Software Quality, Software Process, and Software Testing

Dick Hamlet

Portland State University
Department of Computer Science
Center for Software Quality Research
PO Box 751
Portland, OR 97207
Internet: hamlet@cs.pdx.edu
Phone: (503) 725-3216
FAX: (503) 725-3211

September 22, 1994

Abstract

Software testing should play a major role in the definition and improvement of software development processes, because testing is the most precise, most easily measured and controlled part of the software lifecycle. However, unless testing goals are clearly related to true measurements of software quality, the *testing* may appear to improve, but the *software* will not. Much of current testing theory and practice is built on wishful thinking. In this paper, the state of the testing art, and the theory describing it, is critically examined. It is suggested that only a probabilistic theory, similar to reliability theory, but without its deficiencies, can describe the relationship between test measurements and product quality. The beginnings of a new theory of "dependability" are sketched.

Keywords: software development process, assessing software quality, coverage testing, finding failures, reliability, dependability, trustworthiness

1 Introduction

Renewed interest in quality software in the 1990s stems from attention to the development process. In the United States, the Software Engineering Institute (SEI) has proposed a model of organizational structure capable of producing good software. In Europe, the International Standards Organization ISO-9000 standards address similar concerns. Schemes like "total quality management" (TQM) extend to the larger organization in which software development is embedded. Software development is often in a sorry state, certainly in need of attention to the procedures being used. However, the "process" movement inherits from software engineering an emphasis on subjective, management-oriented methods. The implication is that software development is a mysterious art, and only careful control of its practicioners can avert disaster. Software engineering suggested ways to decompose development into manageable (literally!) stages; "process" focuses on the procedural details of each stage and how to define, measure, and modify them.

But a laudable interest in the development process *per se* has the unfortunate side effect of downplaying its technical aspects. Organization and systematic, carefully monitored procedures are usually a part of successful engineering, but a relatively minor part. The essence of engineering is scientific support for its methods. All the organization in the world will not save a process based on an erroneous understanding of reality; in fact, excellent organization can have the pernicious effect of shoring up methods that should be scrapped. The only real measure of quality software is a "product" measure. It is the software that counts, not its pedigree. Of course, attention to the process can be helpful in producing a good product, but only if there are solid, demonstrable connections between what is done and what results from doing it. To carry out procedures for their own sake – for example, because they can be easily monitored and adjusted – is to mistake form for substance.

Proponents of the process viewpoint argue that the link with product is established by feedback from failures. They say that a released software deficiency will be traced to its procedural source, and an improved process will not again allow that particular problem to occur. But this view ignores the complexity of the process. Consider the following example:

> An operating system crashes. Analysis of the failure reveals that two successive system calls are at fault: the first call established an internal system state in which the second call's normal operation brought the system down. Further analysis shows that this combination was never imagined during development. It was not mentioned in the requirements, never specified, not considered in design or implementation, and although the two calls were successfully tested in isolation, the combination was not tested.

The example is not far-fetched. How can the process that led to the release of the faulty operating system be corrected? It is probably implicit in requirements for system calls that they may be used in any sequence, although not all sequences are useful. So there is nothing to correct in the requirements phase. The specification for each call fails to mention the other, but there is nothing wrong with that if they are unrelated. Probably the fault is in

design, where the strange internal state was invented, or in implementation, where there was too much freedom in representing the state. But it is not a process correction to say "don't do that!" – only hindsight identifies what not to do. It won't do to ask the designers to systematically consider system-call pairs; the next failure may be the result of a sequence of three calls, or hundreds of calls. We could ask for more extensive inspection of the design and implementation, but how much is enough to catch everything that was not imagined? Finally, what might have been done in testing? Functional testing would not expose the problem, and neither would simple coverage testing. Def-use testing might have found it, and so might mutation testing, but there is no guarantee, if the peculiar state is not created on most instances of the first call. Random testing has only a small chance of trying the failing combination. (These methods will be considered in detail below.) More testing would improve the chances of detection, but again, how much is enough?

The example was invented to ridicule the simplistic idea that product failures always produce meaningful feedback to the development process. But it also shows another important point. In every phase *except* testing, the possible corrections are subjective and unsystematic. People could be advised to "do better" but it is not clear just what that means. In the testing phase, the options were specific and technical. Some systematic methods are better than others in detecting the problem; using more tests should help; etc. Testing has less need of hindsight.

The lesson that testing is a less subjective, more precise part of development than the other phases is not lost on "process" advocates, who may choose to begin work there. Caution is still in order: in the example, a particular kind of failure might be prevented by testing. But how many kinds of failures are there, and how frequent is each kind? In development no one has the luxury of a particular situation to analyze – all field failures lie in the future. Might it not be that corrective feedback is always one step behind, and the supply of unique, new problems nearly inexhaustible? When precise techniques are available, it increases the danger of confusing procedure with results.

This paper is about the scientific, technical basis for software testing. It assesses the state of the testing art in the context of improving quality through the development process. Research is beginning to ask the right questions, and there are promising new results. But indications are that the fundamental role of testing must change from defect-discovery to assurance of quality.

1.1 Software Quality

Software quality is a many-faceted idea, and many attributes of "good" software are subjective. It must be "user-friendly" and "maintainable," for example. From the developer's point of view, perhaps its most important characteristic is rapid development – time to market is a critical factor in remaining in the software business. The fundamental measure of quality is absence of defects. Quality software does not fail. Failure can take the form of a "crash" after which the software cannot be used without some kind of drastic restart (and often a loss of information or invested time); failure can be wrong answers delivered with all the

trappings of computer authority (and thus the more dangerous); and failure can be in the performance dimension – the software works, but too slowly to be useful.

In this paper we identify software quality with the absence of failure. However, we want an engineering measure, not the binary ideal of "correct"/"not correct." Whether or not one believes that it is possible to create "zero-defect" software, to *demonstrate* correctness is impractical. Although proof methods might in principle demonstrate correctness, they have been unable to do so in practice. Part of the reason is that theorem provers are too inefficient and too hard to use; a deeper reason is that the formal specifications required for verification are at least as difficult to create, and as error-prone, as programs. The alternative to formal verification is testing, but tests are only samples of software's behavior, and the best we can hope for is that they establish some kind of statistical confidence in quality.

There is no standard term for good software in the sense of "unlikely to fail." "Reliable" has a related but different technical meaning[1] in engineering (see Section 4). Parnas has used the term "trustworthy" [PvSK90], but he includes the severity of failure: trustworthy software is unlikely to have *catastrophic* failures. What is or is not a catastrophe is sometimes subjective, and may change over time. Furthermore, catastrophe is only easy to recognize after the fact. Here we will be concerned with prediction of unspecified future events, whose severity is unknown. We will use the term "dependable" for the intuitive idea "unlikely to fail" (see Section 6).

1.2 Software Process

The emphasis in this paper will be on software testing as the phase of development easiest to technically relate to software dependability. When there is an objective, quantitative relationship between product quality (here, dependability) and process activities (here, testing methods), the feedback model of process improvement works. When the quality is inadequate, the methods can be changed (for example by devoting more resources to testing) to improve it. It cannot be emphasized too strongly that the feedback cycle must extend all the way to actual measurements of quality in the field. It is all too easy to guess at what is causing a problem, change something in the development process and measure the change *in its own terms*, and conclude that the process is improved. For example, suppose that the operating-system failure above is one of many, and that the testing plan includes only functional and basic structural coverage testing. To guess that more testing is needed, and to (say) double the time spent on testing (perhaps doubling the number of test cases), will probably have *no* effect on these failures, because these methods do not address the problem. But if measurement stops with counting test cases, the process is "improved" while the product is not.

There is no reason why any proposed development method or technique cannot be held to the same standard as testing methods. That is, one could demand that a quantitative relationship be demonstrated between any proposal and product quality. For example, the

[1] Two other senses of "reliable" occur in the testing literature in theoretical papers by Goodenough and Gerhart [GG75] and Howden [How76]. Both notions have technical flaws and are not in current use.

introduction of formal specifications methods is thought by some to be a good idea. This could be established by measuring dependability of software developed in this way. At first, the argument for any method has to be theoretical. It must be shown that the characteristics of the method make a dependability improvement plausible. In unusual circumstances, it may be possible to perform a real experiment to demonstrate the improvement. But in most cases the theoretical arguments will be the only ones available. Once a method is in place in the software process, however, experiments are more easily carried out. It is not difficult to measure software quality using field reports of failure. Any method can be improved by devoting more time to it, providing better training or tools for its users, etc. If the method improves, but quality (dependability) does not, it is a strong indication that the theoretical arguments supporting the method were wrong.

1.3 Formal Methods and Verification

Advocates of formal, mathematical methods in software development[2] argue persuasively that the root cause of poor software quality is a loss of intellectual control. The remedy suggested is better mental tools such as precise notations and formal reasoning. Formal methods are proposed for two distinct roles in software development:

"Notation" role Abstraction and formal notation are powerful aids to precise thinking. By expressing a software specification or design in a concise mathematical form, the human developer is led to be more precise and accurate, to think through issues and decisions that might otherwise have gone unnoticed. The Z specification language is currently a popular vehicle for the notation role. The formalism is viewed as an end in itself. Even if the specification were devised and then immediately discarded (this is not advocated, although it has happened where a contract mandated formal specification), the benefits of better understanding and clearer thinking about the software should accrue.

"Proof" role Formal systems based on symbolic logic may be used to go beyond the notation role to include formal reasoning. Once software properties have been captured formally, the formal description can be used to derive properties that software meeting the description necessarily will possess. The proof role corresponds to the classical purpose of mathematical formalism in the sciences, which is to model reality abstractly, so that abstract reasoning may derive its properties. The power of formalism in the proof role is that theorems proved about the formal abstraction may be difficult and surprising, expressing properties that were not suspected, and could not be established without abstraction. In establishing formal properties, a proof-oriented formalism often makes use of a mechanical theorem prover.

The distinction between the notation and proof roles lies in the extent to which the mathematical notation is used to reason about software, and there is evidently a continuum

[2] The Formal Methodists?

of possibilities ranging from no proofs at all to an effort in which proofs are dominant. The formalism used may influence where on this continuum the developer can move; for example, the Z language is not well adapted to proofs. The cleanroom methodology [CM90] has been used mostly in the notation role, but its formalism is useful for intuitive (not mechanical) proofs. Although serious use of formalism is only beginning to be seen in practice, the notational role is the more common [GCR93]. The proof role has been limited to applications with security constraints, usually in military systems.

The arguments advanced for formal methods today are given largely in process terms. It is claimed that these methods are obvious aids to human effort in an admittedly difficult task. But to evaluate them in product terms requires a different kind of argument. Granted that formalism allows people to do a better job, is it possible to quantify the improvement that will result, say in software dependability? Without this quantification, it is impossible to compare different formal methods, or to compare use of a formal method with testing. It is not so much that arguments for product quality resulting from the use of formal methods *cannot* be given, as that they have not been attempted[3]. The subjective nature of development phases other than testing makes it much harder to produce and to evaluate theoretical arguments for their efficacy. But it is wishful thinking to employ these methods and ignore their quantitative effects (if any).

1.4 Software Testing

Because testing is a precise[4], quantitative activity, it must answer to more than the subjective criteria used to evaluate other phases of software development. However, its very precision is a danger. In testing we know what we are doing, and we can measure it. But we do not necessarily know – and we might not measure – what its *effect* will be.

The common wisdom in testing (and it has become common wisdom largely because of the "process" movement) is usually credited to Glenford Myers [Mye79]:

> The purpose of testing is to find failures.

Myers' insight is certainly a profound improvement on the earlier idea that the purpose of testing is to show that software does not fail. It is all too easy for a slapdash test to succeed even on the worst software, and with the goal achieved, there is strong disincentive to work harder.

Myers' suggestion is in agreement with the testing goal for other engineered products, and certainly in line with the testing of computer hardware. However, the analogy between mass-produced physical objects (each slightly different) subjected to environmental stress, and software (each copy truly identical) that does not wear or get brittle, etc., is a poor one.

[3] An illustrative attempt at the kind of argument required has been given for the cleanroom methodology [HV93].

[4] The *importance* of a development phase is unrelated to its precision. Without question, requirements analysis is the most important phase, and it will forever remain the most imprecise, since it must translate abstract ideas in the application domain to concrete ideas in the software domain.

In hardware testing we know a good deal about the possible failure modes of the device, and tests are designed to expose them. When the tests fail, the part is flawed and is discarded. When the tests succeed, we believe the part is up to spec (in manufacture – its design has not been considered) because we believe that the failure modes tested are an exhaustive list. In software the failure modes are unlimited and unknown. When a test fails, we know something is wrong – that is the basis of Myers' insight. But when all tests succeed, we know only that some failures are precluded, not exactly what these failures are, and nothing about other failure possibilities.

Because no software testing method (or combination of methods) is effective at exposing all possible failures, a very unsatisfactory situation arises. Eventually, all the defects a testing method can find will have been found, so the method's usefulness (for exposing failure) is at an end. But the "tested" software is still of unknown quality. Some defects have been removed; but what defects remain? The analogy to fishing a lake is apt: when you catch no fish, it doesn't necessarily mean there are none in the lake. A program's testing space is bigger than a lake, its defects are as elusive as fish, and testing methods are no better than fishermen's schemes. The fishing analogy can be made quantitative. A deep lake 150 km in diameter (roughly the size of one of the Great Lakes) might contain $10^{11}\mathrm{m}^3$ of water, and trolling it for a day, a fisherman might probe about $10^4\mathrm{m}^3$ (assuming the lure is attractive in a 0.2 m^2 crossection), or about fraction 10^{-7} of the space. A program with a pair of integer inputs running on a 32-bit machine has about 2^{64} possible input values, and testing it for a day at one test/sec is about 3×10^4 tests, or fraction 10^{-15} of the possibilities. It is as ideal to imagine that each instant of fishing time covers a different part of the lake, as to suppose that every test will be truly different. Just as fish do not always bite when the bait comes near, so bugs do not always reveal themselves when tests encounter faulty code. All in all, it seems that a fisherman predicting no fish in Lake Michigan after getting skunked is far more likely to be right, than is a software tester predicting no bugs in a trivial program that tested without failure.

Practical testing methods (like the notions of fishing guides) are based on experience in uncovering software failures (or finding fish). But it is fallacious to believe that these methods have significance when they do not do what they were designed to do. (That is, when they find nothing, it is fallacious to conclude that there is nothing to find.) For assessing the quality of software based on successful tests, different methods are required. Everyone who has done large-scale testing has an intuition that there is a connection between practical testing methods and software quality, but until that connection is supported by theoretical arguments and empirical studies, it remains a "fish story."

There are reasons to test with the intent of finding failures (see Section 3), although obtaining dependable software is not currently one of them. There is even some reason to hope that a connection can be established between failure-finding and the quality of the resulting software. But for now, Myers' advice deals with the process, not the product. Failure detection is certainly a measure of the effort put into testing, and if there is an independent control on defects – for example, a rigorous inspection process – test failures can be a measure of how well that control is working. But since a reasonable development

process corrects defects that are found, testing for failure applies not to the final released software product, but to intermediate, discarded versions that are not released. As such, this testing does not meet the standard for true process quality.

2 Testing Background and Terminology

A *test* is a single value of program input, which enables a single execution of the program. A *testset*[5] is a finite collection of tests. These definitions implicitly assume a simple programming context, which is not very realistic, but which simplifies the discussion. This context is that of a "batch" program with a pure-function semantics: the program is given a single input, it computes a single result and terminates. The result on another input in no way depends on prior calculations. Thus a testset puts this program through its paces, and the order of the tests is immaterial.

In reality, programs may have complex input tuples, and produce similar outputs. But we can imagine coding each of these into a single value, so that the simplification is not a transgression in principle. Interactive programs that accept input a bit at a time and respond to each bit, programs that read and write permanent data, and real-time programs, do not fit this simple model. However, it is possible to treat these more complex programs as if they used testsets, at the cost of some artificiality. For example, an interactive program can be thought of as having testsets whose members (single tests) are *sequences* of inputs. Because testing theory often has negative results to state, these results can be presented in the simple case, and can be expected to carry over to more complex cases.

Each program has a specification that is an input-output relation. That is, the specification S is a set of ordered input-output pairs describing allowed behavior. A program P *meets* its specification for input x iff: if $x \in dom(S)$ then on input x, P produces output y such that $(x, y) \in S$. A program meets its specification (everywhere) iff it meets it on all inputs. Note that where $x \notin dom(S)$, that is, when an input does not occur as any first element in the specification, the program may do anything it likes, including fail to terminate, yet still meet the specification. Thus S defines the required input domain as well as behavior on that domain.

A program P with specification S *fails* on input x iff P does not meet S at x. A program fails, if it fails on any input. When a program fails, the situation, and loosely the input responsible, is called a *failure*. The opposite of fails is *succeeds*; the opposite of a failure is a *success*.

Programmers and testers are much concerned with "bugs" (or "defects," or "errors"). The idea of "bug" in unlike the precise technical notion of "failure" because a bug intuitively is a piece of erroneous program code, while a failure is an unwanted execution result. The technical term for "bug" etc., is *fault*, intuitively the textual program element that is

[5] At a research conference, one of the speakers made substantial use of "testset." In the question period following, a member of the audience commented, "I've just realized that 'testset' is a palindrome!" This insight was applauded heartily.

responsible for one or more failures. However appealing and necessary this intuitive idea may be, it has proved extremely difficult to define precisely. The difficulty is that faults have no unique characterization. In practice, software fails for some testset, and is changed so that it succeeds on that testset. The assumption is made that the change does not introduce any new failures (an assumption false in general). The "fault" is then defined by the "fix," and is characterized, e.g., "wrong expression in an assignment" by what was changed. But the change is by no means unique. Literally an infinity of other changes (including those that differ only by extraneous statements) would have produced exactly the same effect. So "the fault" is not a precise idea. Nevertheless, the terminology is useful and solidly entrenched, and we will use it to refer (imprecisely) to textual elements that lead to failures.

These definitions of failure and fault correspond to an IEEE glossary[6].

2.1 The Oracle Problem

Evidently the most important aspect of any testing situation is the determination of success or failure. But in practice, the process is decidedly error-prone. If a program fails to complete its calculation in an obvious way (for example, it is aborted with a message from the run-time system), then it will likely be seen to have failed. But for elaborate output displays, specified only by an imprecise description in natural language (a very common real situation), a human being may well fail to notice a subtle failure. In one study, 40% of the test failures went unnoticed [BS87]. To do better, the process of failure detection must be automated.

An *oracle* for specification S is a binary predicate J such that $J(x, y)$ holds iff: either $x \notin dom(S)$ or $(x, y) \in S$. (That is, J is a natural extension of the characteristic function of S.) If there is an algorithm for computing J then the oracle is called *effective*. Thus, given a program and a test point, an oracle can be used to decide if the result the program gives is or is not correct (the program does or does not meet its specification at this point). For an effective oracle this decision can be automated[7].

Testing theory, being concerned with the choice of tests and testing methods, usually ignores the oracle problem. It is typically assumed that an oracle exists, and the theoretician then glibly talks about success and failure, while in practice there is no oracle but imperfect human judgement. The theoretician justifies this lack of concern in two ways: (1) the problem is one of specification, not testing – proper specifications should have effective oracles; and, (2) all testing methods suffer equally from the absence of an oracle. Without

[6] *Error* is also a standard IEEE term, referring to something that is wrong with a program's internal state – a fault in the process of becoming a failure. Although this is an increasingly important concept in testing (see section 6.3), "error" will seldom be used here in its IEEE technical sense. The literature is littered with uses that almost always mean "fault," e.g., "error-based testing."

[7] It is sometimes stated that an effective oracle requires (or is) an executable specification. Certainly being able to execute the specification (if the specification domain is known and termination is guaranteed) does yield an effective oracle, since the specification value can be compared with the program output. But there may be other ways to make the decision that do not involve executing the specification. For example, it is possible to check the results of a division by multiplying quotient and dividend; one need not know how to divide to tell if division is correct.

an oracle, testing cannot be done, but whatever is available will work for any method equally well (or equally badly). Point (1) is well taken, and specification research has accepted the responsibility in the strong sense that specification methods seek effective oracles that can be mechanically obtained for any example specification [GHM81, AH92]. But justification (2) contains a subtle flaw: it assumes all testing involves the same size testsets. If method X requires far more tests than method Y, then with an inefficient oracle, X may be intractable while Y is not. Of course, more tests always cost more in machine execution time, but people time is the more important in practice today, and today's oracles are usually people. The size of testsets is an issue in random testing (see section 4.2) and in mutation testing (see section 3.2).

2.2 Unit Testing vs. System Test

The definitions above are framed to apply to complete software systems, programs that can be executed, that have input-output specifications. But testing a large piece of software is a formidable task: the number of possible behaviors may be very large, and it may not be easy to select tests. The tester intuitively feels that the problem has no practical solution, and that whatever tests are conducted, they will barely scratch the surface, and leave much untried. It is natural to want to decompose the testing problem into more manageable, understandable units. Since software itself is composed of units (subroutines, modules, etc.), it is natural to think of testing these. The practical difficulty with "unit testing" is precisely that small units are *not* directly executable, and they may have specifications even more vague than the typical poor one for the whole program.

In principle, the specification problem is also decomposed and simplified by considering modules. But in practice, designers may be very slipshod about unit specifications, relegating them to a comment in the the code like, "update control block." Of course, the code could not have been written without knowledge of the detailed format of this "control block," and without a clear statement of what it means to "update" it. But this information is likely to be distributed across many design documents, and to reside partly in the heads of designers and coders. The tester may be at a complete loss for a module oracle, without which testing is impossible.

Executing a module that expects its inputs from other modules is not a problem in principle. It is solved by writing a "testing harness" – a main program that accepts inputs, sends them to the unit to be tested, and keeps track of the results returned. Such harnesses can be automatically created from the unit's source syntax, using compiler techniques. Missing modules needed by the unit under test are a more difficult problem. If those modules are written and linked in, then a small "unit" is no longer being tested, but rather a "subsystem" of some kind, and its properties do not always reflect properties of its parts. For example, a test may succeed despite a fault in one module, because it calls another compensating module. On the other hand, if the called modules are replaced by "stubs" – dummy routines that do nothing themselves – the unit under test may not have the proper environment, and may itself do nothing significant.

Whatever the practical problems of unit testing, the idea harbors a much more serious difficulty in principle: when all unit tests succeed, and the units are linked together, it does not follow that overall system tests will succeed, or that properties tested for units will still be assured. This failure of unit testing to "compose" will be further discussed in section 3.3.

3 Testing to Detect Failures

If Myers' advice to test for failure was needed by some segment of the testing community, it certainly was not news to those who devise systematic testing methods. These methods look for failures, and the mindset of their inventors is accurately described by the rhetorical question: "How can a tester be prevented from thinking everything is OK, when it is not?" The idea behind systematic testing is coverage. A deficient test, perhaps conducted by someone who really did not want to discover any problems with the software being tested, might succeed but only because it doesn't probe very deeply. Coverage measures this probing. A typical systematic testing method, applied with deficient test data, might issue the generic message:

> "Testing succeeded, but it did not cover <list of elements missed>."

Systematic testing methods differ in the kind of coverage they require. The major division is between "functional" and "structural" methods.

Functional testing is also called "blackbox" testing, or "specification-based" testing. The coverage is by functions (cases) of what the software is supposed to do. In a command-driven system under test, different commands could be taken to be functions; a test would not achieve functional coverage unless it had used every command. Additional functional classes could be defined by considering command-parameter classes, e.g., "long" or "short" for a string, positive/negative/zero for an integer, 1st/2nd/3rd/4th quadrant for an angle, etc. A command with one argument that is an angle would then define four functional classes to be covered, one for each quadrant.

Functional testing is the primary systematic technique, at the heart of any reasonable plan for finding software failures. Without it, the tester has no idea what software will do in the very cases that it is supposed to handle, cases that are going to be tried by its users.

Structural testing is also called "program-based" or "code-based" testing, and also "clear-box" or "white-box" testing[8]. The coverage is by syntactic parts of the program begin tested. The most common kind of structural coverage is "statement testing," which requires that test data force the execution of each and every program statement.

Structural testing requires no knowledge of what a program is supposed to do (except in determining success or failure of each test – see section 2.1), and thus its coverage requirement can be automated. Tools for measuring structural coverage are often efficient and easy to construct. For example, many operating systems have a "profile" utility that counts

[8] "White-box" shows historical ignorance, because the opposite of a circuit that could not be examined in a black crinkle box was a circuit visible through a plexiglass case. Thus "clear-box" is the better term.

execution frequencies by statement, using instrumentation inserted in the source program. Deficiencies in statement coverage show up as profiles containing statements executed zero times. Without statement-coverage testing, the tester has no idea what will happen when the statements that were not tested are executed, as they probably will be when the program is used.

Functional and structural testing intuitively complement each other: structural testing exercises the code that is present, and functional testing covers what should be there. These methods intuitively uncover failures because bugs have a functional and/or structural location, so by systematically exploring these locations, the bugs may be found. Most test plans use a combination of methods. A particularly plausible scheme [Mar91] is to devise tests for functional coverage, and then measure the structural coverage of those same tests, to judge how well they apply to the code being tested. When parts of the code are not covered, the tester should return to the functional cases to identify omitted functions, and repeat the process.

Suppose that a test plan specifies complete coverage (functional or structural, or both), and that it is successfully carried out. How could a program bug nevertheless escape detection? This question is central to understanding in detail why systematic testing cannot establish that a program works. Actual test cases are finite in number (and for practical testing, the number is severely limited by time and resource constraints). Any test can therefore cover at most a finite number of things. Thus the list of functions to be covered (for example), must have a coarse granularity. In the example of a command with an angle argument, it could happen that the program fails when the angle is 1 ± 0.00001 radians. This is in the 1st quadrant, but the tester may not have chosen such points in "covering" the first quadrant. There are an infinity of points (or at least impractically many in the digital quantization) to consider, and it is impossible to try them all. The situation of "covering" an infinity with a small finite number of tests is general in systematic testing. For structural statement testing, there are but a finite number of statements in any program. But each statement may be executed with varying values for the quantities it manipulates, and true coverage would have to try all these values. Statement coverage does not do so, which is its flaw.

3.1 Functional Testing

The bulk of the creative effort in functional testing goes into devising the tests, or put another way, into defining the functional classes that are to be covered, and their representatives. Functional testing fits best at the system level, because software requirements and specifications are most often at that level. User documentation (especially a user's manual with examples) is a good source for a functional breakdown. With the test points in hand, the tasks of executing them, judging the answers, and keeping track of what has been done, are relatively straightforward. Tool support for the bookkeeping is helpful, indeed essential for a large system [BHO89].

At the unit level, functional testing is also of value, but more difficult to carry out,

because unit specifications may be poor or non-existent, and there may be a dearth of intuitive functional classes for a unit. It is common to rely on implicit understanding, or volatile design knowledge that is not carefully recorded, in defining what code units must do. Furthermore, units may have their functionality so intertwined with other units that it is impossible to test them separately. Some units perform a function that does not have any intuitive breakdown into cases; for example, a subroutine computing e^x for argument value x seems inadequately tested using the classes $\{x|x < 0\}$, $\{0\}$, and $\{x|x > 0\}$, but what other classes come to mind?[9]

Although functional testing is defined to make no use of the program structure, in practice there is good reason to add some design information to the specification when defining functional classes. Every program has important internal data structures, and arbitrary restrictions on its operation imposed by implementation limitations; these lead to natural functional classes for testing. For example, a program that uses buffers should be probed with the buffer empty, just full, and over-full. What is being covered by such tests are really design elements, which fall between the functional and the structural. Another example is a program that uses a hash table: the distinction of collision/no-collision defines a pseudo-functional class. Marick [Mar91] calls the kind of testing that includes boundary cases from design "broken-box" testing; it has long been treated as an aspect of functional testing [GG75].

3.2 Structural Testing

Structural-testing methods are conveniently divided into three categories, based on implementation techniques for tools, and the intuition behind the methods. These are: control-flow methods (easy to implement efficiently, easy to understand); data-flow methods (somewhat more difficult to implement, sometimes difficult to grasp); and, data-coverage methods (hard to implement efficiently, hard to understand).

Control-flow Coverage

Statement-coverage testing is the prototype control-flow method. The places in the program that might be reached (the statements), must be reached by an adequate testset. Because statement coverage is almost trivial to measure, most commercial testing tools provide this measurement. *Branch testing* is also widely supported by testing tools. A testset achieves branch coverage if its tests force each program branch to take both TRUE and FALSE outcomes[10]. Branch testing is commonly preferred to statement testing, but without much justification (see section 5).

[9] For scientific subroutines, asymptotic behavior and an investigation of the continuity of a computed function are functional test possibilities.

[10] It is a pitfall of thinking about branch testing to imagine that two distinct test points are required to cover a single branch. It commonly happens that one test suffices, because the branch is within a loop. In particular, the conditional branch that begins a WHILE loop is always covered by any successful test case that enters the loop, since it must eventually return and take the exit direction.

Conditional branches employing AND and OR operators contain "hidden" paths that are not forced by branch- or statement-coverage testing, and in the presence of short-circuit evaluation, it may be of some importance whether or not these paths were taken. *Multi-condition coverage* is a version of branch testing that requires tests to make each sub-expression take both TRUE and FALSE values. *Loop coverage* requires that tests cause the loop body to be executed at least once, and also cause the body to be skipped[11].

The state of the test-tool art is nicely exemplified by the public-domain tool GCT (Generic Coverage Tool) [Mar91] for the C language. GCT comes with impressive test-sets for testing GCT itself. It includes branch, multi-condition, and loop coverage measures (as well as weak mutation, described below).

One structural testing method often spawns another when a deficiency is recognized in the first method. Branch testing and the other GCT control-flow methods could be said to arise from statement testing because they intuitively seem to cover more. The ultimate in control-flow coverage is *path testing*, in which it is required that tests in a testset cause the execution of every path in the program. The rationale behind path testing is that each path represents a distinct case that the program may handle differently (insofar as something different happens along each path), and so these cases must be tried. Unfortunately, there are a potential infinity of paths through any loop, corresponding to $0, 1, 2, ..., n, ...$ times through the body, and even nested IF statements quickly lead to too many paths, so complete path testing is usually not attempted in practice[12].

Testing tools that implement control-flow methods make use of exception reporting to minimize their reports. They identify not what *was* executed, but what *was not*. Tools are thus required to be able to identify the possibilities for coverage. This they necessarily do inaccurately, because determination of exactly which control flows can occur is in general an unsolvable problem. Invariably, practical tools use a simple worst-case algorithm: they treat a control flow as possible iff it forms a path in the uninterpreted program flowgraph. That is, they ignore the logical condition needed to execute the path, and consider only the connections of the graph. As a special case of this inaccuracy, a FOR loop appears to be able to execute its body any of $0, 1, 2, ..., n, ...$ times; in reality, for constant bounds only one count is possible. When a testing tool reports that coverage has failed, and statements, branches, paths, etc., are uncovered, it may be falsely listing situations that actually cannot occur. The tester is then faced with the *infeasible path* problem: to determine which of the reported coverage deficiencies are real (requiring additional test points to cover), and which are impossible to fulfill (to be ignored). In practice, these decisions are not very difficult for human beings to make [FW88].

Testing tools also commonly report fractional coverages, for example, "87% of all branches

[11] Again, this may not require more than one test point, if the loop in question is within another loop.

[12] Some early testing standards called for attempting path coverage, with some fudging for loops. There is confusion in the literature, exemplified by statements like "complete path testing is a perfect testing method that would find all possible failures, but it is impossible to carry it out because programs may have an infinity of paths." The reasoning is bad: on *each* of that infinity of paths, there is also a practical infinity of different data values possible, so that even if all the paths are covered, bugs could be missed [How76].

were covered." Such messages are the basis for testing standards that specify a fraction to be attained (80% is often suggested, but in practice, coverages may be much lower [RB85]). These summary numbers do measure testing effort, but their meaning is obscure, and it is easy to be misled (see section 5.1).

Data-flow Coverage

Complete path testing is considered impractical because of the potential infinity of paths through loops. A heuristic method of loop coverage that requires executing the loop body at least once, and also skipping it entirely, is not satisfactory. Experience shows that many subtle failures show up only on particular paths through loops, paths that are unlikely to be selected to satisfy a heuristic. Complete path testing really has something to offer, because programmers do think in terms of paths as special cases, that should subsequently be tested. *Data-flow testing* is an attempt to prescribe coverage of some of these special paths.

The central idea of data-flow coverage is the *def-use* or *DU pair* for a program variable. Variable V has a def[13] at each place in the program where V acquires a value, e.g., at assignments to V. V has a use at any other place it occurs. Thus for example, in the Pascal statement

```
x := x + 1
```

there is a use of `x` followed by a def of `x`. A DU path for V is a path in the program from a place where V has a def, to a place where V has a use, without any intervening def of V. A DU pair for V is the pair of start-end locations of any DU path. Thus the intuition behind data-flow testing is that DU paths are the special paths that programmers think about, paths on which values are stored, then later used. For example, in the Pascal code (with line numbers for identification):

```
1 x := 0;
2 while x < 2
3    begin
4    writelin("looping");
5    x := x + 1
6    end
```

there are the following DU paths for the variable `x`: 1-2, 1-2-3-4-5, 5-6-2, 5-6-2-3-4-5. Although it is possible to have an infinity of DU paths in a program (but not usual, as the example shows – most such paths are interrupted by a def), the number of DU pairs is always finite. This motivates the definition of the most common kind of data-flow test coverage: a testset achieves *all-uses coverage* if its data points cause the execution of each DU pair for each variable. (When multiple DU paths, perhaps an infinity of them, connect a DU

[13] "Def" is a poor choice of terminology, since the idea is that V acquires a value, not that it is "defined" in the sense of being declared or allocated storage. "Set" would have been a better choice, but "def" is too entrenched in the literature to change.

pair, only one such path need be covered in all-uses testing.) The technical definitions of data-flow testing are surprisingly difficult to give precisely, and there are several versions in the literature [RW85, FW88, PC90].

Among many variations of data-flow ideas, we mention only the extension to *dependency chains* [Cam90, Nta88]. A dependency chain occurs when there is a series of DU paths laid end to end, DU paths on which the use ending one is passed to the def beginning another, perhaps with a change of variable. For example, in the Pascal:

```
1 S := 0;
2 x := 0;
3 while x < 3
4    begin
5    S := S + x;
6    x := x + 1
7    end;
8 writeln(S)
```

there is a dependency chain beginning with the def of `x` at 6, and ending with the use of `S` at 8, passing from `x` to `S` at 5, then from `S` to `S` again at five, on the path 6-7-3-4-5-6-7-3-4-5-6-7-3-8. This dependency chain captures the contribution of both the loop index and the partial sum to the final sum, and it requires two iterations to observe it.

Implementing data-flow test tools is not much more difficult than implementing control-flow tools (indeed, some people do not distinguish data-flow as separate from control-flow). The instrumentation for observing what paths are executed is the same as for statement coverage. However, to calculate which DU pairs/paths/chains exist in the program requires construction of a dependency graph. This construction is static, based as usual on the uninterpreted flow graph, annotated with variable-usage information. Thus data-flow tools are subject to the infeasible-path difficulties described above.

Data Coverage

The defect in all control-flow and data-flow coverage methods is that each statement/-branch/path requires only a single set of internal data values to cover. That is, these testing methods do force the execution of control patterns, but only for a single case out of a potential infinity of cases that use the same pattern. Faults can remain hidden, if the failures they cause require the pattern to be traversed with special data values. The more complex the control pattern, the more important the problem of data choice becomes, because in attempting to cover a difficult control condition, a tester is likely to choose trivial data values, values that have little chance of exposing failure.

So-called "special-values testing," or "boundary testing" suggests the use of extreme values. Used in connection with a control- or data-flow method, boundary testing is very valuable in the hands of an experienced tester. For example, on a dependency chain that makes use of buffer storage, a boundary test would require the chain to be covered with the

buffer empty, with it just full, and with overflow. The applications of special-values testing are not systematic, but based on human insight into what values might cause things to go wrong. Ultimately, the "right" special values are the ones on which the program happens to fail, something that cannot be systematized.

A brute-force kind of "data-coverage" measurement for a testset can be obtained by instrumenting a program to record all of its internal states [Ham93]. The quality of the testset at each point in the program is measured by the variety of values assumed by the variables used there. Although neither the instrumentation nor the analysis of recorded data is difficult, such a system is not storage-efficient. The idea has a more fundamental deficiency because a data-coverage tool cannot report its results by exception. The problem of statically calculating the internal-state possibilities at an arbitrary program location is of course unsolvable; worse, there are no known approximation or useful worst-case algorithms. Thus the situation differs from that for control- and data-flow testing, in which the uninterpreted flow graph allows easy worst-case calculation of the possible paths, including only a few infeasible ones by mistake.

Mutation testing is a systematic method that approximates both boundary testing and data-coverage testing [Ham77]. From the program under test (PuT), a collection of *mutant* programs are created, each differing from the PuT by a small change in syntax. For example, in a conditional statement of the PuT, the relational operator ">" might be replaced with "$\neq$" in a mutant. Other mutants would use other relational operators in turn, the set of variations being determined by the particular mutation system. Now the PuT and all mutant variations are executed on a testset. The PuT must get correct results, and any mutant that does not agree with the PuT (that is, the mutant does not succeed on the testset) is termed *killed*. The testset achieves mutation coverage if it kills all mutants.

The idea behind mutation[14] is that mutant variations explore the quality of the data that reaches each program location. Good testsets cannot be fooled by any mutant. In the example of substituting "$\neq$" for ">", these two operators agree on the whole range where the ">" expression is TRUE. So long as test data falls in that range it is not covering very well. Mutation also systematizes the idea of "boundaries" in data; boundaries are literally edges of regions on which mutants are killed.

In principle, an extended kind of mutation, in which all possible variations of the PuT are considered, would constitute a perfect test method. Killing all the mutants would show that *no* other program agrees with the PuT on a testset, hence the PuT must be correct if any program is. But even the very restricted mutants of the experimental systems are too many to make the technique easily practical. Another implementation difficulty is that mutants may have termination difficulties the PuT did not have, for example, when loop-

[14]Mutation analysis was developed independently (and named) by Lipton [DLS78], with a rather different motivation. For Lipton, mutants capture a range of possibilities among which a programmer might have made a mistaken choice. Killing mutants with test data is experimental verification that the choice represented by the dead mutant is not the correct one. The description of mutation given above corresponds to the way it was implemented by Lipton's group; my implementation was built around a compiler, and was much more efficient, if more difficult to describe.

termination expressions are mutated. Technically, a mutant looping forever can never be killed; in practice, systems impose an arbitrary limit and silently and incorrectly kill long-running mutants.

There is a more difficult practical problem with mutation testing than its implementation, unfortunately. Just as in control-flow testing there may be infeasible paths so that the exception reporting of (say) a DU testing tool requires human investigation, so mutation sets its users an even more difficult task. Some mutants *cannot* be killed, because there is *no* test data on which they differ from the PuT – such a mutant is a program equivalent to the PuT. The problem of deciding mutant equivalence is unsolvable in principle, but also intractable in practice, and is a major stumbling block to acceptance for mutation testing.

A number of suggestions have been made to make mutation testing more practical. The most promising is to use "weak mutation" [How82, Ham77]. Weak mutation considers a mutant to be killed if it differs from the PuT in the state values immediately following the altered statement in the mutant. The mutation described above (now "strong mutation") requires that the program results differ. Of course, if a mutant is killed in the strong sense, it must also be killed in the weak sense by the same test data, but the converse is not true. It may happen that the mutation alters a state, but that subsequent actions ignore or correct this error. Weak mutation can be efficiently implemented (it is a part of the GCT tool mentioned above), and in some important cases (notably mutants of the relational operators) it is possible to derive simple conditions that describe test data that will necessarily kill all mutants. The problem of equivalent mutants remains in the weak case, but its practical difficulty is mitigated by focusing on just one state following the mutation.

3.3 What Should be Covered?

Coverage testing, both functional and structural, is intended to expose failures by systematically poking into the "corners" of software. It is reasonable to inquire into the principles of coverage – why does it work, and what makes it work better? Such principles would be useful in directing testing practice, because they would help a tester to decide which method to use, and how to tailor a method to particular software. For example, using statement coverage, should some statements receive more attention than others? A better understanding of coverage testing might also address the problem of composing test results from unit testing into conclusions for system testing. What coverage at the unit level would reduce failures when the units are combined into a system?

Marick's suggestion that structural coverage be used to assess the quality of functional coverage is one way that methods might be used together[15]. Section 3.2 has suggested that mutation was invented as a complement to control-flow testing. Each kind of coverage testing has its own rationale, and particular failures it is good at finding, so why not use them all?

[15] The converse of Marick's suggestion has not been made, but it sounds almost as reasonable: devise tests to achieve (say) statement coverage, then divide these tests into functional categories and note which functions have been omitted or neglected. One might hope that this would identify missing statements or point to code whose proper testing requires repeated executions.

In short: because there isn't time. Testing using even a single method (or no method at all) is an open-ended activity. It is always possible to take more time and do more testing. For example, functional classes can always be refined into smaller functional classes, requiring larger testsets to cover. Within each functional class, it is always possible to choose more test points. If methods are to be combined, how much should each be used relative to the others?

In research that will be considered in detail in section 5.2, "partition testing" methods were studied. Partition testing is an abstraction of coverage testing, which considers groups of inputs that achieve particular parts of coverage. For example, in complete path testing, the input space is divided into equivalence classes by which path those inputs take. These classes do not overlap (no input can take more than one path), and together they exhaust all inputs (every input must take some path[16]). The equivalence class for each path is the place from which tests must be drawn to cover that path. Other structural testing methods do not induce disjoint classes in this way (for example, the same input may cause execution of more than one statement, in statement testing), but their classes can be combined to create an artificial partition [HT90].

When the failure-finding ability of partition-testing methods was investigated [HT90, WJ91], an unsurprising result emerged: coverage works best at finding failures where there are more failures to find. That is, if the failure points are unequally distributed over the equivalence classes, then covering the classes of that partition will be more likely to find errors than using a partition in which the classes are uniform relative to failure. One way of looking at this result is as useless advice to "look for failures where they are," useless because if that information were available, testing would be unnecessary – we do not know where the failures are. But the advice is not so useless in different forms:

(1) Coverage works better when the different elements being covered have differing chances of failing. For example, path testing will be better at finding errors if some paths are more buggy than others. A corollary of (1) is:

(1a) Do not subdivide coverage classes unless there is reason to believe that the subdivision concentrates the chance of failure in some of the new subclasses. For example, it should help to break control-flow classes down by adding data boundary restrictions, but it would not be useful to refine functional classes based on function-parameter values if one parameter value is no more error-prone than another.

(2) Devise new coverage-testing methods based on probable sources of trouble in the development process. For example, emphasize the coverage of parts of code that has changed late in the development cycle, or code written by the least-experienced programmer, or code that has a history of failure, etc.

Although coverage testing is an "obvious" method, its theory is poorly understood. Open questions will be considered further in section 4.4.

[16] A class must be added to include those inputs that cause the program to go into an unterminating loop or otherwise give no output.

3.4 Testing for Failure in the Software Process

The accepted role for testing in the best development processes today is the one given by Myers: to find failures[17]. But when the development process is considered as a feedback system seeking to improve software quality, failure-finding is a dangerous measure. The trouble is that one cannot tell the difference between good methods used on good software, and poor methods used on poor software. Testing cannot be evaluated in isolation, nor can it be used to monitor other parts of the process, unless an independent control on faults is present. If there is no such control, one never knows whether finding more failures means improved testing, or merely worse software; finding fewer failures might mean better software, or it might mean only poorer testing.

It is not impossible to make sensible use of a failure-finding testing measure in the development process, but to do so requires better fundamental understanding of the interactions between software creation and testing than we possess today.

4 Testing for Reliability

When software is embedded in a larger engineering artifact (and today it is hard to find a product that does not have a software component), it is natural to ask how the software contributes to the reliability of the whole. Reliability is the fundamental statistical measure of engineering quality, expressing the probability that an artifact will fail in its operating environment, within a given period of operation.

4.1 Analogy to Physical Systems

The software failure process is utterly unlike random physical phenomena (such as wear, fabrication fluctuations, etc.) that underly statistical treatment of physical systems. All software failures are the result of discrete, explicit (if unintentional) design flaws. If a program is executed on inputs where it is incorrect, failure invariably occurs; on inputs where it is correct, failure never occurs. This situation is poorly described as probabilistic. Nevertheless, a software reliability theory has been constructed by analogy to the mechanical one [Sho83].

Suppose that a program fails on a fraction Θ of its possible inputs. It is true that Θ is a measure of the program's quality, but not necessarily a statistical one that can be estimated or predicted. The conventional statistical parameter corresponding to Θ is the instantaneous *hazard rate* or *failure intensity* z, measured in failures/sec. For physical systems that fail over time, z itself is a function of time. For example, it is common to take $z(t)$ as the "bathtub curve" shown in figure 1.

[17]The competing technology of software inspection, particularly in the design phase, shows promise of taking over the role of seeking faults. It is cheaper, and apparently effective. In the celebrated example of the space-shuttle software, spare-no-expense inspections simply made testing superfluous: testing seldom found failures in the the inspected software. In response to this, it has been suggested that inspection is just a form of testing, but from a technical standpoint this seems far-fetched – inspection usually does not

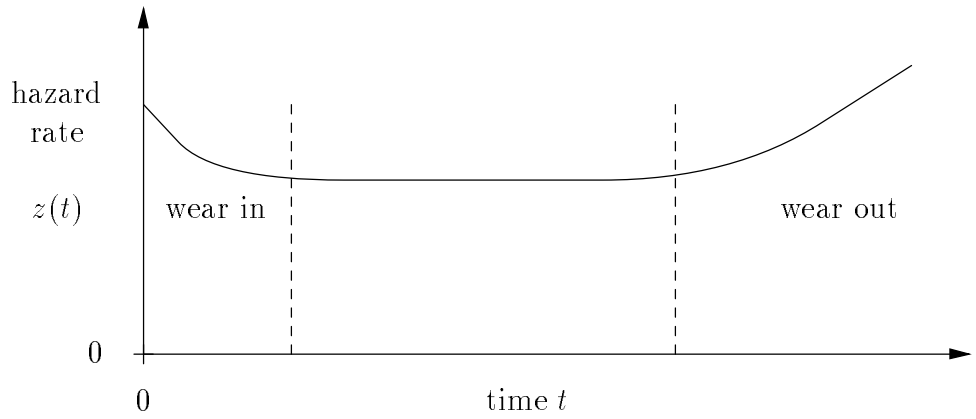

Figure 1: 'Bathtub' hazard rate function

When a physical system is new, it is more likely to fail because of fabrication flaws. Then it "wears in" and the failure intensity drops and remains nearly constant. Finally, near the end of its useful life, wear and tear makes the system increasingly likely to fail.

What is the corresponding situation for software? Is there a sensible idea of a software failure intensity? There are several complications that interfere with understanding. The first issue is time dependence of the failure intensity. A physical-system failure intensity is a function of time because the physical system changes. Software changes only if it is changed. Hence a time-dependent failure intensity is appropriate for describing the development process, or maintenance activities. (The question of changing usage is considered in section 4.4 below.) Only the simplest case, of an unchanging, "released" program is considered here. Thus we are *not* concerned with "reliability growth" during the debugging period [MIO87].

Some programs are in continuous operation, and their failure data is naturally presented as an event sequence. From recorded failure times t_1, t_2, ..., t_n starting at 0, it is possible to calculate a mean time to failure (MTTF)

$$\frac{t_1 + \sum_{i=1}^{n-1} (t_{i+1} - t_i)}{n}.$$

which is the primary statistical quality parameter for such programs. MTTF is of questionable statistical meaning for the same reasons that failure intensity is. It is a (usually unexamined) assumption of statistical theories for continuously operating programs that the inputs which drive the program's execution are "representative" of its use. The inputs supplied, and their representativeness are fundamental to the theory; the behavior in time is peculiar to continuously operating programs. Exactly the same underlying questions arise for the pure-function batch programs whose testing is being considered here. For such a program, the number of (assumed independent) runs replaces time, the failure intensity is "per run" (or sometimes, "per demand" if the system is thought of as awaiting an input). MTTF is then "mean runs to failure" (but we do not change the acronym).

involve execution, which is the hallmark of testing.

4.2 Random Testing

The *random testing* method was not described in section 3, because it is seldom used for finding failures[18]. Random testing recognizes that a testset is a sample taken from a program's input space, and requires that sample to be taken without bias. This is in strong contrast to the methods of section 3, each of which made use of detailed program or specification information. Random testing is intuitively useful for prediction. If an appropriate sample is used for the testset, results on that sample stand in for future program behavior, for which the inputs are unknown.

Random testing cannot be used unless there is a means of generating inputs "at random." Pseudorandom number generation algorithms have long been studied [Knu81], although the statistical properties of the typical generator supplied with a programming language are often poor. Pseudorandom numbers from a uniform distribution can be used as test inputs if a program's range of input values is known. In actual applications, this range is determined by hardware limitations such as word size, but it is better if the specification restricts the input domain. For example, a mathematical library routine might have adequate accuracy only in a certain range given in its specification. A uniform distribution, however, may not be appropriate.

Operational Profile

Statistical predictions from sampling have no validity unless the sample is "representative," which for software means that the testset must be drawn in the same way that future invocations will occur. An *input probability density* $d(x)$ is needed, expressing the probability that input x will actually occur in use. Given the function d, the *operational distribution* $F(x)$ is the cumulative probability[19] that an input will occur:

$$F(x) = \int_{-\infty}^{x} d(z)dz.$$

To generate a testset "according to operational distribution F," start with a collection of pseudorandom reals r uniformly distributed over [0,1], and generate $F^{-1}(r)$. For a detailed presentation, see [Ham94].

The distribution function d should technically be given as a part a program's specification. In practice, the best that can be obtained is a very crude approximation to d called

[18] Perhaps it should be used for failure detection, as discussed in section 5.2. In the cleanroom methodology, *only* random testing is used. Advocates say this is because the methodology produces near zero-defect software, so failure-finding is inappropriate. Critics might say that the advocates don't want to find any defects. In fact, the random testing does find some failures in cleanroom-developed code, but these are "easy bugs" that almost any test would uncover. Cleanroom testing is "betwixt and between" because it is not used to demonstrate that the code is bug-free (it would be interesting to see how the methods of section 3 would do); nor does it establish that the code is particularly reliable, as described in section 4.3, because far too few test points are used.

[19] The formula assumes that d is defined over the real numbers. The lower limit in the integral would change for restricted ranges of reals, and the integral would become a sum for a discrete density.

the *operational profile*. The program input space is broken down into a limited number of categories by function, and attempts are made to estimate the probability with which expected inputs will come from each category. Random testing is then conducted by drawing inputs from each category of the profile (using a uniform distribution within the category), in proportion to the estimated usage frequency.

4.3 Software Reliability Theory

If a statistical view of software failures is appropriate, failure intensity (or MTTF) can be measured for a program using random testing. Inputs are supplied at random according to the operational profile, and the failure intensity should be the long-term average of the ratio of failed runs to total runs. An exhaustive test might measure the failure intensity exactly. But whether or not failure intensity can be estimated with less than exhaustive testing depends on the sample size, and on unknown characteristics of programs. Too small a sample might inadvertently emphasize incorrect executions, and thus to estimate failure intensity that is falsely high. The more dangerous possibility is that failures will be unfairly avoided, and the estimate will be too optimistic. When a release test exposes no failures, a failure-intensity estimate of zero is the only one possible. If subsequent field failures show the estimate to be wrong, it demonstrates precisely the anti-statistical point of view. A more subtle criticism questions whether MTTF is stable – is it possible to perform repeated experiments in which the measured values of MTTF obey the law of large numbers?

In practice there is considerable difficulty with the operational profile:

1. Usage information may be expensive to obtain, or simply not available. In the best cases, the profile obtained is very coarse, having at most a few hundred usage probabilities for rough classes of inputs.

2. Different organizations (and different individuals within one organization) may have quite different profiles, which may change over time.

3. Testing with the wrong profile always gives overly optimistic results (because when no failures are seen, it cannot be because failures have been overemphasized!).

The concept of an operational profile does successfully explain changes observed over time in a program's (supposedly constant) failure intensity. It is common to experience a bathtub curve like figure 1. When a program is new to its users, they subject it to unorthodox inputs, following what might be called a "novice" operational profile, and experience a certain failure intensity. But as they learn to use the program, and what inputs to avoid, they gradually shift to an "average" user profile, where the failure intensity is lower, because this profile is closer to what the program's developer expected and tested. This transition corresponds to the "wear in" period in figure 1. Then, as the users become "expert," they again subject the program to unusual inputs, trying to stretch its capabilities to solve unexpected problems. Again the failure intensity rises, corresponding to the "wear out" part of figure 1.

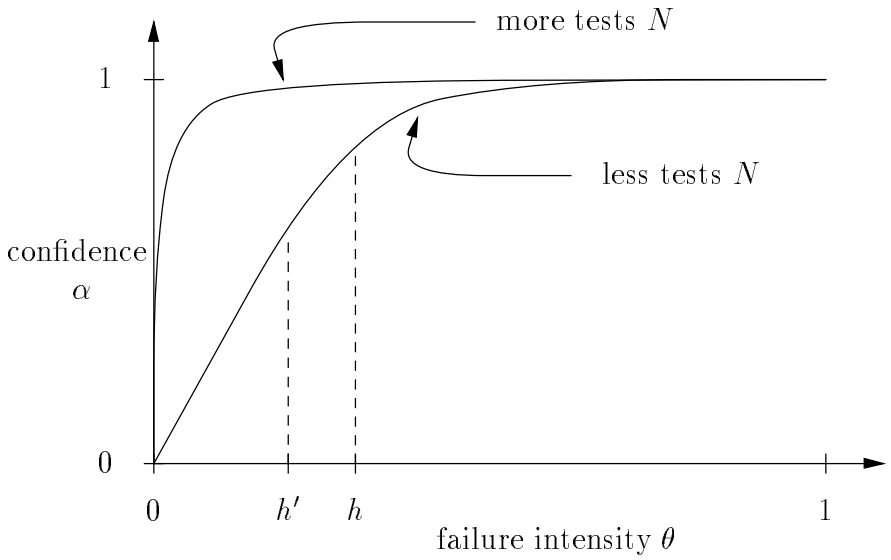

Figure 2: Confidence in failure intensity based on testing

Postulating an operational profile also allows us to derive the software-reliability theory developed at TRW [TLN78], which is quantitative, but less successful than the qualitative explanation of the bathtub curve. Suppose that there is a meaningful constant failure intensity Θ (in failures/run) for a program, and hence a MTTF of $1/\Theta$ runs, and a reliability of $e^{-\Theta M}$ over M runs [Sho83]. We wish to draw N random tests according to the operational profile, to establish an upper confidence bound α that Θ is below some level θ. These quantities are related by

$$1 - \sum_{j=0}^{F} \left(\begin{array}{c} N \\ j \end{array} \right) \theta^j (1-\theta)^{N-j} \geq \alpha \qquad (1)$$

if the N tests uncover F failures.

Some numerical values: for $F = 0$, $N = 3000$, $\alpha = .95$, $\theta = .001$, the MTTF is 1000 runs, and the reliability is 95% (for 50 runs), 61% (for 500 runs), and less than 1% (for 5000 runs). For the important special case $F = 0$, the confidence α is a family of curves indicated in figure 2. For any fixed value of N it is possible to trade higher confidence in a failure intensity such as h for lower confidence in a better intensity such as h'.

Equation 1 predicts software behavior based on testing, even in the practical release-testing case that no failures are observed. The only question is whether or not the theory's assumptions are valid for software. What is most striking about equation 1 is that it does not depend on any characteristics of the program being tested. Intuitively, we would expect the confidence in a given failure intensity to be lower for more complex software.

Another way to derive the relationship between confidence, testset size, and failure rate,

is to treat the test as an experiment checking the hypothesis that the failure rate lies below a given value. Butler and Finelli [BF91] obtain numerical values similar to those prediced by 1 in this way. They define the "ultra-reliable" region as failure rates in the range 10^{-8}/demand and below, and present a very convincing case that it is impractical to gain information in this region by testing. From equation 1, at the 90% confidence level, to predict a MTTF of M requires a successful testset of size roughly $2M$, so to predict ultrareliability by testing at one test point each second around the clock would require three years. (And of course, one must start over if the software is changed because a test fails.) Ultrareliability is appropriate for safety-critical applications like commercial flight-control programs and medical applications; in addition, because of a large customer base, popular PC software can be expected to fail within days of release unless it achieves ultrareliability.

Thus software reliability theory provides a pessimistic view of what can be achieved by testing. The deficiencies of the theory compound this difficulty. If an inaccurate profile is used for testing the results are invalid, and they always err in the direction of predicting better reliability than actually exists.

4.4 Profile Independence

The most dubious assumption made in conventional reliability theory is that there exists a constant failure intensity over the input space. It is illuminating to consider subdividing the input space, and applying the same theory to its parts.

Suppose a partition of the input space creates k subdomains $S_1, S_2, ..., S_k$, and the probability of failure in subdomain S_i (the subdomain failure intensity) is constant at Θ_i. Imagine an operational profile D such that points selected according to D fall into subdomain S_i with probability p_i. Then the failure intensity Θ under D is

$$\Theta = \sum_{i=1}^{k} p_i \Theta_i. \tag{2}$$

However, for a different profile D', different p'_i may well lead to a different $\Theta' = \sum_{i=1}^{k} p'_i \Theta_i$. For all profiles, the failure intensity cannot exceed

$$\Theta_{\max} = \max_{1 \le i \le k} \{\Theta_i\}. \tag{3}$$

because at worst a profile can emphasize the worst subdomain to the exclusion of all others. By coverage testing in all subdomains without failure, a bound can be established on $\Theta_{\max}$, and hence on the overall failure intensity for all distributions. Thus in one sense partition testing multiplies the reliability-testing problem by the number of subdomains. Instead of having to bound Θ using N tests from an operational profile, we must bound $\Theta_{\max}$ using N tests from a uniform distribution over the worst subdomain; but, since we don't know which subdomain is worst, we must bound all k of the Θ_i, which requires kN tests. However, the payback is a profile-independent result, that is, a reliability estimate based on partition testing applies to all profiles.

The obvious flaw in the above argument is that the chosen partion is unconstrained. All that is required is that its subdomains each have a constant failure intensity. (This requirement is a generalization of the idea of "homogeneous" subdomains, ones in which all inputs either fail; or, all the inputs there succeed.) But are there partitions with such subdomains? It seems intuitively clear that functional testing and path testing do not have subdomains with constant failure rates. Indeed, it is the non-homogeneity of subdomains in these methods that makes them less satisfactory for finding failures, as described in section 3. Of course, the failure intensity of a singleton subdomain is either 0 or 1 depending on whether its point fails or succeeds, but these ultimate subdomains correspond to usually impractical exhaustive testing.

The intuition that coverage testing is a good idea is probably based on an informal version of this argument that coverage gets around the operational profile to determine "usage independent" properties of the software. But making the intuitive argument precise shows that the kind of coverage (as reflected in the character of its subdomains) is crucial, and there is no research suggesting good candidate subdomains.

5 Comparing Test Methods

What objective criteria could be used to decide questions about the value of testing methods in general, or to compare the merits of different testing procedures? Historically, methods were evaluated either by unsupported theoretical discussion, or by "experiments" based on circular reasoning. The inventor of a new method was expected to argue that it was subjectively cleverer than its predecessors, and to compare it to other methods *in terms defined only by themselves.* For example, it was common to find a testset that satisfied the new method for some program, then see what fraction of branch coverage that testset attained; or, to find a testset for branch coverage and see how well it did with the new method. The new method was considered to be validated if its testset got high branch coverage but not the other way around. Such studies are really investigating special cases (for a few programs and a few testsets) of the "subsumes" relation[20].

5.1 Comparison Using the Subsumes Relation

Control- and data-flow methods can be compared based on which method is more "demanding." Intuitively, a method is "at least as demanding" as another if its testsets necessarily satisfy the other's coverage. The usual name for this relationship is *subsumes.* If method Z subsumes method X, then it is impossible to devise a method-Z test that is not also a method-X test. The widespread interpretation of "Z subsumes X" was that method Z is

[20]Mutation testing is the method most frequently used for comparison. Such studies take the viewpoint of that mutations are like seeded faults, and hence a method's ability to kill mutants is related to its failure-detection ability. However, if mutation is treated as a coverage criterion as in section 3.2, then such a comparison is like assessing one method of unknown worth with another such method.

superior to method X. (The most-used example is that branch testing is superior to statement testing, because branch coverage strictly subsumes statement coverage. However, it was suggested [Ham89] that subsumption could be misleading in the real sense that natural (say) branch tests fail to detect a failure that (different) natural statement tests find.

For example, suppose that the Pascal subprogram R

```
procedure rootabs(x: real): real;
begin
if x < 0 then x := -x;
rootabs := sq(x)
end
```

has specification $\{(x, \sqrt{|x|}) | x \text{ real}\}$, that is, that the output is to be the square root of the absolute value of an input. It would be natural to branch-test R with the testset $T_b = \{-1, 0, 1\}$, while $T_s = \{-9\}$ might be a statement-test testset, which does not achieve branch coverage. (This example shows why the subsumption of statement- by branch testing is strict.) Only T_s exposes a failure caused by the programmer mistaking Pascal's square function for its square-root function. Concentrating on the predicates that determine control flow leads to neglect of the statements, which here takes the form of trivial data values reaching the faulty function.

A continued exploration [WWH91] showed that the subsumes idea could be refined so that it was less likely to be misleading, and that it could be precisely studied by introducing a probability that each method would detect a failure. The behavior of any method is expressed by the probability that a testset satisfying that method, selected at random from all such testsets, will expose a failure. An example was given in which statement testing was more likely to detect a failure than was branch testing; however, even the contrived example was unable to evidence much superiority for the "less demanding" method, indicating that "subsumes" is not so misleading after all.

In a recent paper [FW93], the subsumes relationship is refined to "properly covers" and it is shown that the new relationship *cannot* be misleading in the probabilistic sense. Suppose two testing methods divide a program's input space into subdomains, one domain for each "element" to be covered[21]. It is the relationship between these two collections of subdomains that determines the failure-detection probability of one method relative to the other. Roughly, method Z *properly covers* method X if Z has a collection of subdomains from which the subdomains of X can be constructed. In the simplest case of partitions whose subdomains are disjoint, if each subdomain of X is a union of subdomains from Z, then Z properly covers X. Intuitively, there is no way for X to have many "good" testsets without Z having equally many, because the X subdomains can be made up from Z subdomains. When the subdomains do not form partitions, one must be careful in using multiply-counted subdomains of Z to make up subdomains of X – a single subdomain may not be used too often.

[21] For example, in all-uses testings, the elements are DU pairs executed; in mutation testing, the elements are mutants killed, etc.

As an example, simple programs exist to show that branch testing does not properly cover statement testing – the misleading program R given above is one. On the other hand, Frankl and Weyuker have shown (for example) that some methods of their dataflow-testing hierarchy are really better than others in the sense of properly covers.

The subsumes relationship began as a generalization of how well test methods do in each other's terms, that is, without any necessary reference to objectively desirable properties of the software being tested. A "more demanding" method Z that strictly subsumes X is actually better only if we assume that what Z demands is really useful beyond its mere definition. The notion of "misleading" introduced an objective measure (failure-detection ability), and showed that for natural testing methods, "subsumes" does not necessarily correlate with the objective measure.

It is easy to devise unnatural testing methods in which "subsumes" is more misleading. For example, the coverage measure (Z) "more than 70% of statements executed" strictly subsumes (X) "more than 50% of statements executed," but if the statements executed using Z happen to be all correct ones in some program, while those executed using X happen to include its buggy code, then X is actually better than Z for this program. This example demonstrates why fractional coverage measures are particularly poor ones.

5.2 Comparison for Reliability Prediction

Failure-detection probability is currently enshrined as the accurate measure of test quality, replacing a circular use of "coverage" to assess the quality of "coverage." But while it is questionable whether failure-detection probability is the appropriate measure of testing quality, it remains the only one on which work has been done. If testing is to be used to assess software quality, which we have argued is necessary for useful feedback in the software development process, then testing methods must be compared for their ability to predict a quality parameter like MTTF. But here the operational profile enters to invalidate comparisons. Only random testing can use the profile – other methods by their nature require testsets that distort usage frequencies.

The theoretical comparisons between random testing and partition testing alluded to in section 3.3 [DN84, HT90, WJ91], are often cited as showing that random testing is superior to coverage testing. Strictly speaking, these studies compare only failure-detection ability, of random testing and the partition-testing abstraction of coverage methods, and they mostly find partition testing to have the edge. The seminal study [DN84] intended to suggest that random testing was a reasonable alternative to the methods described in section 3.2. But because random testing's failure-detection probability is identical to the failure intensity (hazard rate) of reliability theory, it appears that these studies have an additional significance because they make comparisons based on reliability.

Unfortunately, even granting that the partition/random comparison applies to coverage testing, and that failure-detection probability for random testing determines MTTF, the only conclusion that can be reached is the negative one that coverage testing is at best no more significant than random testing, or at worst of no significance. A random test can establish

upper confidence bound α that the failure intensity is not above θ on the basis of N tests with F failures. Equation 1 connects these quantities according to the TRW software reliability theory. If a coverage test is as good a statistical sample as a random test, it might realize a similar or better bound on failure intensity. But the very intuitive factors that make coverage testing desirable for finding failures, make its failure-detection probability different from a failure-intensity. Coverage testing achieves superior failure detection precisely by sampling not the operational profile, but according to classes that emphasize failure. These classes bear no necessary relation to the operational profile, and hence the failure intensity may be large even though the failure-detection probability for coverage is small, if coverage testing found failures in low-profile-usage areas, and neglected high-profile-usage areas. The promise of coverage testing is that it might explore *all* usage areas without requiring knowledge of that usage, but no one knows how to surely realize this promise.

Thus the comparable or better failure-detection probabilities of coverage testing vis a vis random testing are not failure-intensity predictions at all, and there is as yet no information about the relationship between coverage testing and reliability.

6 Dependability

The technical difficulties with the notion of software reliability make it inappropriate for measuring software quality, except in the rare instances when a stable operational profile is available. Intuitively, a different measure is needed, one that is profile independent, thus removing the aspect of reliability that deals with the "operating environment." There are good grounds for removing this aspect, because software is intuitively perfectible, so that no "environment" (which after all is only an input collection that can be investigated ahead of time) can bring it down. The other aspect of reliability, its dependence on the period of operation, is also intuitively suspect for software. No matter how small the positive hazard rate, all physical systems eventually fail; that is, their long-term reliability is near zero. But software need not fail, if its designers' mistakes can be controlled.

We seek a software quality parameter with the same probabilistic character as "reliability," but without its dependence on environment or on period of operation. And we hope to estimate such a parameter by sampling similar to testing. The name we choose is "dependability," and its intuitive meaning is a probability that expresses confidence in the software, confidence that is higher (closer to 1) when more samples of its behavior have been taken. But this "behavior" may differ from that explored by the usual testing.

6.1 Reliability-based Dependability

Attempts to use the Nelson TRW (input-domain) reliability of section 4.3 to define dependability must find a way to handle the different reliability values that result from assuming different operational profiles, since dependability intuitively should not change with different users. It is the essence of dependability that the operating conditions cannot be controlled. Two ideas must be rejected:

U Define dependability as the Nelson reliability, but using a uniform distribution for the profile. This suggestion founders because some users with profiles that emphasize the failure regions of a program will experience *lower* reliability than the defined dependability. This is intuitively unacceptable.

W Define dependability as the Nelson reliability, but in the worst (functional) subdomain of each user's profile. This suggestion solves the difficulty with definition **U**, but reintroduces a dependency on a particular profile. In light of the dubious existence of constant failure intensities in subdomains (section 4.4) the idea may not be well defined.

A further difficulty with any suggestion basing dependability on reliability, is the impracticality of establishing ultrareliability. Other suggestions introduce essentially new ideas.

6.2 Testing for Probable Correctness

Dijkstra's famous aphorism that testing can establish only the *incorrectness* of software has never been very palatable to practical software developers, who believe in their hearts that extensive tests prove *something* about software quality. "Probable correctness" is a name for that illusive "something." The TRW reliability theory of section 4.3 provides only half of what is needed. Statistical testing supports statements like "In 95% of usage scenarios, the software should fail on less than 1% of the runs." These statements clearly involve software quality, but it is not very plausible to equate the upper confidence bound and the chance of success [Ham87], and turn the estimate "99.9% confidence in failure intensity less than .1%" into "probable correctness of 99.9%."

6.3 Testability Analysis

Jeff Voas has proposed [VM92] that reliability be combined with *testability* analysis to do better. Testability is a *lower* bound probability of failure if software contains faults, based on a model of the process by which faults become failures. A testability near 1 indicates a program that "wears its faults on its sleeve": if it can fail, it is very likely to fail under test. This idea captures the intuition that "almost any test" would find a bug, which is involved in the belief that well tested software is probably correct.

To define testability as the conditional probability that a program will fail under test *if it has any faults*, Voas models the failure process of a fault localized to one program location. For it to lead to failure, the fault must be executed, must produce an error in the local state, and that error must then persist to affect the result. The testability of a program location can then be estimated by executing the program as if it were being tested, but instead of observing the result, counting the execution, state-corruption, and propagation frequencies. Testability analysis thus employs a testset, but not an oracle.

When the testability is high at a location, it means that the testset caused that location to be executed frequently; these executions had a good chance of corrupting the local state;

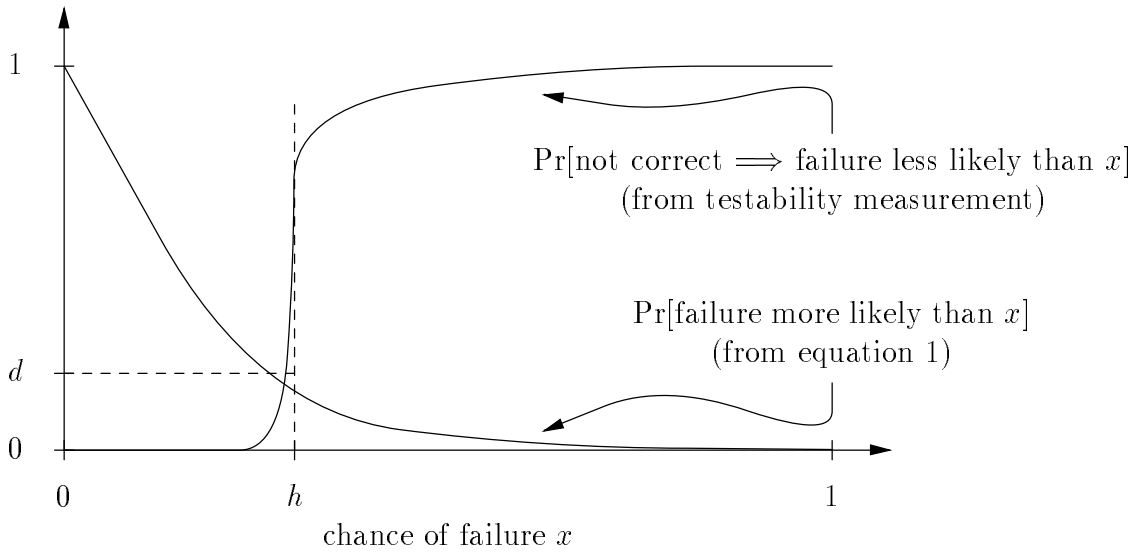

Figure 3: 'Squeeze play' between testability and reliability

and, an erroneous state was unlikely to be lost or corrected. The high testability does not mean that the program will fail; it means that if the location has a fault (but we do not know if it does), then the testset is likely to expose it.

Suppose that all the locations of a program are observed to have high testability, using a testset that reflects the operational profile. Then suppose that this same testset is used in successful random testing. (That is, the results are now observed, and no failures are seen.) The situation is then that (i) no failures were observed, but (ii) if there were faults, failures would have been observed. The conclusion is that there are no faults. This "squeeze play" plays off testability against reliability to gain confidence in correctness.

Figure 3 shows the quantitative analysis of the squeeze play between reliability and testability [HV93]. In figure 3, the falling curve is the confidence from reliability testing (it is $1 - \alpha$ from figure 2); the step function comes from observing a testability h. Together the curves make it unlikely that the chance of failure is large (testing), or that it is small (testability). The only other possibility is that the software is correct, for which $1 - d$ is a confidence bound, where d is slightly more than the value of the falling curve at h. Confidence that the software is correct can be made close to 1 by forcing h to the right [HV93]. For example, with a testability of .001, a random testset of 20,000 points predicts the probability that the tested program is not correct to be only about 2×10^{-9}.

6.4 Self-checking Programs

Manuel Blum has proposed [BK89] a quite different idea to replace reliability. He argues that many users of software are interested in a particular execution of a particular program only – they want assurance that a single result can be trusted. Blum has found a way to

sometimes exploit the low failure intensity of a "quality" program to gain this assurance. (Conventional reliability would presumably be used to estimate the program quality, but Blum has merely postulated that failure is unlikely.) Roughly, his idea is that a program should check its output by performing redundant computations. Even if these make use of the same algorithm, if the program is "close to correct," it is very unlikely that a sequence of checks could agree yet all be wrong.

Blum's idea represents a new viewpoint quite different from testing, because it is a *pointwise* view of quality. Testing attempts to predict future behavior of a program *uniformly*, that is, for all possible inputs; Blum is satisfied to make the prediction one point at a time (hence to be useful, the calculation must be made at runtime, when the point of interest is known). All of testing's problems with the user profile, test-point independence, constant failure intensity, etc., arise from the uniform viewpoint, and Blum solves them at a stroke. Testing to uniformly predict behavior suffers from the difficulty that for a high-quality program, failures are "needles in a haystack" – very unlikely, hence difficult to assess. Only impractically large samples have significance. Blum turns this problem to advantage: since failures are unlikely, on a single input if the calculation can be checked using the same program, the results will probably agree *unless they are wrong* – a wrong result is nearly impossible to replicate. The only danger is that the checking is really an exact repetition of the calculation – then agreement means nothing.

6.5 Defining Dependability

Either Voas's or Blum's idea could serve as a definition for dependability, since both capture a probabilistic confidence in the correctness of a program, a confidence based on sampling.

Dependability might be defined as the confidence in correctness given by Voas's squeeze play. Even if conventional reliability is used for the testing part of the squeeze play, the dependability so defined depends in an intuitively correct way on program size and complexity, because even Voas's simple model of testability introduces these factors. His model also introduces an implicit dependence on the size of both input and internal state spaces, but this part of the model has not yet been explored. Unfortunately, the dependability defined by the squeeze play is not independent of the operational profile.

Dependability might also be defined for a Blum self-checking program as the complement of the probability that checks agree but their common value is wrong. This dependability may be different for different inputs, and must be taken to be zero when the checks do not agree. Thus a definition based on Blum's idea must allow software to announce its own untrustworthiness (for some inputs). In some applications, software that is "self deprecating" ("Here's the result, but don't trust it!") would be acceptable, and preferred over a program that reported the wrong answer without admitting failure. One example is in accounting systems whose calculations are not time-critical; the program could be improved, or the result checked by some independent means. In other applications, notably real-time, safety-critical ones, reporting failure is not appropriate.

The promise of both Voas's and Blum's ideas is that they extend reliability to depend-

ability and at the same time substantially *reduce* the testing cost. Instead of requiring ultrareliability that cannot be measured or checked in practice, their ideas add a modest cost to reliability estimates of about 10^{-4} failures/run, estimates which can be made today. Blum's idea accrues the extra cost at runtime, for each result computed; Voas's idea is more like conventional testing in that it considers the whole input space, before release.

7 Conclusions

A brief history of software testing and its relationship to software quality can be constructed from two famous aphorisms:

> Testing can only show the *presence* of errors [failures, hence the underlying faults], never their *absence*. (Paraphrasing Dijkstra [DDH72])
>
> The purpose of testing is to find errors [failures – in order to fix the faults responsible]. (Paraphrasing Myers [Mye79])

Myers' statement could be thought of as a response to Dijkstra's, but the goal that Dijkstra implicitly assumes, to establish that software will *not* fail, remains. Myers' realistic goal was uncritically taken to be the same, but we have seen that it is not. Software reliability again addresses Dijkstra's goal, but using an engineering model (of which he would surely not approve!). The common wisdom about reliability is that (i) its theoretical assumptions are dubious [Ham92], but this is unimportant, because (ii) high reliability is impossible to measure or predict in practice [BF91].

To complete the picture, the emerging dependability theory answers Dijkstra directly, and may also respond to the question of practicality: it may be possible to measure a confidence probability that software *cannot* fail, that is, that it contains *no* faults, using methods similar to testing. If this promise is realized, the answer to Dijkstra makes for good engineering, if not for good mathematics: software developers will be able to predict trade-offs between their efforts and software quality, and can build software that is "good enough" for its purpose.

We have considered only the question of *measuring* software quality through testing, and using true quality measurements to direct the software development process. The important question of how to *attain* the necessary quality has not been addressed. A variety of methods, ranging from the very straightforward (e.g., inspections that employ people's experience without necessarily understanding just how they use it), to the very abstruse (e.g., completely formal methods of specification, and proof that specifications match intuitive requirements), are the subject of extensive research efforts. I have a vision of the future:

- Software quality will improve in fact, because development methods will be better.
- Software testing for failure will cease to be practiced, because with the better development methods, failure testing will be ineffective: it will find few failures.

- Instead of seeking failures, testing will be done, in conjunction with dependability analysis and measurement, for the purpose of assessing software quality, to determine if enough effort was put into development, and if that effort was effective.

Will the new software be better? Yes, if it needs to be, and the main thing is that the developers will *know* if the software is good enough. On the other hand, terrible software will continue to be produced (sometimes terrible is good enough!). What will become unusual is software that is *not* good enough, but whose developers honestly thought that it was.

Will the new software be cheaper? Probably not. Testing for dependability will not save over testing for failure, and the new up-front development methods will have an additional cost. But costs will be more predictable, and the situation in which there is a software disaster revealed only by final testing, will become rare.

References

[AH92] S. Antoy and R. Hamlet. Self-checking against formal specifications. In *International Conference on Computing and Information*, pages 355–360, Toronto, 1992.

[BF91] R. W. Butler and G. B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. In *Software for Critical Systems*, pages 66–76, New Orleans, LA, 1991.

[BHO89] M. J. Balcer, W. M. Hasling, and T. J. Ostrand. Automatic generation of test scripts from formal test specifications. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West, FL, 1989.

[BK89] M. Blum and S. Kannan. Designing programs that check their work. In *21st ACM Symposium of Theory of Computing*, pages 86–96, 1989.

[BS87] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. on Soft. Eng.*, 13:1278–1296, 1987.

[Cam90] J. Campbell. Data-flow analysis of software change. Master's thesis, Oregon Graduate Center, Portland, OR, 1990.

[CM90] R. H. Cobb and H. D. Mills. Engineering software under statistical quality control. *IEEE Software*, pages 44–54, November 1990.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1972.

[DLS78] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–43, 1978.

[DN84] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10:438–444, 1984.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable famile of data flow testing criteria. *IEEE Trans. on Soft. Eng.*, 14:1483–1498, 1988.

[FW93] P. Frankl and E. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. on Soft. Eng.*, 19:202–213, 1993.

[GCR93] S. Gerhart, D. Craigen, and T. Ralston. Observations on industrial practice using formal methods. In *15th International Conference on Software Engineering*, pages 24–33, Baltimore, MD, 1993.

[GG75] J. Goodenough and S. Gerhart. Towards a theory of test data selection. *IEEE Trans. on Soft. Eng.*, 1975.

[GHM81] J. Gannon, R. Hamlet, and P. McMullin. Data abstraction implementation, specification, and testing. *ACM Trans. Prog. Lang. and Systems*, 3:211–223, 1981.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. on Soft. Eng.*, 3:279–290, 1977.

[Ham87] R. G. Hamlet. Probable correctness theory. *Info. Proc. Letters*, 25:17–25, 1987.

[Ham89] R. G. Hamlet. Theoretical comparison of testing methods. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37, Key West, FL, 1989.

[Ham92] D. Hamlet. Are we testing for true reliability? *IEEE Software*, pages 21–27, July 1992.

[Ham93] R. Hamlet. Prototype testing tools. Technical Report TR93-10, Portland State University, Portland OR, 1993. (To appear in Software – Practice and Experience.).

[Ham94] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. on Soft. Eng.*, 2:208–215, 1976.

[How82] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. on Soft. Eng.*, 8:371–379, 1982.

[HT90] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.

[HV93] D. Hamlet and J. Voas. Faults on its sleeve: amplifying software reliability. In *International Symposium on Software Testing and Analysis*, pages 89–98, Boston, MA, 1993.

[Knu81] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1981.

[Mar91] B. Marick. Experience with the cost of different coverage goals for testing. In *Pacific Northwest Software Quality Conference*, pages 147–164, Portland, OR, 1991.

[MIO87] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, NY, 1987.

[Mye79] G. J. Myers. *The Art of Software Testing.* Wiley-Interscience, New York, NY, 1979.

[Nta88] S. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. on Soft. Eng.*, 14:250–256, 1988.

[PC90] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implication for software testing, debugging, and maintenance. *tse*, 16:965–979, 1990.

[PvSK90] D. L. Parnas, A. van Schouwen, and S. Kwan. Evaluation of safety-critical software. *Comm. of the ACM*, 33:638–648, 1990.

[RB85] J. Ramsey and V. Basili. Analyzing the test process using structural coverage. In *8th International Conference on Software Engineering*, pages 306–312, London, 1985.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Eng.*, 11:367–375, 1985.

[Sho83] M. L. Shooman. *Software Engineering Design, Reliability, and Management.* McGraw-Hill, New York, NY, 1983.

[TLN78] R. Thayer, M. Lipow, and E. Nelson. *Software Reliability.* North-Holland, New York, NY, 1978.

[VM92] J. M. Voas and K. W. Miller. Improving the software development process using testability research. In *Third International Symposium on Software Reliability Engineering*, pages 114–121, Research Triangle Park, NC, 1992.

[WJ91] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. on Soft. Eng.*, 17:703–711, 1991.

[WWH91] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *Symposium on Testing, Analysis, and Verification (TAV4)*, pages 1–10, Victoria, BC, 1991.