean value **True** if the configuration is valid or **False** otherwise. This function has been selected by the test designers as an independently testable feature.

A. Identify Independently Testable Features and Parameter Characteristics
We assume that step A starts by selecting the *Check-configuration* feature to be tested independently of other features. This entails choosing to separate testing of the configuration check per se from its presentation through a user interface (e.g., a web form), and depends on the architectural design of the software system.

Check-Configuration: The *Check-configuration* function checks the validity of a computer configuration. The parameters of check-configuration are:

Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs.
Example: The required "slots" of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.

Set of Components: A set of $\langle slot, component\rangle$ pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value "empty" is allowed (and may be the default selection) for optional slots.

In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

Example: The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, "empty" is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of of hard disk.

Figure 13.3: The functional specification of the feature *Check-configuration* of the web site of a computer manufacturer.

Step A requires the test designer to identify the parameter characteristics, i.e., the elementary characteristics of the parameters and environment elements that affect the unit's execution. A single parameter may have multiple elementary characteristics. A quick scan of the functional specification would indicate *model* and *components* as the parameters of *check configuration*. More careful consideration reveals that what is "valid" must be determined by reference to additional information, and in fact the functional specification assumes the existence of a data base of models and components. The data base is an environment element that, although not explicitly mentioned in the functional specification, is required for executing and thus testing the feature, and partly determines its behavior. Note that our goal is not to test a particular configuration of the system with a fixed database, but to test the generic system which may be configured through different database contents.

Having identified *model*, *components*, and *product database* as the parameters and environment elements required to test the *Check-configuration* functionality, the test designer would next identify the parameter characteristics of each.

Model may be represented as an integer, but we know that it is not to be used arithmetically, but rather serves as a key to the database and other tables. The specification mentions that a model is characterized by a set of slots for required components and a set of slot for optional components. We may identify *model number*, *number of required slots*, and *number of optional slots* as characteristics of parameter *model*.

Parameter *components* is a collection of $\langle slot, selection \rangle$ pairs. The size of a collection is always an important characteristic, and since components are further categorized as required or optional, the test designer may identify *number of required components with non-empty selection* and *number of optional components with non-empty selection* as characteristics. The matching between the tuple passed to *Check-Configuration* and the one actually required by the selected model is important and may be identified as category *Correspondence of selection with model slots*. The actual selections are also significant, but for now the test designer simply identifies *required component selection* and *optional component selection*, postponing selection of relevant values to the next stage in test design.

The environment element *product database* is also a collection, so *number of models in the database* and *number of components in the database* are parameter characteristics. Actual values of database entries are deferred to the next step in test design.

There are no hard-and-fast rules for choosing categories, and it is not a trivial task. Categories reflect the test designer's judgment regarding which classes of values may be treated differently by an implementation, in addition to classes of values that are explicitly identified in the specification. Test designers must also use their experience and knowledge of the application domain and product architecture to look under the surface of the specification and identify hidden characteristics. For example, the specification fragment in Table 13.3 makes no distinction between configurations of models with several required slots and models with none, but the experienced test designer has seen enough failures on "degenerate" inputs to test empty collections wherever a collection is allowed.

The number of options that can (or must) be configured for a particular model of computer may vary from model to model. However, the category-partition method makes no direct provision for structured data, such as sets of $\langle slot, selection\rangle$ pairs. A typical approach is to "flatten" collections and describe characteristics of the whole collection as parameter characteristics. Typically the size of the collection (the length of a string, for example, or in this case the number of required or optional slots) is one characteristic, and descriptions of possible combination of elements (occurrence of a special characters in a string, for example, or in this case the selection of required and optional components) are separate parameter characteristics.

Suppose the only significant variation among $\langle slot, selection\rangle$ pairs was between pairs that are compatible and pairs that are incompatible. If we treated each $\langle slot, selection\rangle$ pair as a separate characteristic, and assumed n slots, the category-partition method would generate all 2^n combinations of compatible and incompatible slots. Thus we might have a test case in which the first selected option is compatible, the second is compatible, and the third incompatible, and a different test case in which the first is compatible but the second and third are incompatible, and so on, and each of these combinations could be combined in several ways with other parameter characteristics. The number of combinations quickly explodes, and moreover since the number of slots is not actually fixed, we cannot even place an upper bound on the number of combinations that must be considered. We will therefore choose the flattening approach and select possible patterns for the collection as a whole.

Should the representative values of the flattened collection of pairs be *one compatible selection, one incompatible selection, all compatible selections, all incompatible selections,* or should we also include *mix of 2 or more compatible and 2 or more incompatible selections*? Certainly the latter is more thorough, but whether there is sufficient value to justify the cost of this thoroughness is a matter of judgment by the test designer.

We have oversimplified by considering only whether a selection is compatible with a slot. It might also happen that the selection does not appear in the database. Moreover, the selection might be incompatible with the model, or with a selected component of another slot, in addition to the possibility that it is incompatible with the slot for which it has been selected. If we treat each such possibility as a separate parameter characteristic, we will generate many combinations, and we will need semantic constraints to rule out combinations like *there are three options, at least two of which are compatible with the model and two of which are not, and none of which appears in the database.* On the other hand, if we simply enumerate the combinations that do make sense and are worth testing, then it becomes more difficult to be sure that no important combinations have been omitted. Like all design decisions, the way in which collections and complex data are broken into parameter characteristics requires judgment based on a combination of analysis and experience.

B. Identify Relevant Values This step consists of identifying a list of relevant values (more precisely, a list of classes of relevant values) for each of the parameter characteristics identified during step A. Relevant values should be identified for each category independently, ignoring possible interactions among values for different categories, which are considered in the next step.

Relevant values may be identified by manually applying a set of rules known as boundary value testing or erroneous condition testing. The boundary value testing rule suggests selection of extreme values within a class (e.g., maximum and minimum values of the legal range), values outside but as close as possible to the class, and "interior" (non-extreme) values of the class. Values near the boundary of a class are often useful in detecting "off by one" errors in programs. The erroneous condition rule suggests selecting values that are outside the normal domain of the program, since experience suggests that proper handling of error cases is often overlooked.

Table 13.1 summarizes the parameter characteristics and the corresponding relevant values identified for feature *Check-configuration*.[6] For numeric characteristics, whose legal values have a lower bound of 1, i.e., *number of models in database* and *number of components in database*, we identify 0, the erroneous value, 1, the boundary value, and $many$, the class of values greater than 1, as the relevant value classes. For numeric characteristics whose lower bound is zero, i.e., *number of required slots for selected model* and *number of optional slots for selected model*, we identify 0 as a boundary value, 1 and *many* as other relevant classes of values. Negative values are impossible here, so we do not add a negative error choice. For numeric characteristics whose legal values have definite lower and upper-bounds, i.e., *number of optional components with selection ≠ empty* and *number of optional components with selection ≠ empty*, we identify boundary and (when possible) erroneous conditions corresponding to both lower and upper bounds.

Identifying relevant values is an important but tedious task. Test designers may improve manual selection of relevant values by using the catalog approach described in Section 13.8, which captures the informal approaches used in this section with a systematic application of catalog entries.

C. Generate Test Case Specifications A test case specification for a feature is given as a combination of values, one for each identified parameter characteristic. Unfortunately, the simple combination of all possible relevant values for each parameter characteristic results in an unmanageable number of test cases (many of which are impossible) even for simple specifications. For

[6]At this point, readers may ignore the items in square brackets, which indicate the constraints as identified in step C of the category-partition method.

Parameter: Model

Model number

malformed	[error]
not in database	[error]
valid	

Number of required slots for selected model(#SMRS)

0	[single] [property empty]
1	[property RSNE] [single]
many	[property RSNE], [property RSMANY]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[property OSNE] [single]
many	[property OSNE][property OSMANY]

Parameter: Components

Correspondence of selection with model slots

omitted slots	[error]
extra slots	[error]
mismatched slots	[error]
complete correspondence	

Number of required components with selection ≠ empty

0	[if RSNE] [error]
$<$ number of required slots	[if RSNE] [error]
$=$ number of required slots	[if RSMANY]

Number of optional components with select ≠ empty

0	
$<$ number of optional slots	[if OSNE]
$=$ number of optional slots	[if OSMANY]

Required component selection

some default	[single]
all valid	
≥ 1 incompatible with slot	
≥ 1 incompatible with another selection	
≥ 1 incompatible with model	
≥ 1 not in database	[error]

Optional component selection

some default	[si
all valid	
≥ 1 incompatible with slot	
≥ 1 incompatible with another selection	
≥ 1 incompatible with model	
≥ 1 not in database	[er

Environment element: Product database

Number of models in database (#DBM)

0	[error]
1	[single]
many	

Number of components in database (#DBC)

0	[error]
1	[single]
many	

Table 13.1: An example category-partition test specification for the the configuration checking feature of the web site of a computer vendor.

example, in the Table 13.1 we find 7 categories with 3 value classes, 2 categories with 6 value classes, and one with four value classes, potentially resulting in $3^7 \times 6^2 \times 4 = 314,928$ test cases, which would be acceptable only if the cost of executing and checking each individual test case were very small. However, not all combinations of value classes correspond to reasonable test case specifications. For example, it is not possible to create a test case from a test case specification requiring a *valid* model (a model appearing in the database) where the database contains zero models.

The category-partition method allows one to omit some combinations by indicating value classes that need not be combined with all other values. The label $[error]$ indicates a value class that need be tried only once, in combination with non-error values of other parameters. When $[error]$ constraints are considered in the category-partition specification of Table 13.1, the number of combinations to be considered is reduced to $1 \times 3 \times 3 \times 1 \times 1 \times 3 \times 5 \times 5 \times 2 \times 2 + 11 = 2711$. Note that we have treated "component not in database" as an error case, but have treated "incompatible with slot" as a normal case of an invalid configuration; once again, some judgment is required.

Although the reduction from 314,928 to 2,711 is impressive, the number of derived test cases may still exceed the budget for testing such a simple feature. Moreover, some values are not erroneous per se, but may only be useful or even valid in particular combinations. For example, the number of optional components with non-empty selection is relevant to choosing useful test cases only when the number of optional slots is greater than 1. A number of non-empty choices of required component greater than zero does not make sense if the number of required components is zero.

Erroneous combinations of valid values can be ruled out with the *property* and *if-property* constraints. The *property* constraint groups values of a single parameter characteristic to identify subsets of values with common properties. The *property* constraint is indicated with label *property PropertyName*, where *PropertyName* identifies the property for later reference. For example, property *RSNE* (required slots non-empty) in Table 13.1 groups values that correspond to non-empty sets of required slots for the parameter characteristic *Number of Required Slots for Selected Model (#SMRS)*, i.e., values *1* and *many*. Similarly, property *OSNE* (optional slots non-empty) groups non-empty values for the parameter characteristic *Number of Optional Slots for Selected Model (#SMOS)*.

The *if-property* constraint bounds the choices of values for a parameter characteristic once a specific value for a different parameter characteristic has been chosen. The *if-property* constraint is indicated with label *if PropertyName*, where *PropertyName* identifies a property defined with the *property* constraint. For example, the constraint *if RSNE* attached to values *0* and $<$ *number of required slots* of parameter characteristic *Number of required components with selection* $\neq$ *empty* limits the combination of these values with the values of the parameter characteristics *Number of Required Slots for Selected Model (#SMRS)*, i.e., values *1* and *many*, thus ruling out the illegal combination of values *0* or $<$ *number of required slots* for *Number of required com-*

ponents with selection $\neq$ empty with value *0* for *Number of Required Slots for Selected Model (#SMRS)*. Similarly, the *if OSNE* constraint limits the combinations of values of the parameter characteristics *Number of optional components with selection $\neq$ empty* and *Number of Optional Slots for Selected Model (#SMOS)*.

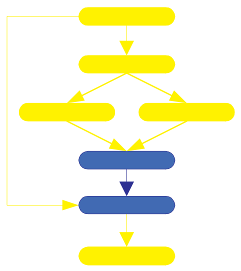
The *property* and *if-property* constraints introduced in Table 13.1 further reduce the number of combinations to be considered to $1 \times 3 \times 1 \times 1 \times (3+2+1) \times 5 \times 5 \times 2 \times 2 + 11 = 1811$. (Exercise Ex13.4 discusses derivation of this number.)

The number of combinations can be further reduced by iteratively adding *property* and *if-property* constraints and by introducing the new *single* constraint, which is indicated with label *single* and acts like the *error* constraint, i.e., it limits the number of occurrences of a given value in the selected combinations to *1*.

Introducing new *property*, *if-property*, and *single* constraints further does not rule out erroneous combinations, but reflects the judgment of the test designer, who decides how to restrict the number of combinations to be considered by identifying single values (*single* constraint) or combinations (*property* and *if-property* constraints) that are less likely to need thorough test according to the test designer's judgment.

The *single* constraints introduced in Table 13.1 reduces the number of combinations to be considered to $1 \times 1 \times 1 \times 1 \times 1 \times 3 \times 4 \times 4 \times 1 \times 1 + 19 = 67$, which may be a reasonable tradeoff between costs and quality for the considered functionality. The number of combinations can also be reduced by applying combinatorial techniques, as explained in the next section.

The set of combinations of values for the parameter characteristics can be turned into test case specifications by simply instantiating the identified combinations. Table 13.2 shows an excerpt of test case specifications. The error tag in the last column indicates test cases specifications corresponding to the *error* constraint. Corresponding test cases should produce an error indication. A dash indicates no constraints on the choice of values for the parameter or environment element.



Choosing meaningful names for parameter characteristics and value classes allows (semi)automatic generation of test case specifications.

13.5 The Combinatorial Approach

However one obtains sets of value classes for each parameter characteristic, the next step in producing test case specifications is selecting combinations of classes for testing. A simple approach is to exhaustively enumerate all possible combinations of classes, but the number of possible combinations rapidly explodes.

Some methods, such as the category-partition method described in the previous section, take exhaustive enumeration as a base approach to generating combinations, but allow the test designer to add constraints that limit

Case	Model	Components	Product database	
TC1	malformed model number	-	-	ERROR
TC2	model number N not in database	-	not including model number	ERROR
...				
TCi	valid model number N with 0 required slots and 0 optional slots	-	includes N	VALID
...				
TCj	valid model number N with K required slots and M optional slots	K valid required components and M valid optional components	includes N and all components	VALID
...				
TCx	-	-	no models in database	ERROR
...				
TCy	-	-	no components in database	ERROR

Table 13.2: An excerpt of test case specifications derived from the value classes given in Table 13.1

growth in the number of combinations. This can be a reasonable approach when the constraints on test case generation reflect real constraints in the application domain, and eliminate many redundant combinations (for example, the "error" entries in category-partition testing). It is less satisfactory when, lacking real constraints from the application domain, the test designer is forced to add arbitrary constraints (e.g., "single" entries in the category-partition method) whose sole purpose is to reduce the number of combinations.

Consider the parameters that control the Chipmunk web-site display, shown in Table 13.3. Exhaustive enumeration produces 432 combinations, which is too many if the test results (e.g., judging readability) involve human judgment. While the test designer might hypothesize some constraints, such as observing that monochrome displays are limited mostly to hand-held devices, radical reductions require adding several "single" and "property" constraints without any particular rationale.

Exhaustive enumeration of all n-way combinations of value classes for n parameters, on the one hand, and coverage of individual classes, on the other, are only the extreme ends of a spectrum of strategies for generating combinations of classes. Between them lie strategies that generate all pairs of classes for different parameters, all triples, and so on. When it is reasonable to expect some potential interaction between parameters (so coverage of individual value classes is deemed insufficient), but covering all combinations is impractical, an attractive alternative is to generate k-way combinations for $k < n$, typically pairs or triples.

How much does generating possible pairs of classes save, compared to

Display Mode	**Language**	**Fonts**
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
	Portuguese	

Color	**Screen size**	
Monochrome	Hand-held	
Color-map	Laptop	
16-bit	Full-size	
True-color		

Table 13.3: Parameters and values controlling Chipmunk web-site display

generating all combinations? We have already observed that the number of all combinations is the product of the number of classes for each parameter, and that this product grows exponentially with the number of parameters. It turns out that the number of combinations needed to cover all possible pairs of values grows only logarithmically with the number of parameters — an enormous saving.

A simple example may suffice to gain some intuition about the efficiency of generating tuples that cover pairs of classes, rather than all combinations. Suppose we have just the three parameters *display mode, screen size,* and *fonts* from Table 13.3. If we consider only the first two, *display mode* and *screen size*, the set of all pairs and the set of all combinations are identical, and contain $3 \times 3 = 9$ pairs of classes. When we add the third parameter, *fonts*, generating all combinations requires combining each value class from *fonts* with every pair of *display mode* $\times$ *screen size*, a total of 27 tuples; extending from n to $n+1$ parameters is multiplicative. However, if we are generating pairs of values from *display mode*, *screen size*, and *fonts*, we can add value classes of *fonts* to existing elements of *display mode* $\times$ *screen size* in a way that covers all the pairs of *fonts* $\times$ *screen size* and all the pairs of *fonts* $\times$ *display mode* without increasing the number of combinations at all (see Table 13.4). The key is that each tuple of three elements contains three pairs, and by careful selecting value classes of the tuples we can make each tuple cover up to three different pairs.

Table 13.5 shows 17 tuples that cover all pairwise combinations of value classes of the five parameters. The entries not specified in the table ("–") correspond to open choices. Each of them can be replaced by any legal value for the corresponding parameter. Leaving them open gives more freedom for selecting test cases.

Generating combinations that efficiently cover all pairs of classes (or triples, or ...) is nearly impossible to perform manually for many parameters with many value classes (which is, of course, exactly when one really needs to use

Display mode × *Screen size*		*Fonts*
Full-graphics	Hand-held	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Full-size	Document-loaded
Text-only	Hand-held	Standard
Text-only	Laptop	Document-loaded
Text-only	Full-size	Minimal
Limited-bandwidth	Hand-held	Document-loaded
Limited-bandwidth	Laptop	Minimal
Limited-bandwidth	Full-size	Standard

Table 13.4: Covering all pairs of value classes for three parameters by extending the cross-product of two parameters

the approach). Fortunately, efficient heuristic algorithms exist for this task, and they are simple enough to incorporate in tools.[7]

The tuples in Table 13.5 cover all pairwise combinations of value choices for parameters. In many cases not all choices may be allowed. For example, the specification of the Chipmunk web-site display may indicate that monochrome displays are limited to hand-held devices. In this case, the tuples covering the pairs ⟨*Monochrome*, *Laptop*⟩ and ⟨*Monochrome*, *Full-size*⟩, i.e., the fifth and ninth tuples of Table 13.5, would not correspond to legal inputs. We can restrict the set of legal combinations of value classes by adding suitable constraints. Constraints can be expressed as tuples with wild-card characters to indicate any possible value class. For example, the constraints

$$\langle *, *, *, \mathit{Monochrome}, \mathit{Laptop} \rangle$$

$$\langle *, *, *, \mathit{Monochrome}, \mathit{Full\text{-}size} \rangle$$

indicates that tuples that contain the pair ⟨*Monochrome*, *Hand-held*⟩ as values for the fourth and fifth parameter are not allowed in the relation of Table 13.3. Tuples that cover all pairwise combinations of value classes without violating the constraints can be generated by simply removing the illegal tuples and adding legal tuples that cover the removed pairwise combinations. Open choices must be bound consistently in the remaining tuples, e.g., tuple

$$\langle \mathit{Portuguese}, \mathit{Monochrome}, \mathit{Text\text{-}only}, \text{-}, \text{-} \rangle$$

must become

$$\langle \mathit{Portuguese}, \mathit{Monochrome}, \mathit{Text\text{-}only}, \text{-}, \mathit{Hand\text{-}held} \rangle$$

Constraints can also be expressed with sets of tables to indicate only the legal combinations, as illustrated in Table 13.6, where the first table indicates

[7]Exercise Ex13.12 discusses the problem of computing suitable combinations to cover all pairs.

Language	*Color*	*Display Mode*	*Fonts*	*Screen Size*
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	–	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	–
French	True-color	–	–	Hand-held
Spanish	Monochrome	–	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	–	Hand-held
Portuguese	Monochrome	Text-only	–	–
Portuguese	Color-map	–	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Table 13.5: Covering all pairs of value classes for the five parameters

that the value class *Hand-held* for parameter *Screen* can be combined with any value class of parameter *Color*, including *Monochrome*, while the second table indicates that the value classes *Laptop* and *Full-size* for parameter *Screen size* can be combined with all values classes but *Monochrome* for parameter *Color*.

If constraints are expressed as a set of tables that give only legal combinations, tuples can be generated without changing the heuristic. Although the two approaches express the same constraints, the number of generated tuples can be different, since different tables may indicate overlapping pairs and thus result in a larger set of tuples. Other ways of expressing constraints may be chosen according to the characteristics of the specifications and the preferences of the test designer.

So far we have illustrated the combinatorial approach with pairwise coverage. As previously mentioned, the same approach can be applied for triples or larger combinations. Pairwise combinations may be sufficient for some subset of the parameters, but not enough to uncover potential interactions among other parameters. For example, in the Chipmunk display example, the fit of text fields to screen areas depends on the combination of language, fonts, and screen size. Thus, we may prefer exhaustive coverage of combinations of these three parameters, but be satisfied with pairwise coverage of other parameters. In this case, we first generate tuples of classes from the parameters to be most thoroughly covered, and then extend these with the

Hand-held devices

Display Mode	Language	Fonts
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
	Portuguese	

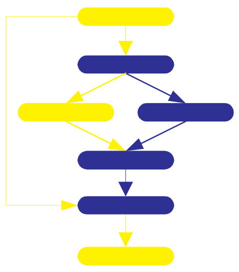
Color	Screen size
Color-map	Hand-held
16-bit	
True-color	

Laptop and Full-size devices

Display Mode	Language	Fonts
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
	Portuguese	

Color	Screen size
Monochrome	Laptop
Color-map	Full size
16-bit	
True-color	

Table 13.6: Pairs of tables that indicate valid value classes for the Chipmunk web-site display controller



parameters which require less coverage.[8]

13.6 Testing Decision Structures

The combinatorial approaches described above primarily select combinations of orthogonal choices. They can accommodate constraints among choices, but their strength is in generating combinations of (purportedly) independent choices. Some specifications, formal and informal, have a structure that emphasizes the way particular combinations of parameters or their properties determine which of several potential outcomes is chosen. Results of the computation may be determined by boolean predicates on the inputs. In some cases, choices are specified explicitly as boolean expressions. More often, choices are described either informally or with tables or graphs that can assume various forms. When such a decision structure is present, it can play a part in choosing combinations of values for testing.

For example, the informal specification of Figure 13.4 describes outputs that depend on type of account (either educational, or business, or individual), amount of current and yearly purchases, and availability of special prices. These can be considered as boolean conditions, e.g., the condition *educational account* is either true or false (even if the type of account is actually represented in some other manner). Outputs can be described as boolean expressions over the inputs, e.g., the output *no discount* can be associated with the boolean expression

$$\begin{array}{ll}
 & \text{individual account} \\
\wedge\neg & \text{current purchase} > \text{tier 1 individual threshold} \\
\wedge\neg & \text{special offer price} < \text{individual scheduled price} \\
\vee & \text{business account} \\
\wedge\neg & \text{current purchase} > \text{tier 1 business threshold} \\
\wedge\neg & \text{current purchase} > \text{tier 1 business yearly threshold} \\
\wedge\neg & \text{special offer price} < \text{business scheduled price}
\end{array}$$

When functional specifications can be given as boolean expressions, a good test suite should exercise at least the effects of each elementary condition occurring in the expression. (In ad hoc testing, it is common to miss a bug in one elementary condition by choosing test cases in which it is "masked" by other conditions.) For simple conditions, we might derive test case specifications for all possible combinations of truth values of the elementary conditions. For complex formulas, testing all 2^n combinations of n elementary conditions is apt to be too expensive; we can select a much smaller subset of combinations that checks the effect of each elementary condition. A good way of exercising all elementary conditions with a limited number of test cases is deriving a set of combinations such that each elementary condition can be shown to independently affect the outcome.

[8]See exercise Ex13.14 for additional details.

Pricing: The *pricing* function determines the adjusted price of a configuration for a particular customer. The scheduled price of a configuration is the sum of the scheduled price of the model and the scheduled price of each component in the configuration. The adjusted price is either the scheduled price, if no discounts are applicable, or the scheduled price less any applicable discounts.

There are three price schedules and three corresponding discount schedules, *Business*, *Educational*, and *Individual*. The Business price and discount schedules apply only if the order is to be charged to a business account in good standing. The Educational price and discount schedules apply to educational institutions. The Individual price and discount schedules apply to all other customers. Account classes and rules for establishing business and educational accounts are described further in [. . .].

A discount schedule includes up to three discount levels, in addition to the possibility of "no discount." Each discount level is characterized by two threshold values, a value for the current purchase (configuration schedule price) and a cumulative value for purchases over the preceding 12 months (sum of adjusted price).

Educational prices The adjusted price for a purchase charged to an educational account in good standing is the scheduled price from the educational price schedule. No further discounts apply.

Business account discounts Business discounts depend on the size of the current purchase as well as business in the preceding 12 months. A tier 1 discount is applicable if the scheduled price of the current order exceeds the tier 1 current order threshold, or if total paid invoices to the account over the preceding 12 months exceeds the tier 1 year cumulative value threshold. A tier 2 discount is applicable if the current order exceeds the tier 2 current order threshold, or if total paid invoices to the account over the preceding 12 months exceeds the tier 2 cumulative value threshold. A tier 2 discount is also applicable if both the current order and 12 month cumulative payments exceed the tier 1 thresholds.

Individual discounts Purchase by individuals and by others without an established account in good standing are based on current value alone (not on cumulative purchases). A tier 1 individual discount is applicable if the scheduled price of the configuration in the current order exceeds the the tier 1 current order threshold. A tier 2 individual discount is applicable if the scheduled price of the configuration exceeds the tier 2 current order threshold.

Special-price non-discountable offers Sometimes a complete configuration is offered at a special, non-discountable price. When a special, non-discountable price is available for a configuration, the adjusted price is the non-discountable price or the regular price after any applicable discounts, whichever is less.

Figure 13.4: The functional specification of feature *pricing* of the Chipmunk web site.

Terminology: Predicates and Conditions

A *predicate* is a function with a boolean (**True** or **False**) value. When the input argument of the predicate is clear, particularly when it describes some property of the input of a program, we often leave it implicit. For example, the actual representation of account types in an information system might be as three-letter codes, but in a specification we may not be concerned with that representation — we know only that there is some predicate *educational-account* which is either **True** or **False**.

An *elementary condition* is a single predicate that cannot be decomposed further. A *complex condition* is made up of elementary conditions, combined with boolean connectives.

The *boolean connectives* include "and" ($\wedge$), "or" ($\vee$), "not" ($\neg$), and several less common derived connectives such as "implies" and "exclusive or."

A systematic approach to testing boolean specifications consists in first constructing a model of the boolean specification and then applying test criteria to derive test case specifications.

STEP 1: derive a model of the decision structure We can produce different models of the decision structure of a specification, depending on the original specification and on the technique we use for deriving test cases. For example, if the original specification prescribes a sequence of decisions, either in a program-like syntax or perhaps as a decision tree, we may decide not to derive a different model but rather treat it as a conditional statement. Then we can directly apply the methods described in Chapter 14 for structural testing, i.e., basic condition, compound condition, or modified condition/decision adequacy criteria. On the other hand, if the original specification is expressed informally as in Figure 13.4, we can transform it into either a boolean expression or a graph or a tabular model before applying a test case generation technique.

Techniques for deriving test case specifications from decision structures were originally developed for graph models, and in particular cause effect graphs, which have been used since the early seventies. Cause-effect graphs are tedious to derive and do not scale well to complex specifications. Tables, on the other hand, are easy to work with and scale well.

A decision structure can be represented with a *decision table* where rows correspond to elementary conditions and columns correspond to combinations of elementary conditions. The last row of the table indicates the expected outputs. Cells of the table are labeled either **true**, **false**, or **don't care** (usually written **–**), to indicate the truth value of the elementary condition. Thus, each column is equivalent to a logical expression joining the required values (negated, in the case of **false** entries) and omitting the elementary conditions with **don't care** values.[9]

[9]The set of columns sharing a label is therefore equivalent to a logical expression in sum-of-

Decision tables are completed with a set of *constraints* that limit the possible combinations of elementary conditions. A constraint language can be based on boolean logic. Often it is useful to add some shorthand notations for common conditions that are tedious to express with the standard connectives, such as *at-most-one(C1, ..., Cn)* and *exactly-one(C1, ..., Cn)*.

Figure 13.5 shows the decision table for the functional specification of feature *pricing* of the Chipmunk web site presented in Figure 13.4.

The informal specification of Figure 13.4 identifies three customer profiles: *educational*, *business*, and *individual*. Table 13.5 has only rows *educational* and *business*. The choice *individual* corresponds to the combination **false**, **false** for choices *educational* and *business*, and is thus redundant. The informal specification of Figure 13.4 indicates different discount policies depending on the relation between the current purchase and two progressive thresholds for the current purchase and the yearly cumulative purchase. These cases correspond to rows 3 through 6 of table 13.5. Conditions on thresholds that do not correspond to individual rows in the table can be defined by suitable combinations of values for these rows. Finally, the informal specification of Figure 13.4 distinguishes the cases in which special offer prices do not exceed either the scheduled or the tier 1 or tier 2 prices. Rows 7 through 9 of the table, suitably combined, capture all possible cases of special prices without redundancy.

Constraints formalize the compatibility relations among different elementary conditions listed in the table: *Educational* and *Business* accounts are exclusive; A cumulative purchase exceeding threshold tier 2, also exceeds threshold tier 1; a yearly purchase exceeding threshold tier 2, also exceeds threshold tier 1; a cumulative purchase not exceeding threshold tier 1 does not exceed threshold tier 2; a yearly purchase not exceeding threshold tier 1 does not exceed threshold tier 2; a special offer price not exceeding threshold tier 1 does not exceed threshold tier 2; and finally, a special offer price exceeding threshold tier 2 exceeds threshold tier 1.

STEP 2: derive test case specifications from a model of the decision structure Different criteria can be used to generate test suites of differing complexity from decision tables.

The *basic condition adequacy criterion* requires generation of a test case specification for each column in the table, and corresponds to the intuitive principle of generating a test case to produce each possible result. *Don't care* entries of the table can be filled out arbitrarily, so long as constraints are not violated.

Δ Basic Condition Coverage

The *compound condition adequacy criterion* requires a test case specification for each combination of truth values of elementary conditions. The compound condition adequacy criterion generates a number of cases exponential in the number of elementary conditions, and can thus be applied only to small sets of elementary conditions.

Δ Compound Condition Coverage

products form.

	Education		**Individual**						**Business**											
Edu.	T	T	F	F	F	F	F	F	-	-	-	-	-	-	-	-	-	-	-	-
Bus.	-	-	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T
$CP > CT1$	-	-	F	F	T	T	-	-	F	F	T	T	F	F	T	T	-	-	-	-
$YP > YT1$	-	-	-	-	-	-	-	-	F	F	F	F	T	T	T	T	-	-	-	-
$CP > CT2$	-	-	-	-	F	F	T	T	-	-	F	F	-	-	-	-	T	T	-	-
$YP > YT2$	-	-	-	-	-	-	-	-	-	-	-	-	F	F	-	-	-	-	T	T
$SP > Sc$	F	T	F	T	-	-	-	-	F	T	-	-	-	-	-	-	-	-	-	-
$SP > T1$	-	-	-	-	F	T	-	-	-	-	F	T	F	T	-	-	-	-	-	-
$SP > T2$	-	-	-	-	-	-	F	T	-	-	-	-	-	-	F	T	F	T	F	T
Out	Edu	SP	ND	SP	T1	SP	T2	SP	ND	SP	T1	SP	T1	SP	T2	SP	T2	SP	T2	SP

Constraints

at-most-one(Edu, Bus)	at-most-one($YP < YT1$, $YP > YT2$)
$YP > YT2 \Rightarrow YP > YT1$	at-most-one($CP < CT1$, $CP > CT2$)
$CP > CT2 \Rightarrow CP > CT1$	at-most-one($SP < T1$, $SP > T2$)
$SP > T2 \Rightarrow SP > T1$	

Abbreviations

Edu.	Educational account
Bus.	Business account
$CP > CT1$	Current purchase greater than threshold 1
$YP > YT1$	Year cumulative purchase greater than threshold 1
$CP > CT2$	Current purchase greater than threshold 2
$YP > YT2$	Year cumulative purchase greater than threshold 2
$SP > Sc$	Special Price better than scheduled price
$SP > T1$	Special Price better than tier 1
$SP > T2$	Special Price better than tier 2

Edu	Educational price
ND	No discount
T1	Tier 1
T2	Tier 2
SP	Special Price

Figure 13.5: The decision table for the functional specification of feature *pricing* of the Chipmunk web site of Figure 13.4.

For the *modified condition/decision adequacy criterion* (MC/DC), each column in the table represents a test case specification. In addition, for each of the original columns, MC/DC generates new columns by modifying each of the cells containing **True** or **False**. If modifying a truth value in one column results in a test case specification consistent with an existing column (agreeing in all places where neither is *don't care*), the two test cases are represented by one merged column, provided they can be merged without violating constraints.

Δ Modified Condition/Decision Coverage

The MC/DC criterion formalizes the intuitive idea that a thorough test suite would not only test *positive* combinations of values, i.e., combinations that lead to specified outputs, but also *negative* combinations of values, i.e., combinations that differ from the specified ones and thus should produce different outputs, in some cases among the specified ones, in some other cases leading to error conditions.

Applying MC/DC to column 1 of table 13.5 generates two additional columns: one for *Educational Account* = **false** and *Special Price better than scheduled price* = **false**, and the other for *Educational Account* = **true** and *Special Price better than scheduled price* = **true**. Both columns are already in the table (columns 3 and 2, respectively) and thus need not be added.

Similarly, from column 2, we generate two additional columns corresponding to *Educational Account* = **false** and *Special Price better than scheduled price* = **true**, and *Educational Account* = **true** and *Special Price better than scheduled price* = **false**, also already in the table.

The generation of a new column for each possible variation of the boolean values in the columns, varying exactly one value for each new column, produces 78 new columns, 21 of which can be merged with columns already in the table. Figure 13.6 shows a table obtained by suitably joining the generated columns with the existing ones. Many **don't care** cells from the original table are assigned either **true** or **false** values, to allow merging of different columns or to obey constraints. The few **don't-care** entries left can be set randomly to obtain a complete test case specification.

There are many ways of merging columns that generate different tables. The table in Figure 13.6 may not be the optimal one, i.e., the one with the fewest columns. The objective in test design is not to find an optimal test suite, but rather to produce a cost effective test suite with an acceptable trade-off between the cost of generating and executing test cases and the effectiveness of the tests.

The table in Figure 13.6 fixes the entries as required by the constraints, while the initial table in Figure 13.5 does not. Keeping constraints separate from the table corresponding to the initial specification increases the number of **don't care** entries in the original table, which in turn increases the opportunity for merging columns when generating new cases with the MC/DC criterion. For example, if *business account* = **false**, the constraint at-most-one(Edu, Bus) can be satisfied by assigning either **true** or **false** to entry *educational account*. Fixing either choice prematurely may later make merging with a newly generated column impossible.

Edu.	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T	F	-
Bus.	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F
CP > CT1	T	T	F	F	T	T	T	T	F	F	T	T	F	F	T	T	T	T	F	F	F	F	T	-	-	F
YP > YT1	F	-	F	-	-	F	T	T	F	F	F	F	T	T	T	T	F	F	T	T	T	-	-	-	T	T
CP > CT2	F	F	F	F	F	F	T	T	F	F	F	F	F	F	F	F	T	T	F	F	F	F	T	T	F	F
YP > YT2	-	-	-	-	-	-	-	-	-	-	-	-	F	F	F	F	-	-	T	T	F	-	-	-	T	F
SP > Sc	F	T	F	T	F	T	-	-	F	T	F	-	F	T	-	T	-	T	-	T	F	T	-	-	-	-
SP > T1	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	-	-	T	T	T
SP > T2	F	-	F	-	F	-	F	T	F	-	F	-	F	-	F	T	F	T	F	T	F	F	F	T	T	-
Out	Edu	SP	ND	SP	T1	SP	T2	SP	ND	SP	T1	SP	T1	SP	T2	SP	T2	SP	T2	SP	Edu	SP	Edu	SP	SP	SP

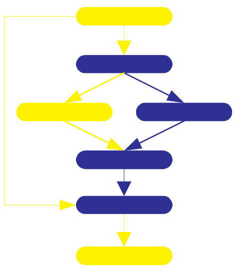
Abbreviations

Edu.	Educational account
Bus.	Business account
CP > CT1	Current purchase greater than threshold 1
YP > YT1	Year cumulative purchase greater than threshold 1
CP > CT2	Current purchase greater than threshold 2
YP > YT2	Year cumulative purchase greater than threshold 2
SP > Sc	Special Price better than scheduled price
SP > T1	Special Price better than tier 1
SP > T2	Special Price better than tier 2

Edu	Educational price
ND	No discount
T1	Tier 1
T2	Tier 2
SP	Special Price

Figure 13.6: The set of test cases generated for feature *pricing* of the Chipmunk web site applying the modified adequacy criterion.

13.7 Deriving Test Cases from Control and Data Flow Graphs



Functional specifications are seldom given as flow graphs, but sometimes they describe a set of mutually dependent steps to be executed in a given (partial) order, and can thus be modeled with flow graphs.

For example the specification of Figure 13.7 describes the Chipmunk functionality that processes shipping orders. The specification indicates a set of steps to check for the validity of fields in the order form. Type and validity of some of the values depend on other fields in the form. For example, shipping methods are different for domestic and international customers, and allowed methods of payment depend on the kind of customer.

The informal specification of Figure 13.7 can be modeled with a control flow graph, where the nodes represent computations and branches model flow of control consistently with the dependencies among computations, as illustrated in Figure 13.8. Given a control or a data flow graph model, we can generate test case specifications using the criteria originally proposed for structural testing and described in Chapters **??** and **??**.

Control flow testing criteria require test cases that exercise all the elements of a particular type in a graph. The *node testing* adequacy criterion requires each node to be exercised at least once and corresponds to the statement testing structural adequacy criterion. It is easy to verify that test *T-node* causes all nodes of the control flow graph of Figure 13.8 to be traversed and thus satisfies the node adequacy criterion.

Δ Node Adequacy Criterion

T-node

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Air	Ind	CC	–	No (abort)

Abbreviations:

Too small	CostOfGoods < MinOrder ?
Ship where	Shipping address, Int = international, Dom = domestic
Ship how	Air = air freight, Land = land freight
Cust type	Bus = business, Edu = educational, Ind = individual
Pay method	CC = credit card, Inv = invoice
Same addr	Billing address = shipping address ?
CC Valid	Credit card information passes validity check?

The *branch testing* adequacy criterion requires each branch to be exercised at least once, i.e., each edge of the graph to be traversed for at least one test case. Test *T-branch* covers all branches of the control flow graph of Figure 13.8 and thus satisfies the branch adequacy criterion.

Δ Branch Adequacy Criterion

Process shipping order: The *Process shipping order* function checks the validity of orders and prepares the receipt.

A valid order contains the following data:

cost of goods If the cost of goods is less than the minimum processable order (*MinOrder*) then the order is invalid.

shipping address The address includes name, address, city, postal code, and country.

preferred shipping method If the address is domestic, the shipping method must be either *land freight*, or *expedited land freight*, or *overnight air*. If the address is international, the shipping method must be either *air freight* or *expedited air freight*; a shipping cost is computed based on address and shipping method.

type of customer which can be *individual*, *business* or *educational*

preferred method of payment Individual customers can use only credit cards, while business and educational customers can choose between *credit card* and *invoice*.

card information if the method of payment is credit card, fields credit card number, name on card, expiration date, and billing address, if different than shipping address, must be provided. If credit card information is not valid the user can either provide new data or abort the order.

The outputs of *Process shipping order* are

validity Validity is a boolean output which indicates whether the order can be processed.

total charge The total charge is the sum of the value of goods and the computed shipping costs (only if validity = true).

payment status if all data are processed correctly and the credit card information is valid or the payment is invoice, payment status is set to valid, the order is entered and a receipt is prepared; otherwise validity = false.

Figure 13.7: The functional specification of feature *process shipping order* of the Chipmunk web site.

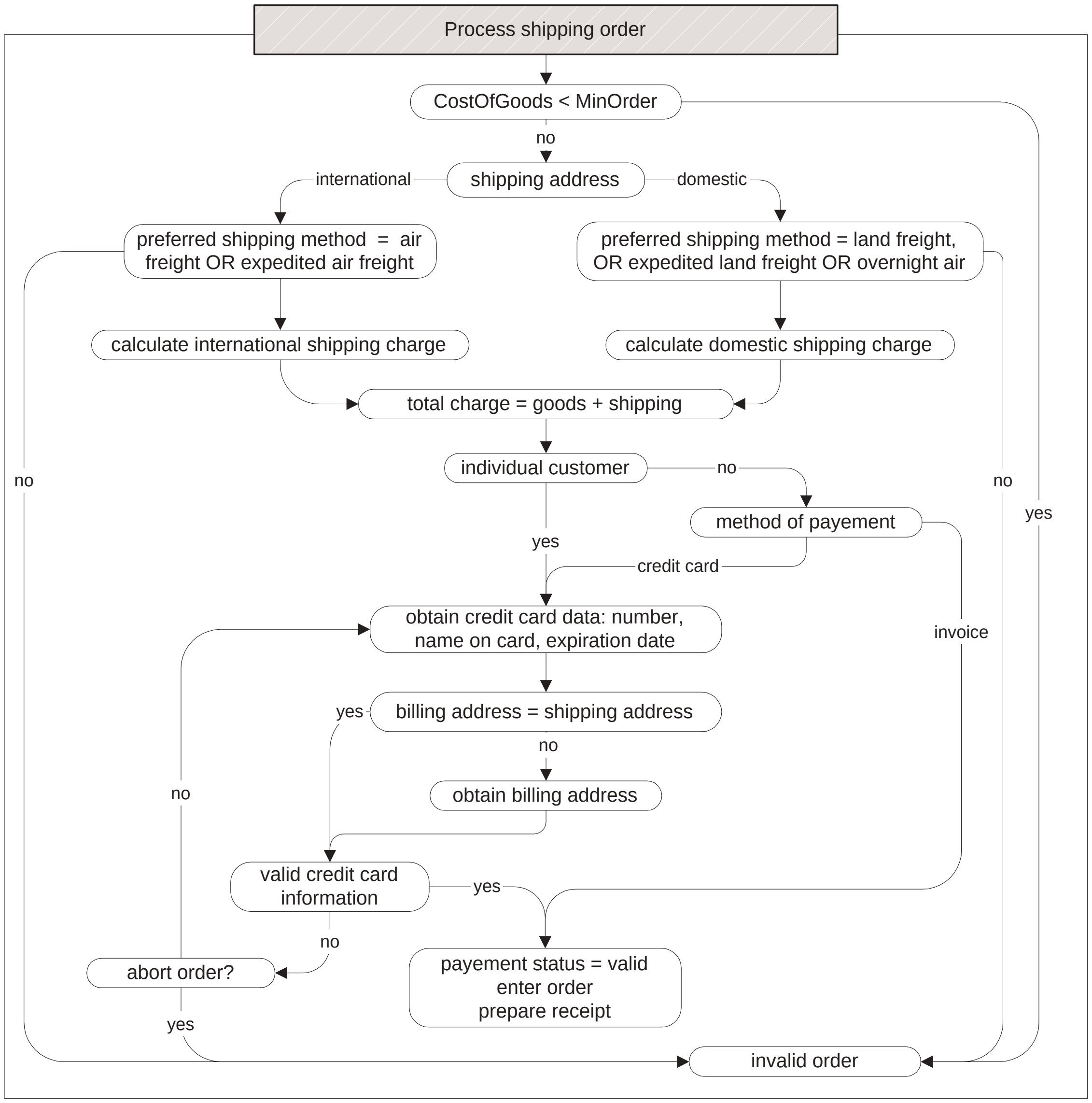
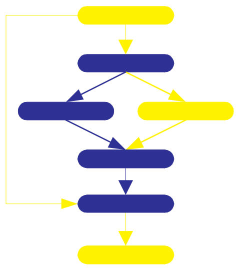

Process shipping order
CostOfGoods < MinOrder
no
yes
shipping address
international
domestic
preferred shipping method = air freight OR expedited air freight
preferred shipping method = land freight, OR expedited land freight OR overnight air
no
no
calculate international shipping charge
calculate domestic shipping charge
total charge = goods + shipping
individual customer
no
yes
method of payement
credit card
invoice
obtain credit card data: number, name on card, expiration date
billing address = shipping address
yes
no
obtain billing address
no
valid credit card information
yes
no
abort order?
yes
payement status = valid
enter order
prepare receipt
invalid order

Figure 13.8: The control flow graph corresponding to functionality *Process shipping order* of Figure 13.7

T-branch

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	–	–	–	–
TC-3	Yes	–	–	–	–	–	–
TC-4	No	Dom	Air	–	–	–	–
TC-5	No	Int	Land	–	–	–	–
TC-6	No	–	–	Edu	Inv	–	–
TC-7	No	–	–	–	CC	Yes	–
TC-8	No	–	–	–	CC	–	No (abort)
TC-9	No	–	–	–	CC	–	No (no abort)

Abbreviations:
(as above)



In principle, other test adequacy criteria described in Chapter 14 can be applied to more control structures derived from specifications, but in practice a good specification should rarely result in a complex control structure, since a specification should abstract details of processing.

13.8 Catalog Based Testing

The test design techniques described above require judgment in deriving value classes. Over time, an organization can build experience in making these judgments well. Gathering this experience in a systematic collection can speed up the process and routinize many decisions, reducing human error. Catalogs capture the experience of test designers by listing all cases to be considered for each possible type of variable that represents logical inputs, outputs, and status of the computation. For example, if the computation uses a variable whose value must belong to a range of integer values, a catalog might indicate the following cases, each corresponding to a relevant test case:

1. The element immediately preceding the lower bound of the interval
2. The lower bound of the interval
3. A non-boundary element within the interval
4. The upper bound of the interval
5. The element immediately following the upper bound

The catalog would in this way cover the intuitive cases of erroneous conditions (cases 1 and 5), boundary conditions (cases 2 and 4), and normal conditions (case 3).

The catalog based approach consists in *unfolding* the specification, i.e., decomposing the specification into elementary items, deriving an initial set of test case specifications from pre-conditions, post-conditions, and definitions, and completing the set of test case specifications using a suitable test catalog.

STEP 1: identify elementary items of the specification The initial specification is transformed into a set of elementary items that have to be tested. Elementary items belong to a small set of basic types:

Pre-conditions represent the conditions on the inputs that must be satisfied before invocation of the unit under test. Preconditions may be checked either by the unit under test (*validated preconditions*) or by the caller (*assumed preconditions*).

Post-conditions describe the result of executing the unit under test.

Variables indicate the elements on which the unit under test operates. They can be input, output, or intermediate values.

Operations indicate the main operations performed on input or intermediate variables by the unit under test

Definitions are shorthand used in the specification

As in other approaches that begin with an informal description, it is not possible to give a precise recipe for extracting the significant elements. The result will depend on the capability and experience of the test designer.

Consider the informal specification of a function for converting URL-encoded form data into the original data entered through an html form. An informal specification is given in Figure 13.7.[10]

The informal description of cgi_decode uses the concept of hexadecimal digit, hexadecimal escape sequence, and element of a cgi encoded sequence. This leads to the identification of the following three definitions:

DEF 1 *hexadecimal digits* are: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e', 'f'

DEF 2 a *CGI-hexadecimal* is a sequence of three characters: '`%xy`', where `x` and `y` are hexadecimal digits

DEF 3 a *CGI item* is either an alphanumeric character, or character '+', or a CGI-hexadecimal

In general, every concept introduced in the description as a support for defining the problem can be represented as a definition.

The description of cgi_decode mentions some elements that are inputs and outputs of the computation. These are identified as the following variables:

[10]The informal specification is ambiguous and inconsistent, i.e., it is the kind of spec one is most likely to encounter in practice.

cgi_decode: Function cgi_decode translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) of most web servers.

CGI translates spaces to '+', and translates most other non-alphanumeric characters to hexadecimal escape sequences. `cgi_decode` maps '+' to ' ', "%xy" (where x and y are hexadecimal digits) to to the corresponding ASCII character, and other alphanumeric characters to themselves.

INPUT: encoded A string of characters, representing the input CGI sequence. It can contain:

- alphanumeric characters
- the character '+'
- the substring '%xy', where x and y are hexadecimal digits.

encoded is terminated by a null character.

OUTPUT: decoded A string containing the plain ASCII characters corresponding to the input CGI sequence.

- Alphanumeric characters are copied into the output in the corresponding position
- A blank is substituted for each '+' character in the input.
- A single ASCII character with hexadecimal value xy_{16} is substituted for each substring '%xy' in the input.

OUTPUT: return value `cgi_decode` returns

- 0 for success
- 1 if the input is malformed

Table 13.7: An informal (and imperfect) specification of function cgi-decode

VAR 1 *Encoded*: string of ASCII characters

VAR 2 *Decoded*: string of ASCII characters

VAR 3 *return value*: Boolean

Note the distinction between a variable and a definition. *Encoded* and *decoded* are actually used or computed, while *hexadecimal digits*, *CGI-hexadecimal*, and *CGI item* are used to describe the elements but are not objects in their own right. Although not strictly necessary for the problem specification, explicit identification of definitions can help in deriving a richer set of test cases.

The description of cgi_decode indicates some conditions that must be satisfied upon invocation, represented by the following preconditions:

PRE 1 *(Assumed)* the input string *Encoded* is a null-terminated string of characters.

PRE 2 *(Validated)* the input string *Encoded* is a sequence of CGI items.

In general, preconditions represent all the conditions that should be true for the intended functioning of a module. A condition is labeled as *validated* if it is checked by the module (in which case a violation has a specified effect, e.g., raising an exception or returning an error code). *Assumed* preconditions must be guaranteed by the caller, and the module does not guarantee a particular behavior in case they are violated.

The description of cgi_decode indicates several possible results. These can be represented as a set of postconditions:

POST 1 if the input string *Encoded* contains alphanumeric characters, they are copied to the corresponding position in the output string.

POST 2 if the input string *Encoded* contains characters '+', they are replaced by ASCII SPACE characters in the corresponding positions in the output string.

POST 3 if the input string *Encoded* contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.

POST 4 if the input string *Encoded* is a valid sequence, cgi_decode returns 0.

POST 5 if the input string *Encoded* contains a malformed CGI-hexadecimal, i.e., a substring '%xy', where either x or y is absent or are not hexadecimal digits, cgi_decode returns 1

POST 6 if the input string *Encoded* contains any illegal character, cgi_decode returns a positive value.

The postconditions should, together, capture all the expected outcomes of the module under test. When there are several possible outcomes, it is possible to capture them all in one complex postcondition or in several simple

PRE 1	(*Assumed*) the input string Encoded is a null-terminated string of characters
PRE 2	(*Validated*) the input string Encoded is a sequence of CGI items
POST 1	if the input string Encoded contains alphanumeric characters, they are copied to the output string in the corresponding positions.
POST 2	if the input string Encoded contains characters '+', they are replaced in the output string by ASCII SPACE characters in the corresponding positions
POST 3	if the input string Encoded contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.
POST 4	if the input string Encoded is well-formed, cgi-decode returns 0
POST 5	if the input string Encoded contains a malformed CGI hexadecimal, i.e., a substring '%xy', where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1
POST 6	if the input string Encoded contains any illegal character, cgi_decode returns a positive value
VAR 1	Encoded: a string of ASCII characters
VAR 2	Decoded: a string of ASCII characters
VAR 3	Return value: a boolean
DEF 1	hexadecimal digits are ASCII characters in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']
DEF 2	CGI-hexadecimals are sequences "%xy", where x and y are hexadecimal digits
DEF 3	A CGI item is an alphanumeric character, or '+', or a CGI-hexadecimal
OP 1	Scan Encoded

Table 13.8: Elementary items of specification cgi-decode

postconditions; here we have chosen a set of simple contingent postconditions, each of which captures one case.

Although the description of cgi_decode does not mention explicitly how the results are obtained, we can easily deduce that it will be necessary to scan the input sequence. This is made explicit in the following operation:

OP 1 Scan the input string *Encoded*.

In general, a description may refer either explicitly or implicitly to elementary operations which help to clearly describe the overall behavior, like definitions help to clearly describe variables. As with variables, they are not strictly necessary for describing the relation between pre- and postconditions, but they serve as additional information for deriving test cases.

The result of step 1 for cgi_decode is summarized in Table 13.8.

STEP 2 Derive a first set of test case specifications from preconditions, postconditions and definitions The aim of this step is to explicitly describe the partition of the input domain:

Validated Preconditions: A simple precondition, i.e., a precondition that is expressed as a simple boolean expression without *and* or *or*, identifies two classes of input: values that satisfy the precondition and values that do not. We thus derive two test case specifications.

A compound precondition, given as a boolean expression with *and* or *or*, identifies several classes of inputs. Although in general one could derive a different test case specification for each possible combination of truth values of the elementary conditions, usually we derive only a subset of test case specifications using the modified condition decision coverage (*MC/DC*) approach, which is illustrated in Section 13.6 and in Chapter **??**. In short, we derive a set of combinations of elementary conditions such that each elementary condition can be shown to independently affect the outcome of each decision. For each elementary condition C, there are two test case specifications in which the truth values of all conditions except C are the same, and the compound condition as a whole evaluates to **True** for one of those test cases and **False** for the other.

Assumed Preconditions: We do not derive test case specifications for cases that violate assumed preconditions, since there is no defined behavior and thus no way to judge the success of such a test case. We also do not derive test cases when the whole input domain satisfies the condition, since test cases for these would be redundant. We generate test cases from assumed preconditions only when the MC/DC criterion generates more than one class of valid combinations (i.e., when the condition is a logical disjunction of more elementary conditions).

Postconditions: In all cases in which postconditions are given in a conditional form, the condition is treated like a validated precondition, i.e., we generate a test case specification for cases that satisfy and cases that do not satisfy the condition.

Definition: Definitions that refer to input or output variables are treated like postconditions, i.e., we generate a set of test cases for each definition given in conditional form with the same criteria used for validated preconditions. The test cases are generated for each variable that refers to the definition.

The elementary items of the specification identified in step 1 are scanned sequentially and a set of test cases is derived applying these rules. While scanning the specifications, we generate test case specifications incrementally. When new test case specifications introduce a finer partition than an existing case, or vice versa, the test case specification that creates the coarser partition becomes redundant and can be eliminated. For example, if an existing test case specification requires a non-empty set, and we have to add two test case specifications that require a size that is a power of two and one which is not, the existing test case specification can be deleted.

Scanning the elementary items of the cgi_decode specification given in Table 13.7, we proceed as follows:

PRE 1: The first precondition is a simple assumed precondition, thus, according to the rules, we do not generate any test case specification. The only condition would be encoded: a null terminated string of characters, but this matches every test case and thus it does not identify a useful partition.

PRE 2: The second precondition is a simple validated precondition, thus we generate two test case specifications, one that satisfies the condition and one that does not:

TC-PRE2-1 Encoded: a sequence of CGI items

TC-PRE2-2 Encoded: not a sequence of CGI items

postconditions: all postconditions in the cgi_decode specification are given in a conditional form with a simple condition. Thus, we generate two test case specifications for each of them. The generated test case specifications correspond to a case that satisfies the condition and a case that violates it.

POST 1:

TC-POST1-1 Encoded: contains one or more alphanumeric characters

TC-POST1-2 Encoded: does not contain any alphanumeric characters

POST 2:

TC-POST2-1 Encoded: contains one or more character '+'

Tc-POST2-2 Encoded: does not any contain character '+'

POST 3:

TC-POST3-1 Encoded: contains one or more CGI-hexadecimals

TC-POST3-2 Encoded: does not contain any CGI-hexadecimal

POST 4: we do not generate any new useful test case specifications, because the two specifications are already covered by the specifications generated from *POST 2*.

POST 5: we generate only the test case specification that satisfies the condition; the test case specification that violates the specification is redundant with respect to the test case specifications generated from *POST 3*

TC-POST5-1 : Encoded contains one or more malformed CGI-hexadecimals

POST 6: as for *POST 5*, we generate only the test case specification that satisfies the condition; the test case specification that violates the specification is redundant with respect to most of the test case specifications generated so far.

TC-POST6-1 Encoded: contains one or more illegal characters

definitions none of the definitions in the specification of cgi_decode is given in conditional terms, and thus no test case specifications are generated at this step.

The test case specifications generated from postconditions refine test case specification TC-PRE2-1, which can thus be eliminated from the checklist. The result of step 2 for cgi_decode is summarized in Table 13.9.

STEP 3 Complete the test case specifications using catalogs The aim of this step is to generate additional test case specifications from variables and operations used or defined in the computation. The catalog is scanned sequentially. For each entry of the catalog we examine the elementary components of the specification and we add test case specifications as required by the catalog. As when scanning the test case specifications during step 2, redundant test case specifications are eliminated.

Table 13.10 shows a simple catalog that we will use for the cgi_decoder example. A catalog is structured as a list of kinds of elements that can occur in a specification. Each catalog entry is associated with a list of generic test case specifications appropriate for that kind of element. We scan the specification for elements whose type is compatible with the catalog entry, then generate the test cases defined in the catalog for that entry. For example, the catalog of Table 13.10 contains an entry for boolean variables. When we find a boolean variable in the specification, we instantiate the catalog entry by generating two test case specifications, one that requires a **True** value and one that requires a **False** value.

Each generic test case in the catalog is labeled *in*, *out*, or *in/out*, meaning that a test case specification is appropriate if applied to either an input variable, or to an output variable, or in both cases. In general, erroneous values should be used when testing the behavior of the system with respect to input variables, but are usually impossible to produce when testing the behavior of the system with respect to output variables. For example, when the value of an input variable can be chosen from a set of values, it is important to test the behavior of the system for all enumerated values and some values outside the enumerated set, as required by entry *ENUMERATION* of the catalog.

PRE 2	*Validated*) the input string Encoded is a sequence of CGI items
[TC-PRE2-2]	Encoded: not a sequence of CGI items
POST 1	if the input string Encoded contains alphanumeric characters, they are copied to the output string in the corresponding positions
[TC-POST1-1]	Encoded: contains alphanumeric characters
[TC-POST1-2]	Encoded: does not contain alphanumeric characters
POST 2	if the input string Encoded contains '+' characters, they are replaced in the output string by ' ' in the corresponding positions
[TC-POST2-1]	Encoded: contains '+'
[TC-POST2-2]	Encoded: does not contain '+'
POST 3	if the input string Encoded contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.
[TC-POST3-1]	Encoded: contains CGI-hexadecimals
[TC-POST3-2]	Encoded: does not contain a CGI-hexadecimal
POST 4	if the input string Encoded is well-formed, cgi_decode returns 0
POST 5	if the input string Encoded contains a malformed CGI-hexadecimal, i.e., a substring "%xy", where either x or y are absent or non hexadecimal digits, cgi_decode returns 1
[TC-POST5-1]	Encoded: contains malformed CGI-hexadecimals
POST 6	if the input string Encoded contains any illegal character, cgi_decode returns a positive value
[TC-POST6-1]	Encoded: contains illegal characters
VAR 1	Encoded: a string of ASCII characters
VAR 2	Decoded: a string of ASCII characters
VAR 3	Return value: a boolean
DEF 1	hexadecimal digits are in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']
DEF 2	CGI-hexadecimals are sequences '%xy', where x and y are hexadecimal digits
DEF 3	CGI items are either alphanumeric characters, or '+', or CGI-hexadecimals
OP 1	Scan Encoded

Table 13.9: Test case specifications for cgi-decode generated after step 2

.

Boolean	
[in/out]	True
[in/out]	False
Enumeration	
[in/out]	Each enumerated value
[in]	Some value outside the enumerated set
Range $L \dots U$	
[in]	$L - 1$ (the element immediately preceding the lower bound)
[in/out]	L (the lower bound)
[in/out]	A value between L and U
[in/out]	U (the upper bound)
[in]	$U + 1$ (the element immediately following the upper bound)
Numeric Constant C	
[in/out]	C (the constant value)
[in]	$C - 1$ (the element immediately preceding the constant value)
[in]	$C + 1$ (the element immediately following the constant value)
[in]	Any other constant compatible with C
Non-Numeric Constant C	
[in/out]	C (the constant value)
[in]	Any other constant compatible with C
[in]	Some other compatible value
Sequence	
[in/out]	Empty
[in/out]	A single element
[in/out]	More than one element
[in/out]	Maximum length (if bounded) or very long
[in]	Longer than maximum length (if bounded)
[in]	Incorrectly terminated
Scan with action on elements P	
[in]	P occurs at beginning of sequence
[in]	P occurs in interior of sequence
[in]	P occurs at end of sequence
[in]	PP occurs contiguously
[in]	P does not occur in sequence
[in]	pP where p is a proper prefix of P
[in]	Proper prefix p occurs at end of sequence

Table 13.10: Part of a simple test catalog.

However, when the value of an output variable belongs to a finite set of values, we should derive a test case for each possible outcome, but we cannot derive a test case for an impossible outcome, so entry *ENUMERATION* of the catalog specifies that the choice of values outside the enumerated set is limited to input variables. Intermediate variables, if present, are treated like output variables.

Entry *Boolean* of the catalog applies to Return value (VAR 3). The catalog requires a test case that produces the value **True** and one that produces the value **False**. Both cases are already covered by test cases *TC-PRE2-1* and *TC-PRE2-2* generated for precondition *PRE 2*, so no test case specification is actually added.

Entry *Enumeration* of the catalog applies to any variable whose values are chosen from an explicitly enumerated set of values. In the example, the values of CGI item (DEF 3) and of improper CGI hexadecimals in POST 5 are defined by enumeration. Thus, we can derive new test case specifications by applying entry *enumeration* to POST 5 and to any variable that can contain CGI items.

The catalog requires creation of a test case specification for each enumerated value and for some excluded values. For encoded, which uses DEF 3, we generate a test case specification where a CGI-item is an alphanumeric character, one where it is the character '+', one where it is a CGI-hexadecimal, and some where it is an illegal value. We can easily ascertain that all the required cases are already covered by test case specifications for *TC-POST1-1*, *TC-POST1-2*, *TC-POST2-1*, *TC-POST2-2*, *TC-POST3-1*, and *TC-POST3-2*, so any additional test case specifications would be redundant.

From the enumeration of malformed CGI-hexadecimals in POST 5, we derive the following test cases: %y, %x, %ky, %xk, %xy (where x and y are hexadecimal digits and k is not). Note that the first two cases, %x (the second hexadecimal digit is missing) and %y (the first hexadecimal digit is missing) are identical, and %x is distinct from %xk only if %x are the last two characters in the string. A test case specification requiring a correct pair of hexadecimal digits (%xy) is a value out of the range of the enumerated set, as required by the catalog.

The added test case specifications are:

TC-POST5-2 encoded: terminated with %x, where x is a hexadecimal digit

TC-POST5-3 encoded: contains %ky, where k is not a hexadecimal digit and y is a hexadecimal digit.

TC-POST5-4 encoded: contains %xk, where x is a hexadecimal digit and k is not.

The test case specification corresponding to the correct pair of hexadecimal digits is redundant, having already been covered by TC-POST3-1. The test case TC-POST5-1 can now be eliminated because it is more general than the combination of TC-POST5-2, TC-POST5-3, and TC-POST5-4.

Entry *Range* applies to any variable whose values are chosen from a finite range. In the example, ranges appear three times in the definition of hexadecimal digit. Ranges also appear implicitly in the reference to alphanumeric characters (the alphabetic and numeric ranges from the ASCII character set) in DEF 3. For hexadecimal digits we will try the special values '/' and ':' (the characters that appear before '0' and after '9' in the ASCII encoding), the values '0' and '9' (upper and lower bounds of the first interval), some value between '0' and '9', and similarly '@', 'G', 'A', 'F', and some value between 'A' and 'F' for the second interval and ' ' ', 'g', 'a', 'f', and some value between 'a' and 'f' for the third interval.

These values will be instantiated for variable encoded, and result in 30 additional test case specifications (5 values for each subrange, giving 15 values for each hexadecimal digit and thus 30 for the two digits of CGI-hexadecimal). The full set of test case specifications is shown in Table **??**. These test case specifications are more specific than (and therefore replace) test case specifications TC-POST3-1, TC-POST5-3, and TC-POST5-4.

For alphanumeric characters we will similarly derive boundary, interior and excluded values, which result in 15 additional test case specifications, also given in Table **??**. These test cases are more specific than (and therefore replace) TC-POST1-1, TC-POST1-2, TC-POST6-1.

Entry *Numeric Constant* does not apply to any element of this specification.

Entry *Non-Numeric Constant* applies to '+' and '%', occurring in DEF 3 and DEF 2 respectively. Six test case specifications result, but all are redundant.

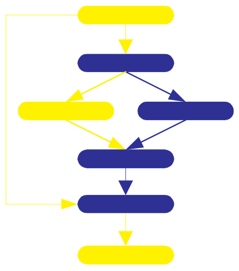
Entry *Sequence* applies to encoded (VAR 1), decoded (VAR 2), and cgi-item (DEF 2). Six test case specifications result for each, of which only five are mutually non-redundant and not already in the list. From VAR 1 (encoded) we generate test case specifications requiring an empty sequence, a sequence containing a single element, and a very long sequence. The catalog entry requiring more than one element generates a redundant test case specification, which is discarded. We cannot produce reasonable test cases for incorrectly terminated strings (the behavior would vary depending on the contents of memory outside the string), so we omit that test case specification.

All test case specifications that would be derived for decoded (VAR 2) would be redundant with respect to test case specifications derived for encoded (VAR 1).

From CGI-hexadecimal (DEF 2) we generate two additional test case specifications for variable encoded: a sequence that terminates with '%' (the only way to produce a one-character subsequence beginning with '%') and a sequence containing '%xyz', where x, y, and z are hexadecimal digits.

Entry *Scan* applies to Scan Encoded (OP 1) and generates 17 test case specifications. Three test case specifications (alphanumeric, '+', and CGI item) are generated for each of the first 5 items of the catalog entry. One test case specification is generated for each of the last two items of the catalog entry when *Scan* is applied to CGI item. The last two items of the catalog entry do not apply to alphanumeric characters and '+', since they have no non-trivial pre-fixes. Seven of the 17 are redundant. The ten generated test case specifications are summarized in Table 13.11.

Test catalogs, like other check-lists used in test and analysis (e.g., inspection check-lists), are an organizational asset that can be maintained and enhanced over time. A good test catalog will be written precisely and suitably annotated to resolve ambiguity (unlike the sample catalog used in this chapter). Catalogs should also be specialized to an organization and application domain, typically using a process such as defect causal analysis or root cause analysis. Entries are added to detect particular classes of faults that have been encountered frequently or have been particularly costly to remedy in previous projects. Refining check-lists is a typical activity carried out as part of process improvement. When a test reveals a program fault, it is useful to make a note of which catalog entries the test case originated from, as an aid to measuring the effectiveness of catalog entries. Catalog entries that are not effective should be removed.



13.9 Deriving Test Cases from Finite State Machines

Finite state machines are often used to specify sequences of interactions between a system and its environment. State machine specifications in one form or another are common for control and interactive systems, such as embedded systems, communication protocols, menu driven applications, threads of control in a system with multiple threads or processes.

In several application domains, specifications may be expressed directly as some form of finite-state machine. For example, embedded control systems are frequently specified with Statecharts, communication protocols are commonly described with SDL diagrams, and menu driven applications are sometimes modeled with simple diagrams representing states and transitions. In other domains, the finite state essence of the systems are left implicit in informal specifications. For instance, the informal specification of feature *Maintenance* of the Chipmuk web site given in Figure 13.9 describes a set of interactions between the maintenance system and its environment that can be modeled as transitions through a finite set of process states. The finite state nature of the interaction is made explicit by the finite state machine shown in Figure 13.10. Note that some transitions appear to be labeled by conditions rather than events, but they can be interpreted as shorthand for an event in which the condition becomes true or is discovered (e.g., “lack component” is shorthand for “discover that a required component is not in stock.”

Many control or interactive systems are characterized by an infinite set of states. Fortunately, the non-finite-state parts of the specification are often simple enough that finite state machines remain a useful model for testing as well as specification. For example, communication protocols are frequently specified using finite state machines, often with some extensions that make

TC-POST2-1	Encoded contains character ’+’
TC-POST2-2	Encoded does not contain character ’+’
TC-POST3-2	Encoded does not contain a CGI-hexadecimal
TC-POST5-2	Encoded terminates with %x
TC-VAR1-1	Encoded is the empty sequence
TC-VAR1-2	Encoded is a sequence containing a single character
TC-VAR1-3	Encoded is a very long sequence
TC-DEF2-1	Encoded contains ’%/y’
TC-DEF2-2	Encoded contains ’%0y’
TC-DEF2-3	Encoded contains ’%xy’, with x in [1..8]
TC-DEF2-4	Encoded contains ’%9y’
TC-DEF2-5	Encoded contains ’%:y’
TC-DEF2-6	Encoded contains ’%@y’
TC-DEF2-7	Encoded contains ’%Ay’
TC-DEF2-8	Encoded contains ’%xy’, with x in [B..E]
TC-DEF2-9	Encoded contains ’%Fy’
TC-DEF2-10	Encoded contains ’%Gy’
TC-DEF2-11	Encoded contains ’%‘y’
TC-DEF2-12	Encoded contains ’%ay’
TC-DEF2-13	Encoded contains ’%xy’, with x in [b..e]
TC-DEF2-14	Encoded contains ’%fy’
TC-DEF2-15	Encoded contains ’%gy’
TC-DEF2-16	Encoded contains ’%x/’
TC-DEF2-17	Encoded contains ’%x0’
TC-DEF2-18	Encoded contains ’%xy’, with y in [1..8]
TC-DEF2-19	Encoded contains ’%x9’
TC-DEF2-20	Encoded contains ’%x:’
TC-DEF2-21	Encoded contains ’%x@’
TC-DEF2-22	Encoded contains ’%xA’
TC-DEF2-23	Encoded contains ’%xy’, with y in [B..E]
TC-DEF2-24	Encoded contains ’%xF’
TC-DEF2-25	Encoded contains ’%xG’
TC-DEF2-26	Encoded contains ’%x‘’
TC-DEF2-27	Encoded contains ’%xa’
TC-DEF2-28	Encoded contains ’%xy’, with y in [b..e]
TC-DEF2-29	Encoded contains ’%xf’
TC-DEF2-30	Encoded contains ’%xg’
TC-DEF2-31	Encoded contains ’%$’
TC-DEF2-32	Encoded contains ’%xyz’
TC-DEF3-1	Encoded contains ’/’
TC-DEF3-2	Encoded contains ’0’
TC-DEF3-3	Encoded contains c, with c in [’1’..’8’]
TC-DEF3-4	Encoded contains ’9’
TC-DEF3-5	Encoded contains ’:’
TC-DEF3-6	Encoded contains ’@’
TC-DEF3-7	Encoded contains ’A’
TC-DEF3-8	Encoded contains c, with c in [’B’..’Y’]
TC-DEF3-9	Encoded contains ’Z’
TC-DEF3-10	Encoded contains ’[’
TC-DEF3-11	Encoded contains ’‘’
TC-DEF3-12	Encoded contains ’a’
TC-DEF3-13	Encoded contains c, with c in [’b’..’y’]
TC-DEF3-14	Encoded contains ’z’
TC-DEF3-15	Encoded contains ’{’
TC-OP1-1	Encoded contains ’^a’
TC-OP1-2	Encoded contains ’^+’
TC-OP1-3	Encoded contains ^%xy’
TC-OP1-4	Encoded contains ’$a$$’
TC-OP1-5	Encoded contains ’+$’
TC-OP1-6	Encoded contains ’%xy$’
TC-OP1-7	Encoded contains ’aa’
TC-OP1-8	Encoded contains ’++’
TC-OP1-9	Encoded contains ’%xy%zw’
TC-OP1-10	Encoded contains ’%x%yz’

where w, x, y, z are hexadecimal digits, a is an alphanumeric character, ^ represents the beginning of the string, and $ represents the end of the string.

Table 13.11: Summary table: Test case specifications for cgi-decode generated with a catalog.

Maintenance: The *Maintenance* function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or web site and the customer is a US or UE resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The mainenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate. If the customer does not accept the estimate, the product is returned to the customer.

Small problems can be repaired directly at the mainenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.

Figure 13.9: The functional specification of feature *Maintenace* of the Chipmuk web site.

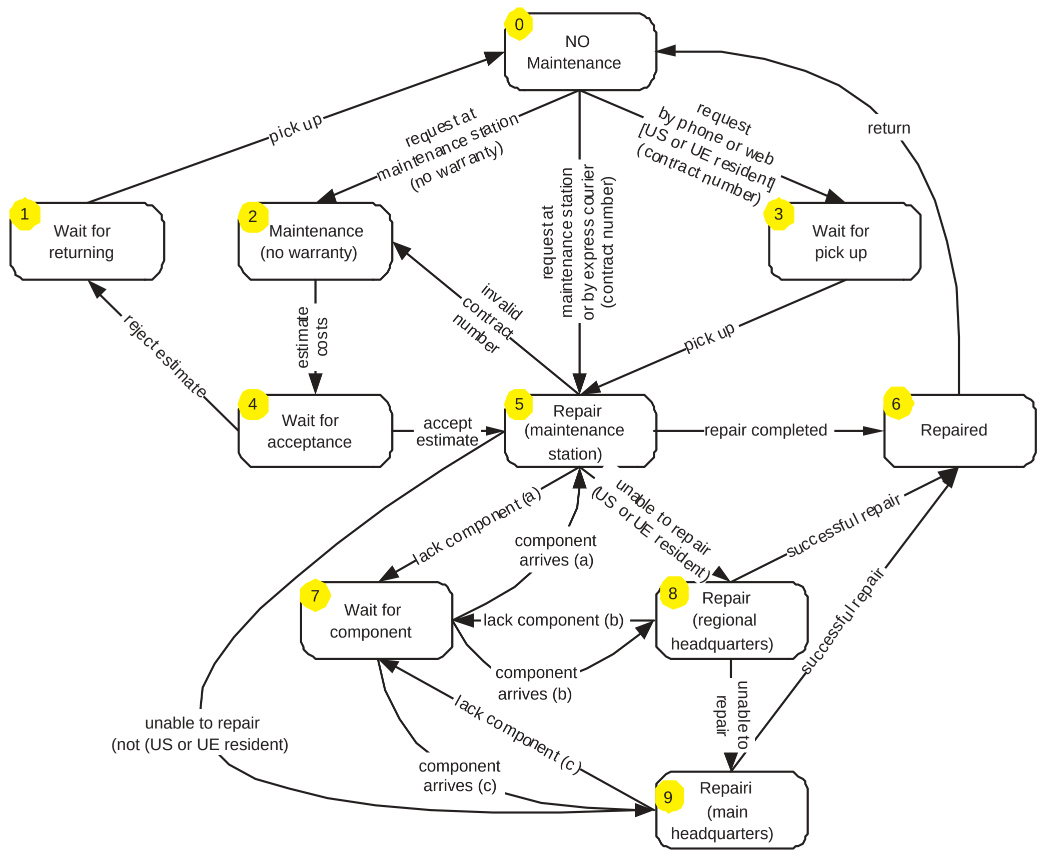


Figure 13.10: The finite state machine corresponding to functionality *Maintenance* specified in Figure 13.9

T-Cover

TC-1	0 – 2 – 4 – 1 – 0
TC-2	0 – 5 – 2 – 4 – 5 – 6 – 0
TC-3	0 – 3 – 5 – 9 – 6 – 0
TC-4	0 – 3 – 5 – 7 – 5 – 8 – 7 – 8 – 9 – 7 – 9 – 6 – 0

Table 13.12: A set of test specifications in the form of paths in a finite-state machine specification. States are indicated referring to the numbers given in Figure 13.10. For example, TC-1 is a test specification requiring transitions (0,2), (2,4), (4,1), and (1,0) be traversed, in that order.

them not truly finite-state. A state machine that simply receives a message on one port and then sends the same message on another port is not really finite-state unless the set of possible messages is finite, but is often rendered as a finite state machine, ignoring the contents of the exchanged messages.

State-machine specifications can be used both to guide test selection and in construction of an oracle that judges whether each observed behavior is correct. There are many approaches for generating test cases from finite state machines, but most are variations on a basic strategy of checking each state transition. One way to understand this basic strategy is to consider that each transition is essentially a specification of a precondition and postcondition, e.g., a transition from state S to state T on stimulus i means "*if* the system is in state S and receives stimulus i, *then* after reacting it will be in state T." For instance, the transition labeled *accept estimate* from state *Wait for acceptance* to state *Repair (maintenance station)* of Figure 13.10 indicates that if an item is on hold waiting for the customer to accept an estimate of repair costs, and the customer accepts the estimate, then the maintenance station begins repairing the item.

A faulty system could violate any of these precondition, postcondition pairs, so each should be tested. For instance, the state *Repair (maintenance station)* can be arrived through three different transitions, and each should be checked.

Details of the approach taken depend on several factors, including whether system states are directly observable or must be inferred from stimulus/response sequences, whether the state machine specification is complete as given or includes additional, implicit transitions, and whether the size of the (possibly augmented) state machine is modest or very large.

Δ Transition Coverage

A basic criterion for generating test cases from finite state machines is *transition coverage*, which requires each transition to be traversed at least once. Test case specifications for transition coverage are often given as sets of state sequences or transition sequences. For example, *T-Cover* in Table 13.12 is a set of four paths, each beginning at the initial state, which together cover all transitions of the finite state machine of Figure 13.10. *T-Cover* thus satisfies the transition coverage criterion.

The transition coverage criterion depends on the assumption that the finite-state machine model is a sufficient representation of all the "important" state, e.g., that transitions out of a state do not depend on how one reached that state. Although it can be considered a logical flaw, in practice one often finds state machines that exhibit "history sensitivity," i.e., the transitions from a state depend on the path by which one reached that state. For example, in Figure 13.10, the transition taken from state *Wait for component* when the component becomes available depends on how the state was entered. This is a flaw in the model — there really should be three distinct *Wait for component* states, each with a well-defined action when the component becomes available. However, sometimes it is more expedient to work with a flawed state-machine model than to repair it, and in that case test suites may be based on more than the simple transition coverage criterion.

Coverage criteria designed to cope with history sensitivity include *single state path coverage*, *single transition path coverage*, and *boundary interior loop coverage*.The *single state path coverage* criterion requires each path that traverses states at most once to be exercised. The *single transition path coverage* criterion requires each path that traverses transitions at most once to be exercised. The *boundary interior loop coverage* criterion requires each distinct loop of the state machine to be exercised the minimum, an intermediate, and the maximum number of times[11]. These criteria may be practical for very small and simple finite-state machine specifications, but since the number of even simple paths (without repeating states) can grow exponentially with the number of states, they are often impractical.

Δ Single State Path Coverage

Δ Single Transition Path Coverage

Δ Boundary Interior Loop Coverage

Specifications given as finite-state machines are typically incomplete, i.e., they do not include a transition for every possible (state, stimulus) pair. Often the missing transitions are implicitly error cases. Depending on the system, the appropriate interpretation may be that these are *don't care* transitions (since no transition is specified, the system may do anything or nothing), *self* transitions (since no transition is specified, the system should remain in the same state), or (most commonly) *error* transitions that enter a distinguished state and possibly trigger some error handling procedure. In at least the latter two cases, thorough testing includes the implicit as well as the explicit state transitions. No special techniques are required; the implicit transitions are simply added to the representation before test cases are selected.

The presence of implicit transitions with a *don't care* interpretation is typically an implicit or explicit statement that those transitions are impossible, e.g., because of physical constraints. For example, in the specification of the maintenance procedure of Figure 13.10, the effect of event *lack of component* is specified only for the states that represent repairs in progress. Sometimes it is possible to test such sequences anyway, because the system does not prevent such events from occurring Where possible, it may be best to treat *don't care* transitions as *self* transitions (allowing the possibility of imperfect translation from physical to logical events, or of future physical layers

[11]The boundary interior path coverage was originally proposed for structural coverage of program control flow, and is described in Chapter 14

Advanced search: The *Advanced search* function allows for searching elements in the website database.

The key for searching can be:

a simple string , i.e., a simple sequence of characters,

a compound string , i.e.,

- a string terminated with character *, used as wild character, or
- a string composed of substrings included in braces and separated with commas, used to indicate alternatives.

a combination of strings , i.e., a set of strings combined with the boolean operators NOT, AND, OR, and grouped within parenthesis to change the priority of operators.

Examples:

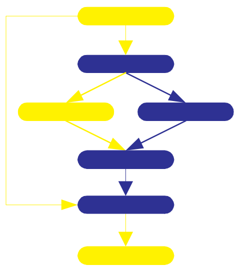
laptop The routine searches for string "laptop"

{DVD*,CD*} The routine searches for strings that start with substring "DVD" or "CD" followed by any number of characters

NOT (C2021*) AND C20* The routine searches for strings that start with substring "C20" followed by any number of characters, except substring "21"

Figure 13.11: The functional specification of feature *advanced search* of the Chipmunk web site.

with different properties), or as *error* transitions (requiring that unanticipated sequences be recognized and handled). If it is not possible to produce test cases for the *don't care* transitions, then it may be appropriate to pass them to other validation or verification activities, for example, by including explicit assumptions in a requirements or specification document that will undergo inspection.



13.10 Deriving Test Cases from Grammars

Sometimes, functional specifications are given in the form of grammars or regular expressions. This is often the case in description of languages, such as specifications of compilers or interpreters. More often syntactic structures are described with natural or domain specific languages, such as simple scripting rules and complex document structures.

The informal specification of the advanced search functionality of the Chipmuk website shown in Figure 13.11 defines the syntax of the search pattern. Not surprisingly, this specification can easily be expressed as a grammar. Figure 13.12 expresses the specification as a grammar in Bachus Naur Form (BNF).

$$\begin{aligned}
\langle \textit{search}\rangle &::= \langle \textit{search}\rangle\ \langle \textit{binop}\rangle\ \langle \textit{term}\rangle \mid \boxed{\texttt{not}}\ \langle \textit{search}\rangle \mid \langle \textit{term}\rangle \\
\langle \textit{binop}\rangle &::= \boxed{\texttt{and}} \mid \boxed{\texttt{or}} \\
\langle \textit{term}\rangle &::= \langle \textit{regexp}\rangle \mid \boxed{\texttt{(}}\ \langle \textit{search}\rangle\ \boxed{\texttt{)}} \\
\langle \textit{regexp}\rangle &::= \textit{Char}\ \langle \textit{regexp}\rangle \mid \textit{Char} \mid \boxed{\texttt{\{}}\ \langle \textit{choices}\rangle\ \boxed{\texttt{\}}} \mid \boxed{\texttt{*}} \\
\langle \textit{choices}\rangle &::= \langle \textit{regexp}\rangle \mid \langle \textit{regexp}\rangle\ \boxed{\texttt{,}}\ \langle \textit{choices}\rangle
\end{aligned}$$

Figure 13.12: The BNF description of functionality *Advanced search*.

A second example is given in Figure 13.13, which specifies a *product configuration* of the Chipmuk website. In this case, the syntactic structure of *product configuration* is described by an XML schema, which defines an element *Model* of type *ProductConfigurationType*. XML schemata are essentially a variant of BNF, so it is not difficult to render the schema in the same BNF notation, as shown in Figure 13.13.

In general, grammars are well suited to represent inputs of varying and unbounded size, boundary conditions, and recursive structures. None of which can be easily captured with fixed lists of parameters, as required by most methods presented in this chapter.

∆ Minimal grammar-based criterion

Generating test cases from grammar specifications is straightforward and can easily be automated. To produce a string, we start from a non-terminal symbol and we progressively substitute non-terminals occurring in the current string with substrings, as indicated by the applied productions, until we obtain a string composed only of terminal symbols. In general at each step, several rules can be applied. A minimal set of test cases can be generated by requiring each production to be exercised at least once. Test cases can be generated by starting from the start symbol and applying all productions. The number and complexity of the generated test cases depend on the order of application of the productions. If we first apply productions with non-terminals on the right hand side, we generate a smaller set of test cases, each one tending to be a large test case. On the contrary, first applying productions with only terminals on the right hand side, we generate larger sets of smaller test cases. An algorithm that favors non-terminals applied to the BNF for Advanced Search of Figure 13.11, generates the test case

not Char {**, Char*} *and (Char or Char)*

that exercise all productions. The derivation tree for this test case is given in Figure 13.15. It shows that all productions of the BNF are exercised at least once.

∆ Boundary condition grammar-based criteria

The minimal set of test cases can be enriched by considering boundary conditions. Boundary conditions apply to recursive productions. To generate test cases for boundary conditions we need to identify the minimum and maximum number of recursive applications of a production and then generate a test case for the minimum, maximum, one greater than minimum and

```
<xsd:schema xmlns="http://www.w3.org/2000/08/XMLSchema">

<xsd:annotation>
  <xsd:documentation>
     Chipmunk Computers - Product Configuration Schema
     Copyright 2001 Mauro Pezze and Michal Young
  </xsd:documentaton>
</xsd:annotation>

<xsd:element name="Model" type="ProductConfigurationType"/>

<xsd:complexType name="ProductConfigurationType">
  <xsd:attribute name="modelNumber" type="xsd:string" use="required"/>
  <xsd:sequence>
     <xsd:element name="Component" minoccurs="0" maxoccurs="unbounded">
         <xsd:element name="ComponentType" type="string"/>
         <xsd:element name="ComponentValue" type="string"/>
     </xsd:element>
  </xsd:sequence>
  <xsd:element name="OptionalComponent" minoccurs="0" maxoccurs="unbounded">
     <xsd:union memberTypes="ComponentKind"

     <xsd:element name="ComponentType" type="string"/>
  </xsd:element>
</xsd:complexType>
```

XML is a parenthetical language: descriptions of items are either enclosed in angular parenthesis (⟨⟩) or terminated with "/item" clauses. Schema and annotation (⟨xsd:schema ...⟩ and ⟨xsd:annotation⟩ ...⟨/xsd:annotation⟩) give information about the XML version and the authors. The first clause (⟨xsd:element ...⟩ describes a Model as an instance of type ProductConfigurationType. The clause ⟨xsd:complexType⟩ ... ⟨/xsd:complexType⟩ describes type ProductConfigurationType as composed of

- a field modelNumber of type String. Field modelNumber is required.
- a possibly empty set of Components, each characterized by fields ComponentType and ComponentValue, both of type string.
- a possibly empty set of OptionalComponents, each characterized by a ComponentType of type string

Figure 13.13: The XML Schema that describes a *Product configuration* of the Chipmuk website

$\langle Model\rangle$	::=	$\langle modelNumber\rangle$ $\langle compSequence\rangle$ $\langle optCompSequence\rangle$
$\langle compSequence\rangle$	::=	$\langle Component\rangle$ $\langle compSequence\rangle$ | $\boxed{\texttt{empty}}$
$\langle optCompSequence\rangle$	::=	$\langle OptionalComponent\rangle$ $\langle optCompSequence\rangle$ | $\boxed{\texttt{empty}}$
$\langle Component\rangle$	::=	$\langle ComponentType\rangle$ $\langle ComponentValue\rangle$
$\langle OptionalComponent\rangle$	::=	$\langle ComponentType\rangle$
$\langle modelNumber\rangle$	::=	$\boxed{\texttt{string}}$
$\langle ComponentType\rangle$	::=	$\boxed{\texttt{string}}$
$\langle ComponentValue\rangle$	::=	$\boxed{\texttt{string}}$

Figure 13.14: The BNF description of *Product Configuration*.

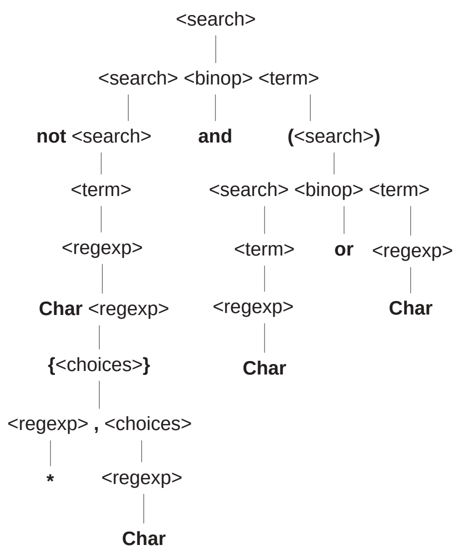


Figure 13.15: The derivation tree of a test case for functionality *Advanced Search* derived from the BNF specification of Figure 13.12.

Model ⟨*Model*⟩	::= ⟨*modelNumber*⟩ ⟨*compSequence*⟩ ⟨*optCompSequenc*
compSeq1 limit=16 ⟨*compSequence*⟩	::= ⟨*Component*⟩ ⟨*compSequence*⟩
compSeq2 ⟨*compSequence*⟩	::= $\boxed{\texttt{empty}}$
optCompSeq1 limit=16 ⟨*optCompSequence*⟩	::= ⟨*OptionalComponent*⟩ ⟨*optCompSequence*⟩
optCompSeq2 ⟨*optCompSequence*⟩	::= $\boxed{\texttt{empty}}$
Comp ⟨*Component*⟩	::= ⟨*ComponentType*⟩ ⟨*ComponentValue*⟩
OptComp ⟨*OptionalComponent*⟩	::= ⟨*ComponentType*⟩
modNum ⟨*modelNumber*⟩	::= $\boxed{\texttt{string}}$
CompTyp ⟨*ComponentType*⟩	::= $\boxed{\texttt{string}}$
CompVal ⟨*ComponentValue*⟩	::= $\boxed{\texttt{string}}$

Figure 13.16: The BNF description of *Product Configuration* extended with production names and limits.

one smaller than maximum number of application of each production.

To apply boundary condition grammar based criteria, we need to add limits to the recursive productions. Names and limits are shown in Figure 13.16, which extends the grammar of Figure 13.14. Compound productions are decomposed into their elementary components. Production names are used for references purpose. Limits are added only to recursive productions. In the example of Figure 13.16, the limit of both productions *compSeq1* and *optCompSeq1* is set to 16, i.e., we assume that each model can have at most 16 required and 16 optional components.

The boundary condition grammar based criteria would extend the minimal set by adding test cases that cover the following choices:

- zero required components (*compSeq1* applied 0 times)
- one required component (*compSeq1* applied 1 time)
- fifteen required components (*compSeq1* applied $n - 1$ times)
- sixteen required components (*compSeq1* applied n times)
- zero optional components (*optCompSeq1* applied 0 times)
- one optional component (*optCompSeq1* applied 1 time)
- fifteen optional components (*optCompSeq1* applied $n - 1$ times)
- sixteen optional components (*optCompSeq1* applied n times)

weight	Model	1
weight	compSeq1	10
weight	compSeq2	0
weight	optCompSeq1	10
weight	optCompSeq2	0
weight	Comp	1
weight	OptComp	1
weight	modNum	1
weight	CompTyp	1
weight	CompVal	1

Figure 13.17: A sample seed that assigns probabilities to productions of the BNF specification of the BNF of *Product Configuration*.

Additional boundary condition grammar based criteria can be defined by also requiring specific combinations of applications of productions, e.g., requiring all productions to be simultaneously applied the minimum or the maximum number of times. This additional requirement applied to the example of Figure 13.16 would require additional test cases corresponding to the cases of (1) both no required and no optional components (both *compSeq1* and *optCompSeq1* applied 0 times), and (2) 16 required and 16 additional components (both *compSeq1* and *optCompSeq1* applied n times).

Δ Probabilistic grammar-based criteria

Probabilistic grammar based criteria assign probabilities to productions, thus indicating which production to select at each step to generate test cases. Unlike names and limits, probabilities are attached to grammar productions as a separate set of annotations, called seed. In this way, we can generate several sets of test cases from the same grammar with different seeds. Figure 13.17 shows a sample seed for the grammar that specify the product configuration functionality of the Chipmuk web site presented in Figure 13.16.

Probabilities are indicated as weights that determine the relative occurrences of the production in a sequence of applications that generate a test case. The same weight for *compSeq1* and *optCompSeq1* indicates that test cases are generated by balancing the applications of these two productions, i.e., they contain the same number of required and optional components. Weight 0 disables the productions, which are then applied only when the application of competing productions reaches the limit indicated in the grammar.

13.11 Choosing a Suitable Approach

We have seen several approaches to functional testing, each applying to different kinds of specifications. Given a specification, there may be one or more techniques well suited for deriving functional test cases, while some other techniques may be hard or even impossible to apply, or may lead to unsatisfactory results. Some techniques can be interchanged, i.e., they can be applied to the same specification and lead to similar results. Other techuiques are complementary, i.e., they apply to different aspects of the same specification or at different stages of test case generation. In some cases, approaches apply directly to the form in which the specification is given, in some other cases, the specification must be transformed into a suitable form.

The choice of approach for deriving functional testing depends on several factors: the nature of the specification, the form of the specification, expertieses and experience of test designers, the structure of the organization, the availability of tools, the budget and quality constraints, and the costs of designing and implementing the scaffolding.

Nature and form of the specification Different approaches exploit different characteristics of the specification. For example, the presence of several constraints on the input domain may suggest the category partition method, while lack of constraints may indicate a combinatorial approach. The presence of a finite set of states could suggest a finite state machine approach, while inputs of varying and unbounded size may be tackled with grammar based approaches. Specifications given in a specific format, e.g., as finite state machines, or decision structures suggest the corresponding techniques. For example, functional test cases for SDL specifications of protocols are often derived with finite state machine based criteria.

Experience of test designers and organization Experience of testers and company procedures may drive the choice of the testing technique. For example, test designers expert in category partition may prefer this technique over a catalog based approach when both are applicable, while a company that works in a specific domain may require the use of catalogs suitably produced for the domain of interest.

Tools Some techniques may require the use of tools, whose availability and cost should be taken into account when choosing a specific testing technique. For example, several tools are available for deriving test cases from SDL specifications. The availability of one of these tools may suggest the use of SDL for capturing a subset of the requirements expressed in the specification.

Budget and quality constraints Different quality and budget constraints may lead to different choices. For example, the need of quickly check a software product without stringent reliability requirements may lead to chose a random test generation approach, while a thorough check of a safety critical application may require the use of sophisticated methods for functional test case generation. When choosing a specific approach, it is important to evaluate all cost related aspects. For example, the generation of a large number of random tests may require the design of sophisticated oracles, which may raise the costs of testing over an acceptable threshold; the cost of a specific tool and the related training may go beyond the advantages of adopting a specific approach, even if the nature and the form of the specification may suggest the suitability of that approach.

Many engineering activities require careful trading off different aspects. Functional testing is not an exception: successfully balancing the many aspects is a difficult and often underestimated problem that requires highly skilled designers. Functional testing is not an exercise of choosing the optimal approach, but a complex set of activities for finding a suitable combination of models and techniques that can lead to a set of test cases that satisfy cost and quality constraints. This balancing extends beyond test design to software design for test. Appropriate design not only improves the software development process, but can greatly facilitate the job of test designers, and thus lead to substantial savings.

Too often test designers make the same mistake as non-expert programmers, that is to start generating code in one case, test cases in the other, without prior analysis of the problem domain. Expert test designers carefully examine the available specifications, their form, domain and company constraints for identifying a suitable framework for designing test case specifications before even starting to consider the problem of test case generation.

Open research issues

Functional testing is by far the most popular way of deriving test cases in industry, but both industrial practice and research are still far from general and satisfactory methodologies. Key reasons for the relative shortage of results are the intrinsic difficulty of the problem and the difficulty of working with informal specifications. Research in functional testing is increasingly active and progresses in many directions.

A hot research area concerns the use of formal methods for deriving test cases. In the past three decades, formal methods have been mainly studied as a means for formally proving software properties. Recently, a lot of attention has been moved towards the use of formal methods for deriving test cases. There are three main open research topics in this area:

- definition of techniques for automatically deriving test cases from particular formal methods. Formal methods present new challenges and opportunities for deriving test cases. We can both adapt existing techniques borrowed from other disciplines or research areas and define new techniques for test case generation. The formal nature can support fully automatic generation of test cases, thus opening additional problems and research challenges.

- adaptation of formal methods to be more suitable for test case generation. As illustrated in this chapter, test cases can be derived in two broad ways, either by identifying representative values or by deriving a model of the unit under test. The possibility of automatically generating test cases from different formal methods offers the opportunities of a large set of models to be used in testing. The research challenge relies in the capability of identifying a tradeoff between costs of generating formal models and savings in automatically generating test cases. The possibility of deriving simple formal models capturing only the aspects of interests for testing has been already studied in some specific areas, like concurrency, where test cases can be derived from models of the concurrency structure ignoring other details of the system under test, but the topic presents many new challenges if applied to wider classes of systems and models.

- identification of a general framework for deriving test cases from any particular formal specification. Currently research is moving towards the study of techniques for generating test cases for specific formal methods. The unification of methods into a general framework will constitute an additional important result that will allow the interchange of formal methods and testing techniques.

Another hot research area is fed by the increasing interest in different specification and design paradigms. New software development paradigms, such as the object oriented paradigm, as well as techniques for addressing increasingly important topics, such as software architectures and design patterns, are often based on new notations. Semi-formal and diagrammatic notations offer several opportunities for systematically generating test cases. Resarch is active in investigating different possibilities of (semi) automatically deriving test cases from these new forms of specifications and studying the effectiveness of existing test case generation techniques[12].

Most functional testing techniques do not satisfactory address the problem of testing increasingly large artifacts. Existing functional testing techniques do not take advantages of test cases available for parts of the artifact under test. Compositional approaches for deriving test cases for a given system taking advantage of test cases available for its subsystems is an important open research problem.

Further Reading

Functional testing techniques, sometimes called “black-box testing” or “specification-based testing,” are presented and discussed by several authors. Ntafos [DN81] makes the case for random, rather than systematic testing; Frankl, Hamlet,

[12]Problems and state-of-art techniques for testing object oriented software and software architectures are discussed in Chapters **??** and **??**

Littlewood and Strigini [FHLS98] is a good starting point to the more recent literature considering the relative merits of systematic and statistical approaches.

Category partition testing is described by Ostrand and Balcer [OB88]. The combinatorial approach described in this chapter is due to Cohen, Dalal, Fredman, and Patton [CDFP97]; the algorithm described by Cohen et al. is patented by Bellcore. Myers' classic text [Mye79] describes a number of techniques for testing decision structures. Richardson, O'Malley, and Tittle [ROT89] and Stocks and Carrington [SC96] are among more recent attempts to generate test cases based on the structure of (formal) specifications. Beizer's *Black Box Testing* [Bei95] is a popular presentation of techniques for testing based on control and data flow structure of (informal) specifications.

Catalog-based testing of subsystems is described in depth by Marick's *The Craft of Software Testing* [Mar97].

Test design based on finite state machines has been important in the domain of communication protocol development and conformance testing; Fujiwara, von Bochmann, Amalou, and Ghedamsi [FvBK$^+$91] is a good introduction. Gargantini and Heitmeyer [GH99] describe a related approach applicable to software systems in which the finite-state machine is not explicit but can be derived from a requirements specification.

Test generation from context-free grammars is described by Celentano et al. [CCD$^+$80] and apparently goes back at least to Hanford's test generator for an IBM PL/I compiler [Han70]. The probabilistic approach to grammar-based testing is described by Sirer and Bershad [SB99], who use annotated grammars to systematically generate tests for Java virtual machine implementations.

Related topics

Readers interested in the complementarites between functional and structural testing as well as readers interested in the testing decision structures and control and data flow graphs may continue with the next Chapters that describe structural and data flow testing. Readers interested in finite state machine based testing may go to Chapters 17 and **??** that discuss testing of object oriented and distributed system, respectively. Readers interested in the quality of specifications may goto Chapters 25 and **??**, that describe inspection techniques and methods for testing and analysis of specifications, respectively. Readers interested in other aspect of functional testing may move to Chapters 16 and **??**, that discuss technuqes for testing complex data structures and GUIs, respectively.

Exercises

Ex13.1. In the “Extreme Programming” (XP) methodology [?], a written description of a desired feature may be a single sentence, and the first step to designing the implementation of that feature is designing and implementing a set of test cases. Does this aspect of the XP methodology contradict our assertion that test cases are a formalization of specifications?

Ex13.2. Compute the probability of selecting a test case that reveals the fault inserted in line 25 of program Root *of Figure 13.1 by randomly sampling the input domain, assuming that type double has range* $-2^{31} \dots 2^{31} - 1$. *Compute the probability of selecting a test case that reveals a fault, asuming that both lines 18 and 25 of program* Root *contains the same fault, i.e., missing condition* $a \neq 0$. *Compare the two probabilities.*

Ex13.3. Identify independently testable units in the following specification.

Desk calculator Desk calculator *performs the following algebraic operations:* sum, subtraction, product, division, *and* percentage *on* integers *and* real numbers. *Operands must be of the same type, except for percentage, which allows the first operator to be either integer or real, but requires the second to be an integer that indicates the percentage to be computed. Operations on integers produce integer results. Program* Calculator *can be used with a textual interface that provides the following commands:*

Mx=N *where Mx is a memory location,i.e., M0,.. M9 and N is a number. Integers are given as non-empty sequences of digits, with or without sign. Real numbers are given as non-empty sequences of digits that include a dot “.”, with or without sign. Real numbers can be terminated with an optional exponent, i.e., character “E” followed by an integer. The command displays the stored number.*

Mx=display *, where Mx is a memory location and* display *indicates the value shown on the last line.*

operand1 operation operand2 *, where* operand1 *and* operand2 *are numbers or memory locations or* display *and* operation *is one of the following symbols: “+”, “-”, “*”, “/”, “%”, where each symbol indicates a particular operation. Operands must follow the type conventions. The command displays the result or the string* Error.

or with a graphical interface that provides a display with 12 characters and the following keys:

$\boxed{0}$, $\boxed{1}$, $\boxed{2}$, $\boxed{3}$, $\boxed{4}$, $\boxed{5}$, $\boxed{6}$, $\boxed{7}$, $\boxed{8}$, $\boxed{9}$ *, the 10 digits*

$\boxed{+}$, $\boxed{-}$, $\boxed{*}$, $\boxed{/}$, $\boxed{\%}$ *, the operations*

$\boxed{=}$ *to display the result of a sequence of operations*

$\boxed{\texttt{C}}$ *, to clear display*

$\boxed{\texttt{M}}$, $\boxed{\texttt{M+}}$, $\boxed{\texttt{MS}}$, $\boxed{\texttt{MR}}$, $\boxed{\texttt{MC}}$ *, where* $\boxed{\texttt{M}}$ *is pressed before a digit to indicate the target memory, 0...9, keys* $\boxed{\texttt{M+}}$, $\boxed{\texttt{MS}}$, $\boxed{\texttt{MR}}$, $\boxed{\texttt{MC}}$ *pressed after* $\boxed{\texttt{M}}$ *and a digit indicate the operation to be performed on the target memory: add display to memory, store display in memory, restore memory, i.e., move the value in memory to the display and clear memory.* Example: $\boxed{\texttt{5}}\boxed{\texttt{+}}\boxed{\texttt{1}}\boxed{\texttt{0}}\boxed{\texttt{M}}\boxed{\texttt{3}}\boxed{\texttt{MS}}\boxed{\texttt{8}}\boxed{\texttt{0}}\boxed{\texttt{-}}\boxed{\texttt{M}}\boxed{\texttt{3}}\boxed{\texttt{MR}}\boxed{\texttt{=}}$ *prints 65 (the value 15 is stored in memory cell 3 and then retrieved to compute 80 - 15).*

Ex13.4. Assume we have a set of parameter caracteristics (categories) and value classes (choices) obtained by applying the category partition method to an informal specification. Write an algorithm for computing the number of combinations of value classes for each of the following restricted cases:

- *(Case 1) Parameter characteristics and value classes are given without constraints*
- *(Case 2) Only constraints* error *and* single *are used (without constraints* property *and* if-property*)*
- *(Case 3) Constraints are used, but constraints* property *and* if-property *are not used for value classes of the same paramter characteristics, i.e., only one of these two types of contrain can be used for value classes of the same parameter characteristic. Moreover, constraints are not nested, i.e., if a value class of a given parameter characteristic A is constrained with* if-property *with repect to a set of different parameter characteristics B, then B cannot be further constrained with* if-property.

Ex13.5. Given a set of parameter characteristics (categories) and value classes (choices) obtained by applying the category partition method to an informal specification, explain either with a deduction or with examples why unrestricted use of constraints property *and* if-property *makes it difficult to compute the number of derivable combinations of value classes.*
Write heuristics to compute a reasonable upper bound for the number of derivable combinations of value classes when constraints can be used without limits.

Ex13.6. Consider the following specification, which extends the specification of the feature Check-configuration *of the Chipmuk web site given in Figure 13.3. Derive a test case specification using the category partition method and compare the test specification you obtain with the specification of Table 13.1. Try to identify a procedure for deriving the test specifications of the new version of the functional specification from the former version. Discuss the suitability of category-partition test design for incremental development with evolving specifications.*

Check-Configuration: *the* Check-configuration *function checks the validity of a computer configuration. The parameters of check-configuration are:*

Product line: *A product line identifies a set of products sharing several components and accessories. Different product lines have distinct components and accessories.*
Example: *Product lines include desktops, servers, notebooks, digital cameras, printers.*

Model: *A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customer's needs.*
Example: *The required "slots" of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.*

Set of Components: *A set of* $\langle slot, component \rangle$ *pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value "empty" is allowed (and may be the default selection) for optional slots.*
In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.
Example: *The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, "empty" is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.*

Set of Accessories: *An accessory is a choice that can be varied within a model, and which is designed to be replaced by the end user. Available choices are determined by a model and its line. Unlike components, an unlimited number of accessories may be ordered, and the default value for accessories is always "empty." The compatibility of some accessories may be determined by the set of components, but accessories are always considered compatible with each other.*

Example: *Models of the notebook family may allow accessories including removable drives (zip, cd, etc.), PC card devices (modem, lan, etc.), additional batteries, port replicators, carrying case, etc.*

Ex13.7. Update the specification of feature Check-configuration *of the Chipmuk web site given in Figure 13.3 by using information from the test specification provided in Table 13.1.*

Ex13.8. Derive test specifications using the category partition method for the following Airport connection check *function:*

Airport connection check: *The airport connection check is part of an (imaginary) travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary. It is described here at a fairly abstract level, as it might be described in a preliminary design before concrete interfaces have been worked out.*

Specification Signature: *Valid_Connection (Arriving_Flight: flight, Departing_Flight: flight) returns Validity_Code*
Validity_Code 0 (OK) is returned if Arriving_Flight and Departing_Flight make a valid connection (the arriving airport of the first is the departing airport of the second) and there is sufficient time between arrival and departure according to the information in the airport database described below.
Otherwise, a validity code other than 0 is returned, indicating why the connection is not valid.
Data types
Flight: A "flight" is a structure consisting of

- *A unique identifying flight code, three alphabetic characters followed by up to four digits. (The flight code is not used by the* valid connection *function.)*
- *The originating airport code (3 characters, alphabetic)*
- *The scheduled departure time of the flight (in universal time)*
- *The destination airport code (3 characters, alphabetic)*
- *The scheduled arrival time at the destination airport.*

Validity Code: The validity code is one of a set of integer values with the following interpretations

0: *The connection is valid.*

10: *Invalid airport code (airport code not found in database)*

15: *Invalid connection, too short: There is insufficient time between arrival of first flight and departure of second flight.*

16: *Invalid connection, flights do not connect. The destination airport of Arriving_Flight is not the same as the originating airport of Departing_Flight.*

20: *Another error has been recognized (e.g., the input arguments may be invalid, or an unanticipated error was encountered).*

Airport Database
The Valid_Connection function uses an internal, in-memory table of airports which is read from a configuration file at system initialization. Each record in the table contains the following information:

- *Three-letter airport code. This is the key of the table and can be used for lookups.*
- *Airport zone. In most cases the airport zone is a two-letter country code, e.g., "us" for the United States. However, where passage from one country to another is possible without a passport, the airport zone represents the complete zone in which passport-free travel is allowed. For example, the code "eu" represents the European countries which are treated as if they were a single country for purposes of travel.*
- *Domestic connect time. This is an integer representing the minimum number of minutes that must be allowed for a domestic connection at the airport. A connection is "domestic" if the originating and destination airports of both flights are in the same airport zone.*
- *International connect time. This is an integer representing the minimum number of minutes that must be allowed for an international connection at the airport. The number -1 indicates that international connections are not permitted at the airport. A connection is "international" if any of the originating or destination airports are in different zones.*

Ex13.9. Derive test specifications using the category partition method for the function SUM *of* Excel^{TM} *from the following description taken from the Excel manual:*

SUM: *Adds all the numbers in a range of cells.*

Syntax

SUM(number1,number2, ...)
Number1, number2, ...are 1 to 30 arguments for which you want the total value or sum.

- *Numbers, logical values, and text representations of numbers that you type directly into the list of arguments are counted. See the first and second examples following.*
- *If an argument is an array or reference, only numbers in that array or reference are counted. Empty cells, logical values, text, or error values in the array or reference are ignored. See the third example following.*
- *Arguments that are error values or text that cannot be translated into numbers cause errors.*

Examples

SUM(3, 2) *equals 5*

SUM("3", 2, TRUE) *equals 6 because the text values are translated into numbers, and the logical value TRUE is translated into the number 1.*
Unlike the previous example, if A1 contains "3" and B1 contains TRUE, then:

SUM(A1, B1, 2) *equals 2 because references to nonnumeric values in references are not translated.*
If cells A2:E2 contain 5, 15, 30, 40, and 50:

SUM(A2:C2) *equals 50*

SUM(B2:E2, 15) *equals 150*

Ex13.10. Eliminate from the test specifications of the feature check configuration given in Table 13.1 all constraints that do not correspond to infeasible tuples, but have been added for the sake of reducing the number of test cases. Compute the number of test cases corresponding to the new specifications. Apply the combinatorial approach to derive test cases covering all pairwise combinations.
Compute the number of derived test cases.

Ex13.11. Consider the value classes obtained by applying the category partition approach to the Airport Connection Check *example of Exercise Ex13.8. Eliminate from the test specifications all constraints that do not correspond to infeasible tuples and compute the number of derivable test cases. Apply the combinatorial approach to derive test cases covering all pairwise combinations, and compare the number of derived test cases.*

Ex13.12. Given a set of parameter characteristics and value classes, write a heuristic algorithm that selects a small set of tuples that cover all possible pairs of the value classes using the combinatorial approach. Assume that parameter characteristics and value classes are given without constraints.

Ex13.13. Given a set of parameter characteristics and value classes, compute a lower bound on the number of tuples required for covering all pairs of values according to the combinatorial approach.

Ex13.14. Generate a set of tuples that cover all triples of language, screen-size, and font and all pairs of other parameters for the specification given in Table 13.3.

Ex13.15. Consider the following columns that correspond to educational and individual accounts of feature pricing *of Figure 13.4:*

	Education		**Individual**					
Edu.	*T*	*T*	*F*	*F*	*F*	*F*	*F*	*F*
CP > CT1	-	-	*F*	*F*	*T*	*T*	-	-
CP > CT2	-	-	-	-	*F*	*F*	*T*	*T*
SP > Sc	*F*	*T*	*F*	*T*	-	-	-	-
SP > T1	-	-	-	-	*F*	*T*	-	-
SP > T2	-	-	-	-	-	-	*F*	*T*
Out	*Edu*	*SP*	*ND*	*SP*	*T1*	*SP*	*T2*	*SP*

write a set of boolean expressions for the outputs and apply the modified condition/decision adequacy criterion (MC/DC) *presented in Chapter 14 to derive a set of test cases for the derived boolean expressions. Compare the result with the test case specifications given in Figure 13.6.*

Ex13.16. Derive a set of test cases for the Airport Connection Check *example of Exercise Ex13.8 using the catalog based approach.*
Extend the catalog of Table 13.10 as needed to deal with specification constructs.

Ex13.17. Derive sets of test cases for functionality Maintenance *applying* Transition Coverage, Single State Path Coverage, Single Tranistion Path Coverage, *and* Boundary Interior Loop Coverage *to the FSM specification of Figure 13.9*

Ex13.18. Derive test cases for functionality Maintenance *applying* Transition Coverage *to the FSM specification of Figure 13.9, assuming that implicit transitions are (1) error conditions or (2) self transitions.*

Ex13.19. We have stated that the transitions in a state-machine specification can be considered as precondition, postcondition pairs. Often the finite-state machine is an abstraction of a more complex system which is not truly finite-state. Additional "state" information is associated with each of the states, including fields and variables that may be changed by an action attached to a state transition, and a predicate that should always be true in that state. The same system can often be described by a machine with a few states and complicated predicates, or a machine with more states and simpler predicates. Given this observation, how would you combine test selection methods for finite-state machine specifications with decision structure testing methods? Can you devise a method that selects the same test cases regardless of the specification style (more or fewer states)? Is it wise to do so?