A review of software testing

P DAVID COWARD

Abstract: Despite advances in formal methods of specification and improved software creation tools, there is no guarantee that the software produced meets its functional requirements. There is a need for some form of software testing. The paper introduces the aims of software testing. This is followed by a description of static and dynamic analysis, and, functional and structural testing strategies. These ideas are used to provide a taxonomy of testing techniques. Each technique is briefly described.

Keywords: software development, software testing, formal methods

Before software is handed over for use, both the commissioner and the developer want the software to be correct. Unfortunately, what is meant by 'correct' is not clear. It is often taken to mean that the program matches the specification. However, the specification itself may not be correct. Correctness is then concerned with whether the software meets user requirements. Whatever the definition of correctness there is always the need to test a system.

Testing is one of the many activities that comprise the larger complex task of software development. The need for testing arises out of an inability to guarantee that earlier tasks in the software project have been performed adequately, and attempts to assess how well these tasks have been performed.

There is no agreed definition of testing. The term is often used to describe techniques of checking software by executing it with data. A wider meaning will be adopted in this paper: testing includes *any* technique of checking software, such as symbolic execution and program proving as well as the execution of test cases with data. Checking, implies that a comparison is undertaken. The comparison is made between the output from the test and an expected output derived by the tester. The expected output is based on the specification and is derived by hand.

Two terms often associated with testing are *verification* and *validation*. Verification refers to ensuring correctness from phase to phase of the software development cycle. Validation involves checking the software against the requirements. These strategies have been termed *horizontal* and *vertical* checks. Sometimes, verification is associated with formal proofs of correctness, while validation is concerned with executing the software with test data. This paper avoids these terms and instead refers only to *testing* and *checking*, both terms being used synonymously.

Testing may be subdivided into two categories: *functional* and *nonfunctional*.

Functional testing addresses itself to whether the program produces the correct output. It may be employed when testing a new program or when testing a program that has been modified. *Regression testing* is the name given to the functional testing that follows modification. Primarily, regression testing is undertaken to determine whether the correction has altered the functions of the software that were intended to remain unchanged. There is a need for the automatic handling of regression testing. Fischer[1] describes software for determining which tests need to be rerun following a modification.

Implementing the functions required by the customer will not necessarily satisfy *all* the requirements placed upon a software system. Additional requirements, which are the subject of nonfunctional testing, involve checking that the software:

- satisfies legal obligations,
- performs within specified response times,
- is written to a particular house style,
- meets documentation standards.

The scope of this paper is limited to addressing the testing of the commissioner's functional requirements. The literature is not united about the aims of software testing. The variety of aims seem to fall into one of two camps:

- testing is concerned with finding faults in the software,
- testing is concerned with demonstrating that there are no faults in the software.

These may be viewed as an individual's attitude towards testing which may have an impact on how testing is conducted. Aiming to find faults is a destructive process, whereas aiming to demonstrate that there are no faults is constructive. Adopting the latter strategy may cause the tester to be gentle with the software, thus, giving rise to the risk of missing inherent faults. The destructive stance is perhaps more likely to uncover faults because it is more probing. Weinberg[2] suggests that programmers regard the software they produce as an extension of their ego. To be destructive in testing is therefore difficult. NASA long ago established teams of software validators separate from the software creators[3] a practice which is

Department of Computer Studies, Bristol Polytechnic, Coldharbour Lane, Frenchay, Bristol BS16 1QY, UK

Reprinted from *Information and Software Technology*, Vol. 30, No. 3, April 1988.

now widespread in large software development organizations.

There are a large number of questions about testing. How much testing should be undertaken? When should we have confidence in the software? When a fault is discovered, should we be pleased that it has been found, or dismayed that it existed? Does the discovery of a fault lead us to suspect that there are likely to be more faults? At what stage can we feel confident that all, or realistically most, of the faults have been discovered? In short, what is it that we are doing when we test software? To what extent is testing concerned with quality assurance?

Perhaps testing is about both finding faults *and* demonstrating their absence. The aim is to demonstrate the absence of faults. This is achieved by setting out to find them. These views are reconciled by establishing the notion of the 'thoroughness of testing'. Where testing has been thorough, faults found and corrected, retested with equal thoroughness, then one has established confidence in the software. If, on the other hand, there is no feel for the thoroughness of the test one has no means of establishing confidence in the results of the testing. Much work has been done to establish test metrics to assess the thoroughness of a set of tests and to develop techniques that facilitate thorough testing.

Testing strategies

There are many widely differing testing techniques. But, for all the apparent diversity they cluster or separate according to their underlying principles. There are two prominent strategy dimensions: function/structural and static/dynamic. A solely functional strategy uses only the requirements defined in the specification as the basis for testing; whereas a structural strategy is based on the detailed design. A dynamic approach executes the software and assesses the performance, while a static approach analyses the software without recourse to its execution.

Functional versus *structural testing*

A testing strategy may be based upon one of two starting points: either the specification or the software is used as the basis for testing. Starting from the specification the required functions are identified. The software is then tested to assess whether they are provided. This is known as *functional testing*. If the strategy is based on deriving test data from the structure of a system this is known as *structural testing*. Functions which are included in the software, but not required; for example, functions which relate to the access of data in a database but which are not specifically asked for by a user, are more likely to be identified by adopting a structural testing strategy in preference to a functional testing strategy.

Functional testing

Functional testing involves two main steps. First, identify the functions which the software is expected to perform. Second, create test data which will check whether these functions are performed by the software. No consideration is given to *how* the program performs these functions.

There have been significant moves towards more systematic elicitation and expression of functional requirements[4–7]. These may be expected to lead to a more systematic approach to functional testing. Rules can be constructed for the direct identification of function and data from systematic design documentation. These rules do not take account of likely fault classes. Weyuker and Ostrand[8] have suggested that the next step in the development of functional testing, is a method of formal documentation which includes a description of faults associated with each part of the design as well as the design features themselves.

Howden[9] suggests this method be taken further. He claims that it is not sufficient to identify classes of faults for parts of the design. Isolation of particular properties of each function should take place. Each property will have certain fault classes associated with it. There are many classifications of faults. One detailed classification is given by Chan[10] and is a refinement of Van Tassel's[11] classification. Chan's classification consists of 13 groups which are subdivided to produce a total of 47 categories.

Functional testing has been termed a black box approach as it treats the program as a box with its contents hidden from view. Testers submit test cases to the program based on their understanding of the intended function of the program. An important component of functional testing is an *oracle*.

An oracle is someone who can state precisely what the outcome of a program execution will be for a particular test case. Such an oracle does not always exist and, at best, only imprecise expectations are available[12]. Simulation software provides a powerful illustration of the problem of determining an oracle. No precise expectation can be determined, the most precise expectation of output that can be provided is a range of plausible values.

Structural testing

The opposite to the black box approach is the white box aproach. Here testing is based upon the detailed design rather than on the functions required of the program, hence the name structural testing.

While functional testing requires the execution of the program with test data, there are two possible scenarios for structural testing. The first scenario, and the one most commonly encountered, is to execute the program with test cases. Second, and less common, is where the functions of the program are compared with the required functions for congruence. The second of these approaches is characterized by symbolic execution and program proving.

Structural testing involving the execution of a program may require the execution of a single path through the program, or it may involve a particular level of coverage such as 100% of all statements have been executed. The notion of a minimally-thorough test has occupied researchers over the years, i.e. they have been trying to discover what is the minimum amount of testing that is required to ensure a degree of reliability. Some of these are shown below:

- All statements in the programs should be executed at least once[13].
- All branches in the program should be executed at least once[13].
- All linear code sequence and jumps (LCSAJs) in the program should be executed at least once[14]. An LCSAJ is a sequence of code ending with a transfer of control out of the linear code sequence.

Probably the most thorough set of test metrics has been specified by Miller[15] who listed 13 structure-based metrics for judging test thoroughness. Obviously, the best test is an exhaustive one where all possible paths through the program are tested. However, there are two obstacles to this goal which account for the existence of the above measures.

The first obstacle is the large number of possible paths. The number of paths is determined by the numbers of conditions and loops in the program. All combinations of the conditions must be considered and this causes a rapidly increasing number of combinations as the number of conditions increases. This is known as the combinatorial explosion of testing. Loops add to the combinatorial explosion and give rise to an excessively large number of paths. This is most acute when the number of iterations is not fixed but determined by input variables.

The second obstacle is the number of infeasible paths. An infeasible path is one which cannot be executed due to the contradiction of some of the predicates at conditional statements. Most developers, when asked, would be surprised at the existence of infeasible code in a system. However, such code can be quite extensive, for example, in a recent study of a sample of programs, which involve examining 1000 shortest paths, only 18 were found to be feasible[16].

As an example of path infeasibility consider the following block of code.

```
1  Begin
2    Readln (a);
3    If a>15
4    then
5      b:=b+1
6    else
7      c:=c+1;
8    if a<10
9    then
10     d:=d+1
11 end;
```

There are four paths through this block as follows:

Path 1 lines 1,2,3,4,5,8,11.

Path 2 1,2,3,6,7,8,9,10,11.

Path 3 1,2,3,6,7,8,11.

Path 4 1,2,3,4,5,8,9,10,11.

Path 1 can be executed so long as the value of *a* is greater than 15 after the execution of line 2.

Path 2 can be executed so long as the value of *a* is less than 10 after the execution of line 2.

Path 3 can be executed so long as the value of *a* lies in the range 10 to 15 inclusive after the execution of line 2.

Path 4 cannot be executed regardless of the value of *a* because *a* cannot be both greater than 15 and less than 10 simultaneously. Hence this path is infeasible.

Even trivial programs contain a large number of paths. Where a program contains a loop which may be executed a variable number of times the number of paths increases dramatically. A path exists for each of the following circumstances: where the loop is not executed, where the loop is executed once, where the loop is executed twice etc.

The number of paths is dependent on the value of the variable controlling the loop. This poses a problem for a structural testing strategy. How many of the variable-controlled-loop-derived paths should be covered? Miller and Paige[17] sought to tackle this problem by introducing the notion of a level-i path and have employed testing metrics which utilize this notion.

A further difficulty in achieving 100% for any metric of testing coverage is the presence of island code. This is a series of lines of code, following a transfer of control or program termination, and which is not the destination of a transfer of control from elsewhere in the program. An example of island code is a procedure that is not invoked. Island code should not exist. It is caused by an error in the invocation of a required procedure, or the failure to delete redundant code following maintenance.

Static versus *dynamic analysis*

A testing technique that does not involve the execution of the software with data is known as *static analysis*. This includes program proving, symbolic execution and anomaly analysis. Program proving involves rigorously specifying constraints on the input and output data sets for a software component such as a procedure using mathematics. The code that implements the procedure is then proved mathematically to meet its specification. Symbolic execution is a technique which executes a software system, with symbolic values for variables being used rather than the normal numerical or string values. Anomaly analysis searches the program source for anomalous features such as island code.

Dynamic analysis requires that the software be executed. It relies on the use of probes inserted into a program [18,19]. These are program statements which make calls to analysis routines that record the frequency of execution of elements of the program. As

a result the tester is able to ascertain information such as the frequency that certain branches or statements are executed and also any areas of code that have not been exercised by the test.

Dynamic analysis can act as a bridge between functional and structural testing. Initially functional testing may dictate the set of test cases. The execution of these test cases may then be monitored by dynamic analysis. The program can then be examined structurally to determine test cases which will exercise the code left idle by the previous test. This dual approach results in the program being tested for the function required and the whole of the program being exercised. The latter feature ensures that the program does not perform any function that is not required.

Taxonomy of testing techniques

It is only over the last 15 years that testing techniques have achieved importance. Consequently, there is no generally accepted testing technique taxonomy. The degree to which the techniques employ a static *versus* dynamic analysis or a functional *versus* structural strategy provides one possible basis for a simple classification of testing techniques. The following grid outlines one classification. The techniques in the grid are described later in the paper. Domain testing, described later in this section, has been included under both structural and functional strategies.

Table 1. Simple classification of testing techniques

	Structural	Functiona
Static	Symbolic execution Program proving Anomaly analysis	
Dynamic	Computation testing Domain testing Automatic path-based test data generation Mutation analysis	Random testing Domain testing Cause-effect graphing Adaptive perturbation testing

Static-structural

No execution of the software is undertaken. Assessment is made of the soundness of the software by criteria other than its run-time behaviour. The features assessed vary with the technique. For example, anomaly analysis checks for peculiar features such as the existence of island code. On the other hand, program proving, aims to demonstrate congruence between the specification and the software.

Symbolic execution

Symbolic execution, sometimes referred to as symbolic evaluation, does not execute a program in the traditional sense of the word. The traditional notion of execution requires that a selection of paths through the program is exercised by a set of test cases. In symbolic execution actual data values are replaced by symbolic values. A program executed using inputs consisting of actual data values results in the output of a series of actual values. Symbolic execution on the other hand produces a set of expressions, one expression per output variable. Symbolic evaluation occupies a middle ground of testing between testing data and program proving. There are a number of symbolic execution systems[20–23].

The most common approach to symbolic execution is to perform an analysis of the program, resulting in the creation of a flow-graph. This is a directed graph which contains decision points and the assignments associated with each branch. By traversing the flow-graph from an entry point along a particular path a list of assignment statements and branch predicates is produced.

The resulting path is represented by a series of input variables, condition predicates and assignment statements. The execution part of the approach takes place by following the path from top to bottom. During this path traverse each input variable is given a symbol in place of an actual value. Thereafter, each assignment statement is evaluated so that it is expressed in terms of symbolic values of input variables and constants.

Consider paths 1–11 through the program in Figure 1. The symbolic values of the variables and the path condition at each branch are given in the right hand columns for the evaluation of this path.

At the end of the symbolic execution of a path the output variable will be represented by expressions in terms of symbolic values of input variables and constants. The output expressions will be subject to constraints. A list of these constraints is provided by the set of symbolic representations of each condition predicate along the path. Analysis of these constraints may indicate that the path is not executable due to a contradiction. This infeasibility problem is encountered by all forms of path testing.

A major difficulty for symbolic execution is the handling of loops (or iterations). Should the loops be symbolically evaluated once, twice, a hundred times or not at all? Some symbolic executors take a pragmatic approach. For each loop three paths are constructed, each path containing one of the following: no execution of the loop, a single execution of the loop and two executions of the loop.

		Path Condition	a	b	c	d
1	Begin	–	–	–	–	
2	Read a, b, c, d	–	a	b	c	d
3	a := a+b	–	a+b	b	c	d
4	IF a>c	a+b<=c	a+b	b	c	d
5	THEN d := d+1					
6	ENDIF	a+b<=c	a+b	b	c	d
7	IF b=d	a+b<=c AND b<>d	a+b	b	c	d
8	THEN WRITE ('Success', a, d)					
9	ELSE WRITE ('Fail', a, d)	a+b<=c AND b<>d	a+b	b	c	d
10	ENDIF	a+b<=c AND b<>d	a+b	b	c	d
11	END	a+b<=c AND b<>d	a+b	b	c	d

Figure 1. Program fragment and symbolic values for a path

Partition analysis

Partition analysis uses symbolic execution to identify subdomains of the input data domain. Symbolic execution is performed on both the software and the specification. The path conditions are used to produce the subdomains, such that each subdomain is treated identically by both the program and the specification. Where a part of the input domain cannot be allocated to such a subdomain then either a structural or functional (program or specification) fault has been discovered. In the system described by Richardson[24] the specification is expressed in a manner close to program code. This is impractical. Specifications need to be written at a higher level of abstraction if this technique is to prove useful.

Program proving

The most widely reported approach to program proving is the 'inductive assertion verification' method developed by Floyd[25]. In this method assertions are placed at the beginning and end of selected procedures. Each assertion describes the function of the procedure mathematically. A procedure is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon procedure entry ensures the truth of its output assertion upon procedure exit[26].

There are many similarities between program proving and symbolic execution. Neither technique executes with actual data and both examine the source code. Program proving aims to be more rigorous in its approach. The main distinction between program proving and symbolic execution is in the area of loop handling. Program proving adopts a theoretical approach in contrast to symbolic execution. An attempt is made to produce a proof that accounts for all possible iterations of the loop. Some symbolic execution systems make the assumption that if the loop is correct when not executed, when executed just once and when executed twice, then it will be correct for any number of iterations.

Program proving is carried out as the following steps:

- Construct a program.
- Examine the program and insert mathematical assertions at the beginning and end of all procedures blocks.
- Determine whether the code between each pair of start and end assertions will achieve the end assertion given the start assertion.
- If the code achieves the end assertions then the block has been proved correct.

If the code fails to achieve the end assertion then mistakes have been made in either the program or the proof. The proof and the program should be checked to determine which of these possibilities has occurred and appropriate corrections made.

DeMillo *et al*[27] describe how theorems and proofs can never be conceived as 'correct' but rather, only 'acceptable' to a given community. This acceptability is achieved by their being examined by a wide audience who can find no fault in the proof. Confidence in the proof increases as the number of readers, finding no faults, increases. This approach has clear parallels with the confidence placed in software. The wider the audience that has used the software and found no fault the more confidence is invested in the software.

When a program has been proved correct, in the sense that it has been demonstrated that the end assertions will be achieved given the initial assertions, then the program has achieved partial correctness. To achieve total correctness it must also be shown that the block will terminate, in other words the loops will terminate[28].

Anomaly analysis

The first level of anomaly analysis is performed by the compiler to determine whether the program adheres to the language syntax. This first level of analysis is not usually considered testing. Testing is usually deemed to commence when a syntactically correct program is produced.

The second level of anomaly analysis searches for anomalies that are not outlawed by the programming language. Examples of such systems which carry out such an analysis are Dave[29], Faces[30] and Toolpack[31]. Anomalies which can be discovered by these systems include:

- The existence of (island code) unexecutable code,
- Problems concerning array bounds,
- Failure to initialize variables,
- Labels and variables which are unused,
- Jumps into and out of loops.

Some systems will even detect high complexity and departure from programming standards.

Discovery of these classes of problem is dependent on the analysis of the code. The first phase of anomaly analysis is to produce a flow-graph. This representation of the software can now be easily scanned to identify anomalies. Determining infeasible paths is not within the bounds of anomaly analysis.

Some features of anomaly analysis have been grouped under the title of *data flow analysis*. Here, emphasis is placed on a careful analysis of the flow of data. Software may be viewed as flow of data from input to output. Input values contribute to intermediate values which, in turn, determine the output values. 'It is the ordered use of data implicit in this process that is the central objective of study in data flow analysis'[32] The anomalies detected by data flow analysis are:

- Assigning values to a variable which is not used later in the program,
- Using a variable (in an expression or condition) which has not previously been assigned a value,
- (Re)assigning a variable without making use of a previously assigned value.

Data flow anomalies may arise from mistakes such as misspelling, confusion of variable names and incorrect

parameter passing. The existence of a data flow anomaly is not evidence of a fault, it merely indicates the possibility of a fault. Software that contains data flow anomalies may be less likely to satisfy the functional requirements than software which does not contain them.

The role of data flow analysis is one of a program critic drawing attention to peculiar uses of variables. These peculiarities must be checked against the programmer's intentions and, if in disagreement, the program should be corrected.

Dynamic-functional

This class of technique executes test cases. No consideration is given to the detailed design of the software. Cause-effect graphing creates test cases from the rules contained in the specification. Alternatively, test cases may be generated randomly. Domain testing creates test cases based on a decomposition of the required functions. Adaptive testing attempts to create further, more effective, test cases by modifying previous test cases. In all the approaches there is the need for an oracle to pronounce on the correctness of the output.

Domain testing

This is the least well defined of the dynamic-functional approaches. Test cases are created based on an informal classification of the requirements into domains. Either data or function may provide the basis for the domain partitioning. The test cases are executed and compared against the expectation to determine whether faults have been detected.

Random testing

Random testing produces data without reference to the code or the specification. The main software tool required is a random number generator. Duran and Ntafos[33,34] describe how estimates of the operational reliability of the software can be derived from the results of random testing.

Potentially, there are some problems for random testing. The most significant is that it may seem that there is no guarantee to complete coverage of the program. For example, when a constraint on a path is an equality e.g. A=B+5 the likelihood of satisfying this constraint by random generation seems low. Alternatively, if complete coverage is achieved then it is likely to have generated a large number of test cases. The checking of the output from the execution would require an impractical level of human effort.

Intuitively, random testing would appear to be of little practical value. Some recent studies have attempted to counter this view by randomly testing instrumented programs[33–35]. Ince and Hekmatpour record that an average branch coverage of 93% was achieved for a small set of randomly generated test cases. The key to this approach is to examine only a small subset of the test results. The subset is chosen to give a high branch coverage.

Adaptive perturbation testing

This technique is based on assessing the effectiveness of a set of test cases. The effectiveness measure is used to generate further test cases with the aim of increasing the effectiveness. Both Cooper[36] and Andrews[37] describe systems which undertake this automatically.

The cornerstone of the technique is the use of executable assertions which the software developer inserts into the software. An assertion is a statement about the reasonableness of values of variables. The aim is to maximize the number of assertion violations. An initial set of test cases are provided by the tester. These are executed and the assertion violations recorded. Each test case is now considered in turn. The single input parameter of the test case that contributes least to the assertion violation count is identified. Optimization routines are then used to find the best value to replace the discarded value such that the number of assertion violations is maximized. The test case is said to have undergone *perturbation*. This is repeated for each test case. The perturbed set of test cases are executed and the cycle is repeated until the number of violated assertions can be increased no further.

Cause-effect graphing

The strength of cause-effect graphing lies in its power to explore input combinations. The graph is a combinatorial logic network, rather like a circuit, making use of only the Boolean logical operators AND, OR and NOT. Myers[38] describes a series of steps for determining cases using cause-effect graphs as follows:

- Divide the specification into workable pieces. A workable piece might be the specification for an individual transaction. This step is necessary because a cause-effect graph for a whole system would be too unwieldy for practical use.
- Identify causes and effects. A cause is an input stimulus, e.g. a command typed in at a terminal, an effect is an output response.
- Construct a graph to link the causes and effects in a way that represents the semantics of the specification. This is the *cause-effect graph*.
- Annotate the graph to show impossible effects and impossible combinations of causes.
- Convert the graph into a limited-entry decision table. Conditions represent the causes, actions represent the effects and rules (columns) represent the test cases.

In a simple case, say with three conditions, one may be tempted to feel that the cause-effect graph is an unnecessary intermediate representation. However, Myers illustrates the creation of test cases for a specification containing 18 causes. To progress immedi-

ately to the decision table would have given 262 potential test cases. The purpose of the cause-effect graph is to identify a small number of useful test cases.

Dynamic-structural

Here the software is executed with test cases. Creation of the test cases is generally based upon an analysis of the software.

Domain and computation testing

Domain and computation testing are strategies for selecting test cases. They use the structure of the program and select paths which are used to identify domains. The assignment statements on the paths are used to consider the computations on the path. These approaches also make use of the ideas of symbolic execution.

A *path computation* is the set of algebraic expressions, one for each output variable, in terms of input variables and constants for a particular path. A *path condition* is the conjunction of constraint on the path. A path domain is the set of input values that satisfy the path condition. An empty path domain means that the path is infeasible and cannot be executed.

The class of error that results when a case follows the wrong path due to a fault in a conditional statement is termed a *domain error*. The class of error that results when a case correctly follows a path which contains faults in an assignment statement is termed a computation error.

Domain testing is based on the observation that points close to, yet satisfying boundary conditions are most sensitive to domain errors[39]. The domain testing strategy selects test data on and near the boundaries of each path domain[8,40].

Computation testing strategies focus on the detection of computation errors. Test data for which the path is sensitive to computation errors are selected by analysing the symbolic representation of the path computation[39]. Clarke and Richardson[24] list a set of guidelines for selecting test data for arithmetic and data manipulation computations.

Automatic test data generation

Use is made of automatic generation of test data when the program is to be executed and the aim is to achieve a particular level of courage indicated by a coverage metric.

It has been suggested that test data can be generated from a syntactic description of the test data expressed in, say, BNF[41]. This may seem novel as it is not usual to prepare such a syntactic description of the data, but it is a technique familiar to compiler writers[42,43]. In the case of compilers a carefully prepared data description: that of the programming language, is available. The principle may be transferable to test data generation in general.

Many automatic test data generators have used the approach of path identification and symbolic execution to aid the data generation process, for example, CASEGEN[23] and the FORTRAN testbed[44]. The system of predicates produced for a path is part-way to generating test data. If the path predicates cannot be solved due to a contradiction, then the path is infeasible. Any solution of these predicates will provide a series of data values for the input variables so providing a test case.

Repeated use of the path generation and predicate solving parts of such a system may produce a set of test cases in which one has confidence of high coverage of the program. The initial path generation will provide the highest coverage. Subsequent attempts to find feasible paths which incorporate remaining uncovered statements, branches and LCSAJs will prove increasingly difficult, some impossibly difficult.

A path-based approach which does not use symbolic execution is incorporated in the SMOTL system[45]. The system has a novel approach to minimizing the number of paths required to achieve full branch coverage.

A program that has been tested with a high coverage may still not meet its specification. This may be due to the omission in the program of one of the functions defined in the specification. Data that is generated from the specification would prove useful in determining such omissions. To achieve this automatically requires a rigorous means of specification. The increasing use of formal specification methods may provide the necessary foundations on which to build automated functional test data generators.

Mutation analysis

Mutation analysis is not concerned with creating test data, nor of demonstrating that the program is correct. It is concerned with the quality of a set of test data[46,47]. Other forms of testing use the test data to test the program. Mutation analysis uses the program to test the test data.

High quality test data will harshly exercise a program thoroughly. To provide a measure of how well the program has been exercised mutation analysis creates many, almost identical, programs. One change is made per mutant program. Each mutant program and the original program are then executed with the same set of test data. The output from the original program is then compared with the output from each mutant program in turn. If the outputs are different then that particular mutant is of little interest as the test data has discovered that there is a difference between the programs. This mutant is now *dead* and disregarded. A mutant which produced output that matches with the original is interesting. The change has not been detected by the test data, and the mutant is said to be *live*.

Once the output from all the mutants has been examined, a ratio of dead to live mutants will be available. A high proportion of live mutants indicates a poor set of test data. A further set of test data must be devised and the process repeated until the number of

live mutants is small, indicating that the program has been well tested.

A difficulty for mutation analysis occurs when a mutant program is an equivalent program to the original program. Although the mutant is textually different from the original it will always produce the same results as the original program. Mutation analysis will record this as a live mutant even though no test data can be devised to kill it. The difficulty lies in the fact that determining the state of equivalence is, in general, unsolvable and hence cannot be taken into account when assessing the ratio of live to killed mutants.

Mutation analysis relies on the notion that if the test data discovers the single change that has been made to produce the mutant program then the test data will discover more major faults in the program. Thus, if the test data has not discovered any major faults, and a high proportion of the mutants have been killed, then the program is likely to be sound.

Summary

The principal objective of software testing is to gain confidence in the software. This necessitates the discovery of both errors of omission and commission. Confidence arises from thorough testing. There are many testing techniques which aim to help achieve thorough testing.

Testing techniques can be assessed according to where along the two main testing strategy dimensions they fall. The first dimension, the functional-structural dimension, assesses the extent to which the function description in the specification, as opposed to the detailed design of the software, is used as a basis for testing. The second dimension, the static-dynamic dimension, considers the degree to which the technique executes the software and assesses its run-time behaviour, as opposed to inferring its run-time behaviour from an examination of the software. These two dimensions can be used to produce four categories of testing technique:

- static-functional
- static-structural
- dynamic-functional
- dynamic-structural

As with all classifications this one is problematic at the boundaries. Some techniques appear to belong equally well in two categories.

The aims of testing techniques range from: demonstrating correctness for all input classes (e.g. program proving), to, showing that for a particular set of test cases no faults were discovered (e.g. random testing). Debate continues as to whether correctness can be proved for life-size software and about what can be inferred when a set of test cases finds no errors. A major question facing dynamic testing techniques is whether the execution of a single case demonstrates anything more than that the software works for that particular case. This has led to work on the identification of domains leading to the assertion that a text case represents a particular domain of possible test cases.

Many of the structural techniques rely on the generation of paths through the software. These techniques are hampered by the lack of a sensible path generation strategy. There is no clear notion of what constitutes a *revealing* path worthy of investigation, as opposed to a *concealing* path which tells the tester very little.

Testers often utilize their experience of classes of faults associated with particular functions and data types to create additional test cases. To date there is no formal way of taking account of these heuristics.

Symbolic execution looks to be a promising technique. Yet, few full symbolic execution systems currently exist[48]. Of the experimental systems that have been developed none address commercial data processing software written in languages such as COBOL.

Whenever a program is executed with data values, or symbolically evaluated, the success of the testing lies in the ability to recognize that errors have occurred. Who is responsible for deeming an output correct? The notion of an oracle is used to overcome this difficulty. Whoever commissions the software is deemed capable of assessing whether the results are correct. This may be satisfactory in many situations such as commercial data processing software. However, there are instances when this is not a solution. For example, software to undertake calculations in theoretical physics may be developed precisely because the calculations could not be undertaken by hand.

One of the few pieces of empirical data on testing techniques is provided in a study by Howden[49]. The study tested six programs of various types using several different testing techniques. The results are encouraging for the use of symbolically evaluated expressions for output variables. Out of a total of 28 errors five were discovered where it would be 'possible for the incorrect variable to take on the values of the correct variable during testing on actual data, thus hiding the presence of the error.' The paper concluded that the testing strategy most likely to produce reliable software was one that made use of a variety of techniques. Over the last few years effort has been directed at construction of integrated, multitechnique software development environments.

Formal proofs, dynamic testing techniques and symbolic execution used together look likely to provide a powerful testing environment. What is necessary now is an attempt to overcome the division that has arisen between the formalists and the structuralists. The level of mathematics required by many approaches to program proving is elementary in comparison with the abilities necessary to produce the software itself. On the other hand, the formalists must resist the temptation to proclaim that their approach is not just necessary but that it is also sufficient. For the production of correct software the wider the range of testing techniques used the better the software is likely to be.

References

1 **Fischer, K F** 'A test case selection method for the validation of software maintenance modification' *Proc. COMPSAC 1977,* pp 421–426 (1977)

2 **Weinberg, G M** *The Psychology of Computer Programming* Van Nostrand Reinhold (1971)

3 **Spector, A and Gifford, D** 'Case study: the space shuttle primary computer system' *Commun. ACM* Vol 27 No 9 (1984) pp 874–900

4 **DeMarco, T** *Structured Analysis and System Specification,* Yourden Press (1981)

5 **Hayes, I** *Specification Case Studies* Prentice-Hall International (1987)

6 **Jackson, M** *Principles of Program Design,* Academic Press (1975)

7 **Jones, C B** *Systematic Software Development using VDM* Prentice-Hall International (1986)

8 **Weyuker, F J and Ostrand, T J** 'Theories of program testing and the application of revealing subdomains' (IEEE) *Trans. Software Eng.* Vol 6 No 3 (1980) pp 236–246

9 **Howden, W E** 'Errors, design properties and functional program tests' *Computer Program Testing* **(Eds Chandrasekaran, B and Radicchi, S)** North-Holland, (1981)

10 **Chan, J** *Program debugging methodology* M Phil Thesis, Leicester Polytechnic (1979)

11 **Van Tassel, D** *Program Style, Design, Efficiency, Debugging and Testing* Prentice-Hall (1978)

12 **Weyuker, E J** 'On testing non-testable programs' *The Comput. J.* Vol 25 No 4 (1982) pp 465–470

13 **Miller, J C and Maloney, C J** Systematic mistake analysis of digital computer programs, *Commun. ACM* pp 58–63 Vol 6 (1963)

14 **Woodward, M R, Hedley, D and Hennell, M A** 'Experience with path analysis and testing of programs' (IEEE) *Trans. Software Eng.* Vol 6 No 6 (1980) pp 278–285

15 **Miller, E F** 'Software quality assurance' *Presentation* London, UK (14–15 May 1984)

16 **Hedley, D and Hennell, M A** 'The cause and effects of infeasible paths in computer programs' *Proc. Eight Int. Conf. Software Eng.* (1985)

17 **Miller, E F and Paige, M R** Automatic generation of software tescases, *Proc. Eurocomp Conf.* pp 1–12 (1974)

18 **Knuth, D E and Stevenson, F R** 'Optimal measurement points for program frequency count' *BIT* Vol 13 (1973)

19 **Paige, M R and Benson, J P** 'The use of software probes in testing FORTRAN programs' *Computer* pp 40–47 (July 1974)

20 **Boyer, R S, Elpas, B and Levit, K N** 'SELECT – a formal system for testing and debugging programs by symbolic execution' *Proc. Int. Conf. Reliable Software* pp 234–244 (1975)

21 **Clarke, L A** 'A system to generate test data and symbolically execute programs' (IEEE) *Trans. Software Eng.* Vol 2 No 3 (1976) pp 215–222

22 **King, J C** Symbolic execution and program testing, *Commun. ACM* Vol 19 No 7 (1976) pp 385–394

23 **Ramamoothy, C V, Ho, S F and Chen, W J** 'On the automated generation of program test data' (IEEE) *Trans. Software Eng.* Vol 2 No 4 (1976) pp 293–300

24 **Richardson, D J and Clarke L A** 'A partition analysis method to increase program reliability' *Proc. Fifth Int. Conf. Software Eng.* pp 244–253 (1981)

25 **Floyd, R W** 'Assigning meaning to programs' *Proc. of the Symposia in Applied Mathematics* Vol 19 pp 19–32 (1967)

26 **Hantler, S L and King, J C** 'An introduction to proving the correctness of programs' *Computing Surveys* Vol 18 No 3 (1976) pp 331–353

27 **Demillo, R A, Lipton, R J and Perlis, A J** 'Social processes and proofs of theorems and programs' *Commun. ACM* Vol 22 No 5 (1979) pp 271–280

28 **Elpas, B, Levitt, K N, Waldinger, R J and Wakemann, A** 'An assessment of techniques for proving program correctness' *Computing Surveys* Vol 4 No 2 (1972) pp 97–147

29 **Osterweil, L J and Fosdick, L D** 'Some experience with DAVE- a FORTRAN program analyser' *Proc. AFIPS Conf.* pp 909–915 (1976)

30 **Ramamoorthy, C V and Ho, S F** 'FORTRAN automatic code evaluation system' *Rep. M-466* Electron. Resl Lab, University California, Berkeley, CA, USA (August 1974)

31 **Osterweil, L J** 'TOOLPACK – An experimental software development environment research project (IEEE) *Trans. Software Eng.* Vol 9 No 6 (1983) pp 673–685

32 **Fosdik, L D and Osterwell, L J** 'Data flow analysis in software reliability' *Computing Surveys* Vol 8 No 3 (1976) pp 305–330

33 **Duran, J W and Ntafos, S C** 'A report on random testing' *Proc. Fifth Int. Conf. Software Eng.* pp 179–183 (1981)

34 **Duran, J W and Ntafos, S C** 'An evaluation of random testing' (IEEE) *Trans. Software Eng.* Vol 10 No 4 (1984) pp 438–444

35 **Ince, D C and Hekmatpour, S** 'An evaluation of some black-box testing methods' *Technical report No 84/7* Computer Discipline, Faculty of Mathematics, Open University, Milton Keynes,UK (1984)

36 **Cooper, D W** 'Adaptive testing' *Proc. Second Int. Conf. Software Eng.* pp 223–226 (1976)

37 **Andrews, D and Benson, J P** 'An automated program testing methodology and its implementation' *Proc. Fifth Int. Conf. Software Eng.* pp 254–261 (1981)

38 **Myers, G J** *The Art of Software Testing* John Wiley (1979)

39 **Clarke, L A and Richardson, D J** 'The application of error-sensitive testing strategies to debugging' *ACM SIGplan Notices* Vol 18 No 8 (1983) pp 45–52

40 **White, L J and Cohen, E I** 'A domain strategy for computer program testing' (IEEE) *Trans. Software Eng.* Vol 6 No 3 (1980) pp 247–257

41 **Ince, D C** The automatic generation of test data, *The Comput. J.* Vol 30 No 1 (1987) pp 63–69

42 **Bazzichi, F and Spadafora, I** 'An automatic generator for compiler testing' (IEEE) *Trans. Software Eng.* Vol 8 No 4 (1982) pp 343–353

43 **Payne, A J** 'A formalized technique for expressing compiler exercisers', *SIGplan Notices* Vol 13 No 1 (1978) pp 59–69

44 **Hedley, D** *Automatic test data generation and related topics* PhD Thesis, Liverpool University (1981)

45 **Bicevskis, J, Borzovs, J, Straujums, U, Zarins, A and Miller, E F** 'SMOTL – a system to construct samples for data processing program debugging' (IEEE) *Trans. Software Eng.* Vol 5 No 1 (1979) pp 60–66

46 **Budd, T A and Lipton, R J** 'Mutation analysis of decision table programs' *Proc. Conf. Information Science and Systems* pp 346–349 (1978)

47 **Budd, T A, Demillo, R A, Lipton, R J and Sayward, F G** 'Theoretical and empirical studies on using program mutation to test the functional correctness of programs' *Proc. ACM Symp. Principles of Prog. Lang.* pp 220–222 (1980)

48 **Coward, P D** 'Symbolic execution systems – a review' *The Software Eng. J.* (To appear)

49 **Howden, W** 'An evaluation of the effectiveness of symbolic testing' *Software Pract. Exper.* Vol 8 (1978) pp 381–397 □