Chapter 10

Functional Testing

A functional specification is a description of intended program 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. Deriving test cases 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.

10.1 Overview

In testing and analysis aimed at verification — that is, at finding any discrepancies between what a program does and what it is intended to do — one must obviously

---

1We 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.

2Here 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 22.
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.

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 are instantiated to produce individual test cases.

The core of functional test case design is partitioning the possible behaviors of the program into a finite number of homogeneous classes, where each such class reasonably be expected consistently to 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 benefit 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 aspects that must be taken into 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 helps by decomposing the functional test design process into elementary steps. In this way, it is possible to control the complexity of the process and separate human intensive activities from activities that can be automated.

Sometimes, 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.

10.2 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 \(2^{64} \approx 10^{19}\) 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. Of course, we cannot know in advance which test cases reveal faults. At a

---

3We are using the term “partition” in a common but rather sloppy sense. A true partition would form disjoint classes, the union of which is the entire space. Partition testing separates the behaviors or input space into classes whose union is the entire space, but the classes may not be disjoint.

4Note 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.
Test cases and test suites can be derived from several sources of information, including specifications (functional and model-based 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 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, criteria based solely on the code would not require the program to be exercised with a PDF file, since each part 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.

Functional specifications often address 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 for 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).
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.

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 10.1).

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”

Draft version produced February 22, 2006
class Roots{
    double root_one, root_two;
    int num_roots;
    /** Find the two roots of $ax^2 + bx + c$, * that is, the values of \(x\) for which the result is 0. */
    public roots(double a, double b, double c) {
        double q;
        double r;
        // Apply the textbook quadratic formula:
        // $Roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
        q = b*b - 4*a*c;
        if (q > 0 && a != 0) {
            // If $b^2 > 4ac$, there are 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) { // (BUG HERE)
            // The equation has exactly one root
            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 10.1: 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.
values. The test designer’s intuition comports with the observation that random sam-
ing 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.

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. 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.

Partition testing separates the input space into classes whose union is the entire
space, but the classes may not be disjoint (and thus the term “partition” is not mathematically accurate, although it has become established in testing terminology). Figure
10.2 illustrates a desirable case: All inputs that lead to a failure belong to at least one
class that contains only inputs that lead to failures. In this case, sampling each class in
the quasi-partition selects at least one input that leads to a failure, revealing the fault.
We could easily turn the quasi-partition of Figure 10.2 into a true partition, by consider-
ing intersections among the classes, but sampling in a true partition would not improve
the efficiency or effectiveness of 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
Figure 10.2: A quasi partition of a program’s input space. Black circles represent inputs that lead to failures. All elements of the input domain belong to at least one class, but classes are not disjoint.
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, includ-
ing 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 knowledge 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.

10.3 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 gen-
eration 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 it 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 rela-
tively 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 ap-
licable 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 10.3 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, designing different test cases 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, 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 spread across several modules. Thus, identifying functional features does not correspond to identifying single modules at the design level, but rather to suitably slicing the specifications to attack their complexity incrementally.

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

---

5Here the word “user” designates the individual using the specified service. It can be the user of the system, when dealing with a system specification, but it can be another module of the system, when dealing with detailed design specifications.
Figure 10.3: The main steps of a systematic approach to functional program testing.
Units and Features

Programs and software systems can be decomposed in different ways. For testing, we may consider externally observable behavior (features), or the structure of the software system (units, subsystems, and components).

Independently testable feature: An independently testable feature (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 granularity levels, 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 an 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.

Unit: We reserve the term “unit,” not for any fixed syntactic construct in a particular programming language, but for the smallest unit of work assignment in a software project. Defining “unit” in this manner, rather than (for example) equating units with individual Java classes or packages, or C files or functions, reflects a philosophy about test and analysis. A work unit is the smallest increment by which a software system grows or changes, the smallest unit that appears in a project schedule and budget, and the smallest unit that may reasonably be associated with a suite of test cases.

It follows from our definition of “unit” that, when we speak of unit testing, we mean the testing associated with an individual work unit.

We reserve the term function for the mathematical concept, i.e., a set of ordered pairs having distinct first elements. When we refer to “functions” as syntactic elements in some programming language, we will qualify it to distinguish that usage from the mathematical concept, e.g., a “function” is a set of ordered pairs but a “C function” is syntactic element in the C programming language.
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 for 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 callbacks 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 should be selected to form test cases. Representative values can be identified directly from informal specifications expressed in natural language. Alternatively, 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 through 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 means 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 a 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 procedure 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 would otherwise be derived for patterns of length zero. Combinatorial considerations reduce the set of test cases by limiting the number of combinations 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 generated test cases generated and modify the selected subset of test cases according to budget considerations. Subsets of combinations of values, i.e., potential special cases, can often be 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 test case specification. Test cases are implemented by creating the scaffolding required for their execution.

### 10.4 Choosing a Suitable Approach

In the next chapters we will see 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 techniques are complementary, i.e., they apply to different aspects of the same specification or at different stages of test case generation.

The choice of approach for deriving functional test cases depends on several factors: the nature of the specification, the form of the specification, expertise and experience of test designers, the structure of the organization, availability of tools, budget and quality constraints, and the costs of designing and implementing 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 using a partitioning method with constraints, such as the category-partition method described in Chapter 11, while unconstrained combinations of values may suggest a pairwise combinatorial approach. If transitions among a finite set of system states are identifiable in the specification, a finite state machine approach may be indicated, while inputs of varying and unbounded size may be tackled with grammar based approaches. Specifications given in a specific format, e.g., as decision structures, suggest corresponding techniques. For example, functional test cases
for SDL\textsuperscript{6} 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 that technique over a catalog based approach when both are applicable, while a company that works in a specific application area may require the use of domain-specific catalogs.

**Tools** Some techniques may require the use of tools, whose availability and cost should be taken into account when choosing a 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, if the primary constraint is rapid, automated testing, and reliability requirements are not stringent, random test case generation may be appropriate. In contrast, thorough testing of a safety critical application may require the use of sophisticated methods for functional test case generation. When choosing an approach, it is important to evaluate all relevant costs. For example, generating a large number of random test cases may necessitate design and construction of sophisticated test oracles, or the cost of training to use a new tool may exceed the advantages of adopting a new approach.

Many engineering activities require careful analysis of trade-offs. Functional testing is no exception: Successfully balancing the many aspects is a difficult and often underestimated problem that requires 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 yield a set of test cases to 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 lead to substantial savings.

**Open research issues**

Functional testing is by far the most common way of deriving test cases in industry, but neither industrial practice nor research have established general and satisfactory methodologies. Research in functional testing is increasingly active and progresses in many directions.

Deriving test cases from formal models is an active research area. In the past three decades, formal methods have been mainly studied as a means for proving software

\textsuperscript{6}SDL (Specification Description Language) is a formal specification notation based on extended finite-state machines, widely used in telecommunication systems and standardized by the International Telecommunication Union.
properties. Recently, attention has 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 models. 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. Formal notations can support 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. A variety of formal models could be used in testing. The research challenge lies in identifying a trade-off between costs of creating formal models and savings in automatically generating test cases.

- Development of a general framework for deriving test cases from a range of formal specifications. Currently research addresses techniques for generating test cases from individual formal methods. Generalization of techniques will allow more combinations of formal methods and testing.

Another important research area is fed by interest in different specification and design paradigms, e.g., software architectures, software design patterns, service-oriented applications, etc. Often these approaches employ new graphical or textual notations. Research is active in investigating different approaches to fully or semi-automatically deriving test cases from these artifacts and studying the effectiveness of existing test case generation techniques.

Increasing size and complexity of software systems is a challenge to testing. Existing functional testing techniques do not take advantage 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, Littlewood and Strigini [FHL98] is a good starting point to the more recent literature considering the relative merits of systematic and statistical approaches.

**Related topics**

Readers interested in practical technique for deriving functional test specifications from informal specifications and models may continue with the next two chapters, which de-
scribe several functional testing techniques. Reader interested in the complementarities between functional and structural testing may continue with Chapters 12 and 13 which describe structural and data flow testing.
Exercises

Ex10.1. In the “Extreme Programming” (XP) methodology [Bec99], 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?

Ex10.2. Compute the probability of selecting a test case that reveals the fault inserted in line 25 of program Root of Figure 10.1 by randomly sampling the input domain, assuming that type double has range $-2^{31} \ldots 2^{31} - 1$. Compute the probability of selecting a test case that reveals a fault, assuming that both lines 18 and 25 of program Root contains the same fault, i.e., missing condition $a \neq 0$. Compare the two probabilities.

Ex10.3. 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:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, the 10 digits

+, -, *, /, %, the operations
Choosing a Suitable Approach

"=" to display the result of a sequence of operations

C, to clear display

M, M+, MS, MR, MC, where M is pressed before a digit to indicate the target memory, 0...9, keys M+, MS, MR, MC pressed after 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: 5 + 1 0 M 3 MS 8 0 M 3 MR = prints 65 (the value 15 is stored in memory cell 3 and then retrieved to compute 80 - 15).