J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

PROGRAM TESTING

Goals and Methods

Different goals can be identified for software testing. Two of the most common are reliability and fault reduction, but others are important also, such as the evaluation of performance, security, and usability. Reliability is qualitative, when we establish that a program implements required functionality. It is also statistical, when we establish that the expected number of failures per CPU hour is less than some required bound. Fault reduction concentrates on the detection of faults. This is obviously related to reliability, but it is often a preliminary phase in which we eliminate as many faults as possible, before considering reliability estimation. It is generally accepted that software must undergo some form of testing, even when highly effective production processes are followed or mathematical proofs are used to demonstrate the correctness of algorithms or other critical system properties. We simply cannot have confidence in a system that has not been tested.

Software *testing* refers to methods in which a piece of finished software is run on actual data. *Analysis* refers to methods in which software or other development products are examined for necessary or sufficient correctness. This article does not consider the more formal kinds of analytical methods, in which the goal is to prove correctness and for which a set of formal system specifications is necessary, but it includes descriptions of the less formal methods associated with and normally included in discussions of testing.

Theoretical Limits

We can prove that there is no general, algorithmic procedure which, given a program and its specification, generates a finite set of tests to determine if the program is correct. This follows from the more general result that it is not possible to determine if a program is equivalent to a given specification. This latter result is proved using results from the theory of computation and is related to the famous halting problem: it is not possible to determine, in general, if a program terminates for an arbitrary input case. This implies that we know of no general method for determining correctness, even independently of what the program is supposed to do, assuming that termination is a necessary property.

One of the more famous negative quotes about testing is "testing can only be used to prove the presence of bugs, not their absence." This quote has been used to justify more formal approaches to the software correctness problem in which a program is proved equivalent to a formal specification. However, the formal approach is not a guaranteed solution to program correctness, because proofs must be constructed by hand. Support tools can be used, including proof checkers, but the proofs are labor intensive, complex, and error-prone. Formal approaches are often suited to analyzing algorithms and designs, where they focus on essential logic, rather than implemented programs, where they get lost in the details.

The previous quote about testing implies that when we run a program on a test and it operates correctly, that we conclude only that it runs on that specific case, which is of little value if the program has an effectively

infinite input space. This, together with its known theoretical limitations, puts testing in a bad light, and implies that it is not a useful approach. However, there are two arguments in favor of testing which contradict this conclusion. First, the results of a single test may contain more information than whether or not a program is correct on a single point in its input domain. If random tests are selected, then we can make useful estimates of the size of the input domain over which the program might fail. If a program has a succession of long successful runs in a stable production environment, then it is reasonable to expect similar behavior in the future. Second, we have a large body of empirical evidence about the kinds of code or data where programs are typically incorrect, and know that certain tests have far more fault-revealing power than others. Even though we know we cannot prove that a program is correct by testing, we can test to establish statistical estimates of its reliability, and, from empirical evidence, we know that testing removes a large percentage of a program's faults.

Empirical justification for testing methods can be extended to informal analytical methods. Cost-effective procedures and techniques have been developed to discover faults. Informal analysis is also justified on procedural grounds. Even if formal analysis were easy to use, it could not be applied in the early stages of software development where the use of informal documents is necessary. Early defect detection is critical to control cost. The later a defect is discovered, the more costly it is to repair it and its associated development product.

Classification of Methods

Testing and Analysis. There are a variety of dimensions along which it is possible to classify methods. The most basic classification is into analysis and testing. In some of the older literature these are referred to as *static* and *dynamic analysis* methods (1). Static analysis usually refers to informal or partial methods, as opposed to formal analysis.

Life-Cycle Phase and Phase Products. Methods may be associated with particular life-cycle phases or designed for particular phase products. Analysis methods, as opposed to testing, are often associated with early life-cycle phases. Testing is often associated with coding, validation and verification, and maintenance phases. Certain kinds of test construction methods are oriented toward design products, and others are based directly on code. Specific analysis techniques have been developed for specific kinds of requirements and design products.

Errors, Faults, and Failures. Another approach to classification involves the use of the IEEE (Institute for Electrical and Electronic Engineering) definitions for *errors, faults,* and *failures.* An error is a mistake made by a programmer which results in a fault in the code. The fault in the code causes a failure during program execution. Failures are associated with incorrect output or incorrect performance characteristics. In this article we use the word *defect* in a general sense to refer to a software error, fault, or failure. Methods are oriented toward one of these three ways of looking at defects. Error-oriented testing and analytical methods are based on observations of the ways in which programmers make errors, such as forgetting to take care of extreme cases. Fault-oriented methods focus on typical bugs found in code, such as the use of the wrong variable name. Failure-oriented methods identify undesirable behavior and then see if such behavior is possible.

Statistical and Nonstatistical Methods. Both analysis and testing may be nonstatistical. They may be general methods or designed for specific classes of faults. Reviews are usually nonstatistical. Testing methods may be either. Nonstatistical testing methods may be aimed at specific faults or may be general guidelines for decomposing a program's input domain into relevant test cases. Statistical methods generally rely on knowledge of a program's input distribution and are usually testing techniques. It is, however, possible to associate statistical information about effectiveness with any method, by gathering empirical data.

Positive and Contrapositive Methods. In the positive approach to evaluating a program, we try to demonstrate that it operates correctly by showing that it performs as intended or that it has certain desirable properties. In the contrapositive approach, we show that the program is acceptable by showing that it does

not have undesirable properties. Typical positive approaches include functional testing, where we use testing to show that a program works on standard input cases. Typical contrapositive methods involve static analysis methods that look for uninitialized data or inconsistent interfaces.

In this article, method descriptions are organized as follows. Analysis has been separated from testing, and for each of these, we distinguish between positive and contrapositive methods. Within contrapositive methods, we classify according to a method's error, failure, or fault orientation. In some cases, a method could be classified in different ways. For example, structured walkthroughs are classified as positive methods, even though they may involve focusing on potential faults, because walkthroughs are positively oriented. Additional sections discuss method adequacy, standards, process improvement, and management.

Analysis Methods

Positive Methods.

Code Reading. Code reading is carried out by individual programmers when they check over their code for reasoning or implementation defects. This kind of review is supported by coding standards and documentation. One of the advantages of structured programming is that, if programs are constructed from the basic structured constructs, using a compositional approach in which one structure is nested inside another, the result is a top down design. Reading code with such a design requires only a simple pointer to scroll through the text of a single construct and a stack to remember the location of the text pointers when a transfer is made to a lower level of detail upon entry into an embedded construct.

Code reading is often more effective if it is repeated while focusing on different properties. For example, if a program processes all of the records in a file, it can be read with respect to proper processing of the file as a whole and then with respect to the processing actions taken for individual records. A strategy similar to that described later for functional testing could be used to identify different aspects of a program's functional requirements, and code reading could focus on these, one at a time. If checklists are used to identify potential problem constructs, then the method becomes contrapositive in its orientation, as in inspections, discussed later.

Supporting documentation in the form of comments can be used. When reading code, comments can be used to note when abstract operations or program states are established and to record working assumptions about expected previously established properties that need to be checked out at some later point. For example, on entry to a data structure searching routine, a comment may denote the assumption that the structure is initialized in some way. Comment-supported analysis is discussed later in the section on necessity analysis.

Structured Walkthroughs. This is one of the earliest review methods (2). It is specifically designed to be carried out by a group, whose members have roles to play, and depends on management guidelines. It is used for a variety of review items, including requirements, design, code, maintenance, and test plan artifacts. It is meant to be applied to documents which have reached completion, as opposed to partial products.

The goal of a review is peer group analysis of a product, rather than management performance evaluation. It is carried out by a coordinator, a presenter, a recorder, and reviewers. The coordinator arranges the session, manages it, and arranges for follow-on reviews. The presenter is the person responsible for the review item, and presents it to the group. The recorder keeps minutes. Other possible team members include a domain expert, a standards expert, or a user representative.

Walkthrough sessions are scheduled for one to two hours, and participants of a walkthrough are expected to review walkthrough materials beforehand. Minor errors are corrected as they are found, and corrections of a more serious nature are developed off-line and reviewed in a later session.

Walkthroughs are also carried out at various points, but it is recommended that they be used for complete deliverables. They are applied to specifications and designs, as well as code. When applied to designs, the consideration of alternatives is often useful in evaluating a design's effectiveness. If applied to code, the code

should have been compiled, but not thoroughly tested. Experience indicates that many bugs are found in code when multiple reviewers are present, as opposed to the effort of a single code reviewer.

Walkthroughs are applied to process documentation, such as test plans, and program products. An early review of a test and analysis plan identifies problems and induces commitment on the part of the development staff.

Contrapositive Methods.

Error-Oriented Analysis.

Necessity Analysis. Errors are classified as *errors* of *omission* or *commission*. Errors of omission result from the programmer forgetting something or from requirements problems in which some processing alternative was never considered. Necessity analysis involves identifying program states or other properties necessary for correct program operation. If a program does not have such a property, we can assume an error of omission. Necessity analysis is either global or local.

In global analysis, we assume the existence of objects with attributes that take on different property values. Abstract program states consist of combinations of properties. Global necessity analysis involves identifying states that should be theoretically possible in a program. This means that there should be potential flows of control along which they take place. The term "theoretically possible" is used because it is not easy to determine if required control flows actually occur, that is, if they are feasible in the sense that there is data that cause them to be followed. Global analysis is carried out by inserting program assertions that indicate when objects acquire or lose properties.

Local analysis is implemented by *assertion* comments that describe changes to object properties and by *hypothesis* comments that describe expected state properties associated with specific program locations. If a hypothesis is not justified from assertions and code, then a programming error has occurred. Extensive work involving the use of local necessity analysis is in (3,4).

Fault-Oriented Analysis.

Static Analysis. This is a catch-all term which is applied to a number of different kinds of analysis. The most common and best known involves determining if a program references variables or data structures that have not been set (5). Related analyses identify data not used. Other kinds of static analysis are related to the semantics of particular programming languages, such as checking to see that there are no illegal jumps into or out of the middle of loops.

Inspections. Reviews that use checklists were popularized by Fagan in (6) and are sometimes called inspections, as opposed to code reading and walkthroughs. Checklists are arranged in groups describing different kinds of program properties that are fault-prone. For example, a data checklist might describe faults in which code was not written to cover boundary cases or some kind of data was not used.

In Ref. 6a checklists are described for requirements, specifications, designs, and implementation. Each of these refers to potential problems. For example, the design checklist asks: "Was the design understandable?", "Were the reasons for major data elements described?" and "Were the design objectives clear?" The specification checklist asks: "Were the specifications reviewed for completeness?" and "Was performance adequately specified?"

Failure-Oriented Analysis.

Safety Analysis. This approach involves identifying hazardous program behavior or states, that is, failures. The analyst determines which kinds of causative data or program states could produce those failure states. This backward analytical process is repeated until it is determined that causative states or data can occur and we have a problem, or we know that they do not occur. In the former case, the potential for hazardous behavior is established. The process is made more systematic in a variety of ways. For example, we could distinguish between conjunctive and disjunctive causes. In the case of conjunction, all the listed causal factors are needed. We need to prove only that one of them cannot appear. In the case of disjunction, any cause is sufficient, and we need to demonstrate that all of them cannot appear.

Nonstatistical Testing Methods

Positive Methods.

Functional Program Testing. One of the more widely used methods at all stages of testing involves identifying expected functionality and constructing associated program tests. Traditionally, functions were identified from informal program requirements documents. Structured functional descriptions might be used to assist in identification. For example, a program for validating record fields might contain the following structured function description:

```
validate records
    validate name field
    validate address field
    validate account debit amount
    validate date
    if invalid field
    print field
    print error message
    if valid field
    add record to account transactions file
```

In general, there are two ways of identifying functions during functional testing: operational identification and data partitioning. In the operational approach we identify algorithms, steps in algorithms, system actions, and other kinds of program and system operations. In the data approach we identify data subcases associated with different subfunctions or operations. These are two ways of looking at the same thing, but one point of view is often more natural or convenient than the other, so that both should be considered.

Operationally Oriented Functional Testing. Empirical observations of faults indicated that it is often enough to simply test a program's functions to provoke fault-revealing failures (7). This led to the idea of "broad spectrum" functional testing in which the goal is to identify and test design and implementation functions, as well as requirements level functions. These are often identified from design and implementation documents. In the structured approach to systems development, we use module structure diagrams to identify functions. In state diagram models, functionality is associated with state transitions. If pseudocode is used during detailed design, then the prose parts of the pseudocode often refers to detailed design functions. During implementation, functions are created that control loop termination and select between alternatives. These lower level implementation functions also need to be tested. At all levels of abstraction and development, it is important to identify relevant operations and construct associated tests. Detailed information about the broad spectrum functional approach is in (8).

Complete functional testing requires that intermediate values of program variables be observable. This is necessary to analyze embedded implicit design and implementation functions. The examination of intermediate values during testing is found by many to be very effective for evaluating a program, supporting the argument for explicit testing of lower level functions.

Functional system capabilities and their corresponding tests are documented in the form of a function/test matrix. Matrices are also used to record other information, such as test status.

Data-Oriented Functional Testing. Data-oriented functional testing is often associated with procedures for decomposing program input domains into subdomains. One approach is to try to subdivide the input domain of a program into subsets, such that for any subset the program either succeeds or fails on all data in that subset, presumably because the program performs the same indivisible function for each subset (9). Systematic

approaches involve identifying data objects and their attributes. Values of these attributes are used to identify data object property classes. Combinations of property classes define subdomains or domain equivalence classes (10). In those cases where there are too many combinations, compromises are used in which it is only necessary to "cover" all pairs of property classes with some test.

It is useful in data-oriented testing to consider different functional capabilities associated with output in addition to input data classes. For example, a numerical analysis program may be able to integrate some kinds of functions but not others. It has two kinds of output, depending on what it finds out while trying to compute an integral.

For some applications, data-oriented testing may be systematically implemented with formal grammars. These define the input space and different kinds of data subspaces. For example, the input to a compiler parser could be characterized by a programming language grammar. The grammar could be used to generate different kinds of programs and program constructs.

Data-oriented functional testing at the higher system levels of testing is associated with requirements or design documents. For example, if structured analysis is used to analyze requirements, then data dictionaries describing data transform items are available. If state diagrams are used for design modeling, data descriptions associated with states provide information about different functional subcases.

Contrapositive Methods.

Error-Oriented Testing. Functional testing is a positive method in the sense that it tries to show that a program is acceptable by demonstrating that it has desired functionality. Error-oriented methods are contrapositive in that they try to show that a program is acceptable because the programmer did not make mistakes. There are several approaches. The most common concentrates on typical input data cases that programmers often fail to handle correctly resulting in incorrect program behavior. Typical examples include extremal values, such as minimum data structure sizes or boundary data values.

Other examples of error-prone data are identified with logical classes of implemented functions, called clichés (11), which have associated special cases that have to be tested. For example, if a program has a component that searches for a data item in a data structure, we need to consider the case where the data item is not present. We need to consider also other special cases where the item is present but duplicated.

Fault-Oriented Testing. Fault oriented methods revolve around identifying specific program faults, such as a wrong operand or misspelled variable name. The most widely known fault-based method is *mutation testing* (12). Suppose that T is a set of tests. The basic idea is to introduce some change into a program P, producing a mutant P′, and then to see if there is a test in T that distinguishes P from P′. If not, we conclude that T is inadequate to distinguish faults associated with the change made in P to produce P′ or that P and P′ are coincidentally equivalent programs. The changes made to evaluate a test set are called *mutations*. They include alterations, such as using a different variable name or adding or subtracting a constant from an expression.

In general, mutation testing is not widely adopted as a testing method because of the expense of using it. There are many mutations that might be made to a program, and the program needs to be run for each of these. Methods have been devised for speeding up this procedure, including parallel program execution, but the widest use of mutation analysis is as a standard. Suppose that we wish to compare two testing methods. We evaluate them by seeing how effective they are in distinguishing between a program P and its mutations P′.

Other fault-based approaches include *weak mutation testing* (13). In this method, we identify the classes of data needed to make an individual program statement and its mutated versions act differently. The method requires local monitoring of program statements and the data over which they are executed. Weak mutation testing allows testing many mutations at once, but requires that, when a statement is not covered by some necessary distinguishing data class, the programmer confirm that there is no input data that causes its execution over such data. This is a modified version of the coincidental program equivalency problem that exists for general mutation testing.

One of the early forms of program defect classification involved *computation, domain,* and *missing path* errors (14). Domain errors occur when a program performs the correct computations but applies it to the wrong data. This kind of defect is particularly subtle if the correct and incorrect domains are close. For example, suppose that a program performs different computations when it detects that an account balance has reached zero, but it only does this when there is a debit, that is, when the balance is less than zero. Domain analysis (15) recognizes the importance of generating tests that lie on and near the boundaries of data subdivisions. In our example, this would be balances that are zero, and slightly larger or smaller than zero.

Failure-Oriented Testing. As in the case of failure-oriented analysis, the emphasis here is on output and how it is incorrect. Failure-oriented testing identifies incorrect or hazardous data or behavior and attempts to generate tests that cause it to occur (16). In this approach, as in others, the process of generating the special kinds of tests needed by the method may be enough to cause the identification of faults, before any testing is carried out.

Statistical Testing Methods

Statistical testing is used to predict software reliability. To use statistical testing, it is necessary to know a program's operational distribution, the frequency with which different inputs can occur.

In general, statistical testing is a positively oriented approach, in which we demonstrate that a system works as expected. Two kinds of statistical testing are discussed, associated with different kinds of conclusions that we draw on the basis of our tests.

Confidence Estimation for Failure-Density Bounds. One approach to statistical testing results in confidence in a bound on a program's failure density. The *failure density* for a program is the fraction (weighted by its operational distribution) of the input domain over which the program fails. Suppose that a program is executed over many randomly drawn tests and no failures are seen. Then we expect the failure density for the program to be small. More formally, suppose that F is some desired bound on the failure density, and we see no failures in N tests. If the failure density is not bounded by F, then the probability of this happening is at most $(1 - F)^N$. Then we say that we can have confidence at least $1 - (1 - F)^N$ that the failure density must be bounded by F (17,18,34).

In the more general case, we consider the situation where we see n failure-free executions in N tests, where n may be less than N. In this case we use similar reasoning to show that we have confidence at least C that the failure density is bounded by $1 - P_F$, where

$$P_F = \binom{N}{0} F^0 (1-F)^N + \binom{N}{1} F(1-F)^{N-1} + \cdots + \binom{N}{n} F^N (1-F)^{N-n}$$

One of the problems with statistical testing is the large numbers of tests that must be used to gain high levels of confidence in tight bounds. For example, we need only seven tests to be 80% confident that a program's failure density is less than 20%. But we need 460 tests to be 99% confident in a failure-density bound of 1%. For some programs, for which automated test data generation and automated output validation is possible, large numbers of tests can be run, but this is generally not the case.

Software Reliability. Software reliability is defined as the probability that a program operates without failures for a specified time interval. Time is measured in different ways, but in general, it is accepted that execution time provides superior results. Of particular interest is a program's *failure intensity,* that is, the

number of program failures per CPU hour. This is contrasted with statistical testing in which we estimate, with some confidence, the probability that a program will fail on a randomly drawn input.

Reliability measurement is based on the assumption that the general form of a failure-intensity function can be determined and that the parameters needed to adjust that general form for a particular program can be predicted or estimated. From the point of view of testing, we would run a program on its operational distribution and then fit the parameters from observed behavior. The reliability formula used in one of the more common models (19) is given by

$$f(x) = f_0(1 - x/F)$$

where f is the failure intensity as a function of x, the expected or average number of failures experienced to some point in time, f_0 is the initial failure intensity, and F is the total number of failures that will occur over the life of the system. Initial parameter estimations are derived from characteristics of the software, such as its size. After the system is running, parameters are estimated from observations. For example, the initial failure intensity is observed, and then after some period of time the total number of failures are estimated using the observed initial failure intensity and the observed number of failures up to that point.

There are a number of advantages to the reliability approach. The alternative confidence estimation approach is most easily used when we see a run of failure-free tests. The reliability approach makes the realistic assumption that a program always fails in use and that what is of interest is how often this occurs. In addition, the reliability approach is more easily used when a program is put into an operational environment for testing, such as in beta-testing. In this case we expect that the system will experience failures, faults will be found and corrected, and new faults may be introduced. The reliability approach accommodates all of these possible variations because it only looks at behavior over time.

The reliability approach is also better suited to situations where a program has an internal state and/or is nonterminating. The statistical confidence approach emphasizes testing functional input/computation/output programs over independent, random tests. This is more difficult for systems which have an internal state or whose input is affected by interaction with the program.

The drawback to the reliability approach is that it may not have a consistently high degree of accuracy because it depends on adopting a model and estimating parameters in that model. Alternative models can be used which result in indifferent results, and parameter estimates may be inaccurate.

Operational Distributions. Different methods are used to specify a program's operational distribution. In the simplest case, an input domain is partitioned into discrete subsets, or partitions, with an associated frequency, within which data is selected according to the uniform distribution. One advantage of this approach is that it achieves the effects of statistical testing without actually knowing the subdomain frequencies. Suppose that N random tests from a program's input domain are sufficient to determine a required level of confidence in a bound. Then if N tests are randomly selected from each partition element, we achieve the same level of confidence achieved by those N random tests that were selected according to the operational distribution from the whole input domain (20).

If a program is periodic and carries out a fixed set of interactions in a fixed order during repeated equally long time periods, then the structure of its operational distribution is only slightly more complex than that of a simple functional program. For each of these interactions we assume that we have an individual operational distribution and the construction of a test case involves choosing an input for each of the input interactions.

More complex programs involve variable length interactive sequences and require more complex operational models. Possible approaches include the use of Markov state diagrams (21). In this approach a system is in one of a number of states. Input received during a state causes a self-transition to the same state or a transition to a new state. Probabilities are associated with each of the classes of input that are associated with the transitions. The model has an initial and a termination state. Instances of system interactions correspond

to paths from the initial to the termination state, which are randomly generated by "walking" through the model. One of the advantages of the Markov model is that expected test properties, such as the length of the average test sequence, are analytically determined.

More sophisticated models take an entire event's history into account during test sequence generation, not just the previous state/event, as in the Markov model. One approach (22) involves identifying event history classes which correspond to saved internal states. An event is an interaction with the system, and a history is a finite sequence of events. It is assumed that the set of all events is divided into classes and that, for each event history class, we know the probability of a subsequent event for each event class. In addition, for each event class, we know the distribution of individual events in that class. We assume that we know which new event history class occurs for each event history followed by an event. To complete the model, we need to know the frequencies for the expected lengths of event sequences.

The non-Markov event history model is used to generate test data as follows. First we choose an event history length. We assume that we start with an empty event history. From this we choose some event class according to our event class distribution. Then we choose a particular event from within this class. This gives us an initial event history. Using our knowledge of its class, then we pick a new event class and new event. From this we construct an augmented event history. This continues until we have an event sequence of the desired length, which constitutes a single test case.

To use the Markov and event history test generation models for confidence estimation, we need to generate finite system interactive sequences and to assume that there is no behavior affecting internal system data carried over from one session to the next. A "session" is the use of the system through an interaction sequence. This assumption is not explicit in the reliability model but there is a corresponding underlying assumption that internal state effects on the failure behavior of a system are predictable and subordinate to the effects of system usage interactive sequences.

When there is persistent internal data associated with separable functional capabilities, such as, for example, accounting procedures, then we might be able to isolate this aspect of the system. We would model the rest of the system using one of the previous models and come to statistical conclusions about all properties of the system other than persistent, data-dependent properties. But there are kinds of systems, such as data bases, with large amounts of retained data and for which internal data states have a major role in determining the actions of the system. These require other approaches.

Adequacy and Completeness

One of the problems with informal methods is knowing if their application is systematic and complete. For example, how do we know that an inspection is thorough? How can we know that a set of functional tests is adequate?

Analysis and Completeness. In our discussion of reviews, we identified the role of a recorder, who is responsible for prior distribution of documentation, keeping track of results, and the preparation of a summary report. If reviews involve checklists, then a procedure in which the reviewer confirms the use of each checklist item is used. When checklists are not used, other kinds of evidence are desirable. In general, review completeness is documented with review session planning forms and session review completion records.

In the case of code reading, where the review activity is not a group activity and formal records are not kept, one approach to adequacy documentation is to require comments. Adequate comments document a systematic understanding of the code. Attempts have been made to require a certain density of comments, such as one comment per five lines of operational code. This is viewed as a code reading adequacy check, in which the comments are viewed as records of an informal analysis carried out by the programmer. Less informal approaches involve local necessity analysis of the kind described previously in the section on reviews. For example, comments are used to document necessary input data properties in module headers. Type checking

accomplishes some of these objectives, but generally we need to document other properties, such as data *flavors,* which change during a program scope and which are not associated with a single type (23). Other comments are used to document places where properties of data objects are set (assertions) and others to document assumptions about previously established properties (assumptions). Comments are also used to document reasoning involving expected previous and future program operations (3,24).

Testing and Completeness. Completeness measures for testing are statistical or nonstatistical. For some methods the distinction between test generation and adequacy of test sets does not exist. For example, confidence-based test data selection involves generating enough tests so that a required level of confidence is reached. The confidence level is a measure of test completeness. In the case of mutation testing, enough tests have to be generated so that all nonequivalent mutants are distinguished from the original program by at least one test. This is both a test generation strategy and a measure of test set completeness.

Most testing methods associated with test completeness rather than test selection involve some form of coverage measure. Different kinds of program components or observable states are identified, and test coverage is said to be complete if it involves tests that cause all or some required percentage of these items to be executed. Of course, this can also be viewed as a test generation strategy, in which the goal is to generate tests that cause the required coverage. But if we assume that testing normally involves methods, such as functional testing, in addition to coverage-oriented methods, then it seems reasonable to view a coverage method as an adequacy check rather than a primary test generation method.

Nonstatistical Coverage Measures. The most common form of test coverage adequacy is branch coverage. This requires that each branch be executed on at least one test. In practice, testers settle for some percentage of the branches, such as 85%, except when low-level unit testing is involved. A variety of tools are available which monitor branch coverage during testing, combine the results of multiple tests, and prepare reports on coverage.

If all statements are executed during a set of tests, then we conclude that we have tested the operations created to implement a program's required subfunctions. If all branches are covered, we conclude that we have tested special data cases associated with functional subcases. From this point of view, branch and statement coverage are measures of adequacy for functional testing. However, certain kinds of embedded functionality are tested only when particular combinations of program components are executed. Two approaches involving combinations are discussed.

Linear Code and Jump Sequences (*LCAJSs*) are sections of code involving a sequence of imperative program statements that start at the beginning of a program or the target statement for some jump, traverse a sequence of jump-free code, and then end with a statement that causes a jump in control flow. LCAJS (25) test data coverage has been extensively used as a test to measure adequacy.

Other approaches to complex coverage involve combinations of statements and branches with a data-flow relationship (26). For example, suppose that one statement assigns a value to a variable that is subsequently referenced in another statement. Then the second statement has a data-flow dependency on the first. Simple data-flow coverage requires that all such combinations or some percentage of them be tested on at least one test. Extensive work has been carried out analyzing different possible kinds of data-flow coverage possibilities, but the methods have not been widely used in practice and have not replaced simple coverage approaches involving single branches.

More demanding coverage requirements are proposed which focus on complex Boolean expressions, like those found in branching statement conditions. In addition to testing the true and false branches for the conditions, it is also required that individual components of the expression take on a range of combinations of values. One simple extension requires that each component take on both a true and false value on some test. Others require combinations of values which would reveal if some component is "stuck at false" or "stuck at true" in the manner of hardware circuit testing (27).

Additional suggested coverage methods include *path testing* in which we try to cover complete program paths. Because even a simple loop-free program contains many paths, such methods are usually impractical.

Variations designed to deal with loops require that each loop be tested on at least one test that causes multiple iterations, one that takes each exit, and one that causes minimal loop iterations (28). Other methods involve the use of *basis paths*. If we are trying to cover all possible branches, then it is efficient to begin with a path that covers as many branches as possible, then one that covers as many uncovered branches as possible, and so on. Tools have been built to generate such paths (29). Unfortunately, these paths may be syntactically but not semantically possible, so that they may be used only as guides. In addition, at higher program levels, they are so long that it is difficult to determine which data causes them to be executed.

Methods that define coverage in terms of program structures, such as statements, branches, LCAJs, and paths are called *structural coverage* measures. One coverage measure is said to *subsume* another if satisfaction of the first method guarantees satisfaction of the second. Extensive studies of subsumption relationships have been carried out (30,31).

Statistical Coverage Measures. Traditional coverage measures have a number of shortcomings. One is that they are best suited to unit testing and are difficult to use for entire systems or subsystems. Something more abstract is needed in this kind of situation. One solution is to use a statistical approach, similar to that used in statistical testing. Assume that states in a program correspond to combinations of object properties which are set and reset as a program executes. Statistical coverage adequacy corresponds to confidence that new states will not appear in subsequent tests or, more specifically, confidence that the probability of seeing a new state in some additional test is less than a required bound.

Suppose, for example, that we are interested in coverage with respect to Ada task interaction. Objects are tasks, and the relevant state properties are their status with respect to call and accept entry operations. States correspond to combinations of task interaction status, and coverage is adequate when we are confident that no new task interaction status combination will appear on subsequent tests. The same formula used to compute confidence in a bound on a program's failure density is used to compute a bound on the probability of occurrence of new program states (32).

The statistical approach can also be used with complex, lower level coverage measures, such as those in which combinations of data-flow-related items must be covered. Because some of these are infeasible in the sense that there is no data to cause them to be executed, it is difficult to require that some percentage be covered, because that percentage may exceed the total executable percentage of such coverage items. In the statistical approach we require that we continue covering such items until it is unlikely that some new coverage item will appear on a subsequent test.

The statistical coverage approach achieves high levels of partial coverage if it is inexpensive to measure coverage. To guarantee that a program is valid over a set of tests that achieves high coverage, it is not necessary to validate behavior for the tests whose repeated coverage only increases confidence that no new coverage will occur. It is only necessary to validate behavior for one instance of each achieved coverage.

Complexity-Based Coverage for Analysis and Testing. It has been observed that complex modules are more error-prone than simpler modules and should therefore be more thoroughly tested or analyzed. Complexity measures, such as the cyclomatic number for a program's flow graph have been suggested (29), where complexity is equal to

$$\text{edges} - \text{nodes} + 2$$

It can be argued that structural coverage normally results in more tests for more complex programs, because such programs have more structure to cover. But if we have a small number of modules that are much more complicated than the others, then adequate testing requires that we give them an even closer look during analysis.

Auxiliary Methods

In the discussion so far, we have been concerned primarily with general purpose methods. In this section we address some of the issues associated with programming styles or application areas. Particular emphasis is given to object-oriented programming. A very brief discussion of some of the relevant topics for particular application areas is also given. Many of the methods have been discussed are supported with special purpose tools which were mentioned during their description. A few of the more important, useful general purpose tools are also noted here.

Object-Oriented Testing.

Analysis. In the object-oriented approach, objects and classes are developed during requirements and design and then refined and augmented during implementation. Analysis and test generation is carried out at all stages of this process. Object-oriented systems often consist of many small subsystems or classes, so that integration analysis is especially important. One class in a subsystem often has expectations about the responsibilities supported by other subsystem classes. In addition, it is often the case that functionality in a single class or object is somewhat arbitrarily assigned to different possible methods so that one method may expect that relevant functional capabilities are assigned to some other method. These kinds of expectations need to be documented and checked. One approach is to use local necessity comments analysis like that described previously.

Nonstatistical Testing. Unit testing of object-oriented systems is more complex than that of procedural systems because we are testing objects that consist of collections of methods that interact with common data structures, not just single isolated procedures. Individual methods are tested by conventional techniques, but object testing involves unit-level integration issues. Testing for individual object classes requires identifying relevant classes of method combinations and classes of method input data (33).

Unit object testing may run into observability problems. Objects typically have private data that is not observable outside an object. This makes it difficult to trace intermediate values and, hence, the correctness of subfunctions in methods or of states within which methods are being used. One suggested approach is to include a monitor object to which methods are expected to send test value data.

Integration testing requires testing interacting methods and objects and also testing inheritance. Inherited methods must be tested within their new context, and it has been found that it is not enough to simply reuse inherited class tests. This is especially true if polymorphism is involved. In general, integration testing is a more important issue in the object-oriented approach in three ways. First, more integration is involved because object-oriented systems consist of many small interacting object modules. Second, methods must be tested within the context of different object states, in combination with other methods in their parent objects, and not just as isolated procedures. And finally, inherited methods must be retested within the context of their new subclasses because they may interact with data and new methods in that class.

Systems testing for object-oriented systems typically involves developing scenarios describing system use. Functional systems testing involves identifying both scenario classes and method subdomain input data classes.

Statistical Testing. Statistical testing for object-oriented systems requires random selection of method subdomain data and usage scenarios. More complex operational distribution modeling than that needed for simple functional systems, such as those that use event histories or Markov state diagrams, are needed.

Adequacy and Completeness. Traditional branch coverage is used to test individual methods, but more abstract coverage is needed for collections of collaborating classes and objects. One approach is to monitor observable events, such as method calls, and to use abstract system state coverage. In this case, a state of the system corresponds to the combination of methods that the objects are currently "in" (i.e., methods to which messages had been sent that are currently being processed). Integration testing might be limited to combinations consisting only of pairs of object methods, where one object, while processing a message, sends a message to some other object.

Another approach to coverage is to consider states of individual objects known from state-denoting assertions. For example, a stack might be asserted to be in the states "empty," "partial," and "overflow." Adequate integration or system testing requires the coverage of system states formed from combinations of individual object states. In both this and the previous method-oriented approach, it is not easy to predetermine the set of possible state combinations that could occur, so that insuring partial or full state coverage is facilitated by the statistical state coverage approach.

Application-Oriented Methods.

User Interface Testing. The most important specialized method in this applications area is prototyping. A typical tool allows the user to describe screens and events that cause transitions between one screen and another. In situations where user interaction is involved, the tools allow the programmer to insert dummy code in the form of stubs, which simulate instances of expected system responses.

Network Testing. Developmental testing of large systems that involve many interconnected sites and/or pieces of equipment require the use of simulators. Examples include networks in which simulators are used to model the effects of multiple machines. In the case of real-time systems, the simulators are used in place of actual machines or model a software system connected to a real machine.

Real-Time Systems Testing. In addition to simulation capabilities, real-time systems require testing techniques for monitoring critical performance requirements. The inclusion of intermediate code for monitoring intermediate states may disrupt performance and may not be feasible. Specialized machines that monitor traffic on buses, recording data in a large data repository for later analysis, partially solve the problem.

Support Tools. A variety of support tools are useful during testing and analysis, in addition to special testing tools, such as coverage analyzers.

Standards

A variety of general purpose testing and analysis standards have been developed that describe strategies and required contents for test and analysis plans. They are often discussed independently of the details of particular testing methods.

IEEE Standards. IEEE proposed and currently adopted standards include: Software Quality Assurance Plans (STD 730.1-1989), Software Test Documentation, Classifications of Errors, Faults and Failures, Reviews and Audits (STD-1028-1988), Reliability Measurement, Unit Testing, and Verification Plans. These standards are very general and describe minimal requirements in terms of sections to be included in an activity or plan. For example, the Software Test Documentation standard requires sections on test planning, design of tests, test values and expected outputs, test procedures, test reporting, test history, and test summary.

United States Government. Standards involving testing and analysis have been prepared by branches of the American armed services and other government agencies. They include Technical Review and Audits (DOD 1521), Software Development Process (DOD 2167A), Software Engineering (DOD 2168), Software Quality Evaluation (DOD-STD-268), and Software Quality Assurance Standard for FAA (FAA-STD-018).

ISO 9000. The International Standards Organization has established five standards for quality. Of these, there are three corresponding to levels of certification: 9001, 9002, and 9003. Others include standard 9000, an overview, and 9004 which describes internal approaches for establishing a quality approach. The most relevant standard for testing is 9003, which relates to quality assurance during final inspection and testing. The other two, 9001 and 9002, relate to earlier or more general aspects of quality control. Standard 9001 contains a list of 20 requirements that must be satisfied for satisfactory quality assurance. Examples of those whose details are the most relevant are: Inspection and Testing, Control of Inspecting and Testing, and Inspection and Test Status.

SEI Capability Maturity Model (*CMM*). The CMM, developed at the Software Engineering Institute, identifies five different levels of process maturity. The model is not specifically concerned with testing and

analysis, but this important aspect of development is related to its levels. At level one there is no formal process, and everything depends on the capabilities of a (hopefully) exceptional programmer. In terms of testing and analysis, this corresponds to an approach where the development team has no standards or guidelines and is simply expected to do its best. At level two, management methods are applied to monitor the process. This implies that test and analytical activities are scheduled and their activities are monitored. It involves documents that describe the tests carried out. At the third level, there is a well-defined development process. The places at which test documents are created, and the places where analysis activities, such as reviews, are carried out are identified. At the fourth level, measurement is emphasized. At this level, we keep track of the effectiveness of the methods that were used, the kinds of errors, faults and failures that they detected, the cost of method use, and the defects which methods did not detect that were found later in the software life cycle. At level five we implement an approach to improvement and optimization that depends on the data gathered during earlier phases. For example, we might have a procedure for increasing the number of reviews or eliminating an expensive testing procedure.

Management

The organization of a testing and analysis effort is related to the process used to develop software. In the typical life-cycle model, the process contains requirements, design, coding, test and validation, and maintenance stages. Planning for testing and analysis begins with a project testing and analysis plan, which is established early in the life cycle. This defines goals, allocates responsibilities, identifies needed resources, lists methods and tools to be used, and coordinates testing and analysis with project control issues such as version management and release.

In the early stages of software development, two kinds of activities take place: reviews and generation of test descriptions. They involve checking requirements and design documents, and the identification of tests designed to demonstrate functionality. Once the software is developed, code reviews take place and actual testing begins.

Testing is often divided into unit, integration, systems, and acceptance testing stages. In the first, individual modules or procedures are tested. This procedure is often carried out by the programmers who developed the software and is the area of testing for which considerable technology has been developed.

In integration testing, groups of interacting modules are tested. Several different strategies are used here, such as bottom up, top down, functional, and threads-oriented. In the bottom up method, low-level modules, which do not use other modules, are tested first. Then higher level modules, which call these, are tested, and so on. This approach requires the use of driver code for calling lower level modules. The top down approach starts with the top level modules that are not called by other modules and works its way down the call/use tree. This approach requires developing stubs, or dummy modules that stand in for untested called modules. In the functional approach, individual modules that cooperate to produce a functional effect or support a single responsibility are tested together. In the threads approach, control threads associated with typical calls on the system, starting with top level modules and threading their way through called modules to bottom level modules, are identified and tested. The goal in integration testing is to make sure that modules work together, and specific tests are chosen that have this objective, such as tests to confirm that parameters in calling and called routines match. The planning for both integration and unit testing begins during design when modules are identified and specified.

In systems testing, the entire system is tested over a full range of expected functionality. The planning for these tests occurs during requirements. Different methods are used to organize such early test planning, including matrices that show functions and requirements along one dimension and tests along the other. Tests should cover both valid and invalid data, and test for performance and other properties as well as functionality. Systems testing uses instrumentation in which code is inserted that records state conditions, so that different

kinds of systems coverage can be measured. Acceptance testing is much like system testing, except that it is performed in the intended application environment, possibly in parallel with a current system which it will replace.

Nonstatistical testing is normally used to establish basic functionality and detect faults. When a software component is operating successfully, statistical methods can be used to estimate its reliability. Statistical methods are particularly relevant for integration, systems, and acceptance testing.

Maintenance requires retesting after additions and changes are made. This involves reexamining previous tests at all levels to determine which tests should be rerun and which new tests are needed. This activity should be anticipated through proper maintenance of test documentation. Management involves organizing changes and related tests into batches, which correspond to new releases or versions.

Multiple methods are necessary for testing and analysis for several reasons. Validation of a program requires that it have required functionality, demonstrated through the use of positive methods, such as functional testing, and that it be free of common faults, demonstrated through the use of contrapositive methods, such as code inspection. Another reason for using multiple methods is the observation that different defects are often associated with different points of view. Some defects are more readily discoverable if we take an error-oriented point of view and others a fault-oriented view. An additional reason involves cost effectiveness. In general, a defect is less expensive to detect and repair if it is discovered as close as possible to its source, using methods relevant to the earlier phases of development. For example, it is better to repair a requirements defect during requirements review than program testing.

Proper management requires formal defect reporting and record keeping. Defect reports, like those described in the section on effectiveness and process improvement can be used. This information is kept as part of a test folder for a test or group of test cases. The folder identifies a review or test set to be carried out, its objectives, expected results, initialization requirements, resource requirements, and responsible and necessary personnel. The folder is also used for test version management, as tests are altered and improved. The maintenance of empirical information about the results of testing and analysis is important for managing individual projects and for process improvement, in which methods are refined, added, and deleted.

BIBLIOGRAPHY

1. E. Miller W. E. Howden *Software Testing and Validation Techniques*, Long Beach, CA, IEEE, 1981.
2. E. Yourdon *Structured Walkthroughs*, 4th ed., Englewood Cliffs, NJ: Yourdon Press, Prentice-Hall, 1989.
3. K. M. Olender L. J. Osterweil Cecil: A sequencing constraint language for automatic static analysis generation, *IEEE Trans. Softw. Eng.*, **SE-16**: 3, 1990.
4. W. E. Howden Bruce Wieand QDA: A method for systematic informal program analysis, *IEEE Trans. Softw. Eng.*, **SE-20**: 6, 1994.
5. L. J. Osterweil L. D. Fosdick Some experiences with Dave—a FORTRAN program analyzer. *Proc. 1976 National Comput. Conf.*, AFIPS, New Jersey, 1976.
6. M. Fagan Advances in Software Inspections, *IEEE Trans. Softw. Eng.*, **SE-12**: 7, 1986. W. Hetzel *The Complete Guide to Software Testing*, QED Information Sciences, Wellesley, MA, 1994.
7. W. E. Howden Functional program testing, *IEEE Trans. Softw. Eng.*, **SE-6**: 2, 1980.
8. W. E. Howden *Functional Program Testing and Analysis*, New York: McGraw-Hill, 1986.
9. D. J. Richardson L. Clarke Partition analysis: a method combining testing and verification, *IEEE Trans. Softw. Eng.*, **SE-11**: 12, 1985.
10. T. J. Ostrand M. J. Balcer The category partition method for specifying and generating functional tests, *CACM*, **31-6:** 1988.
11. B. Marick *The Craft of Software Testing*, Englewood Cliffs, NJ: Prentice-Hall, 1995.
12. R. A. DeMillo R. J. Lipton F. G. Sayward Hints on test data selection: help for the practicing programmer, *Computer*, **11**: 4 1978.

13. W. E. Howden Weak mutation testing and completeness of program test sets, *IEEE Trans. Softw. Eng.*, **SE-8**: 4, 1982.
14. W. E. Howden Reliability of the path analysis testing strategy, *IEEE Trans. Softw. Eng.*, **SE-2**: 3, 1976.
15. L. J. White E. I. Cohen A domain strategy for program testing, *IEEE Trans. Softw. Eng.*, **SE-6**: 5, 1980.
16. N. Leveson S. Cha T. Shimeall Safety verification of Ada programs using software fault trees, *IEEE Softw.*, **8**: 4, 1991.
17. R. A. Thayer M. Lipow E. C. Nelson *Software Reliability*, Amsterdam: North-Holland, 1978.
18. J. W. Duran S. C. Ntafos An evaluation of random testing, *IEEE Trans. Softw. Eng.*, **SE-10**: 4, 1984.
19. J. D. Musa A. Iannino K. Okumoto *Software Reliability*, New York: McGraw–Hill, 1990.
20. M. Z. Tsoukalas J. W. Duran S. C. Ntafos On some reliability estimation problems in random and partition testing, *Proc. Second Int. Symp. Softw. Reliab. Eng.*, Austin, TX, May, 1991.
21. J. A. Whittaker M. G. Thomason A. Markov chain model for statistical software testing, *IEEE Trans. Softw. Eng.*, **SE-20**: 10, 1994.
22. D. Voit A framework for reliability estimation, *Proc. Int. Symp. Softw. Reliab. Eng.*, IEEE, Monterey, CA, 1994.
23. W. E. Howden Comments analysis and programming errors, *IEEE Trans. Softw. Eng.*, 1990.
24. W. E. Howden G. M. Shi Linear and structural event sequence analysis, *Proc. ISSTA*, San Diego, CA, January, 1995.
25. M. R. Woodward D. Hedley M. A. Hennell Experience with path analysis and testing of programs, *IEEE Trans. Softw. Eng.*, **SE-6**: 3, 1980.
26. J. W. Laski B. Korel A data flow oriented program testing strategy, *IEEE Trans. Softw. Eng.*, **SE-9**: 3, 1983.
27. K. C. Tai Theory of fault-based predicate testing for computer programs, *IEEE Trans. Softw. Eng.*, **SE-22**: 8, 1996.
28. B. Beizer *Black Box Testing*, New York: Wiley, 1995.
29. T. J. McCabe A complexity measure, *IEEE Trans. Softw. Eng.*, **SE-2**: 4, 1976.
30. L. Clarke A. Podgurski D. Richardson S. Zeil A formal evaluation of data flow path selection criteria, *IEEE Trans. Softw. Eng.*, **SE-15**: 11, 1989.
31. E. J. Weyuker The evaluation of program-based software test data adequacy criteria, *CACM*, **31-6**: 1988.
32. W. E. Howden Confidence-based reliability and statistical coverage estimation, *Proc., 8th Int. Symp. Softw. Reliab. Eng.*, Albuquerque, NM, 1997.
33. R. Doong P. Frankl The ASTOOT approach to testing object-oriented programs, *ACM Trans. Softw. Eng. Methodology*, **3-2**: 1994.
34. R. Hamlet J. Voas Faults on its sleeve: Amplifying software reliability testing, *ISSTA*, Boston, June 1993.
35. W. Hetzel *The Complete Guide to Software Testing*, Wellesley, MA: QED Information Sciences, 1984.

WILLIAM E. HOWDEN
University of California–San Diego