Subdomain (Partition) Testing

Dick Hamlet
Department of Computer Science
Portland State University
Portland, OR 97207 USA
+1 503 725 4036
hamlet@cs.pdx.edu

Abstract

In subdomain testing, the input space of the program being tested is divided using a criterion of 'sameness,' grouping input points such that it seems unnecessary to try many in the same group. These subdomains are then sampled: it is required that each be covered by at least one test. There are a number of pitfalls in this procedure because the points in a subdomain are never truly 'the same'; nevertheless, the method is the only imaginable way to systematize testing.

Key terms: test coverage, functional testing, structural testing, input partition, sampling, subdomain homogeneity

1 Introduction

Software testing of any kind is a process of sampling. Each test case for a piece of software must be chosen from a set of possibilities, here called the *test domain*. (See the formal definitions, to follow.) It is a basic fact of practical testing that this domain is almost always hopelessly large, so that the number of test samples possible during software development is a minuscule fraction of the possibilities. Thus the choice of test cases is the critical factor in any testing activity. At the same time, once software has been released its users proceed to sample the test domain, and may do so far more thoroughly than the developer could. A trivial example will illustrate this unfortunate situation. Imagine a program that makes a numerical calculation using two non-negative integer inputs. Further stipulate that the program correctly checks that its inputs *are* integers in the range $[0, 2^{32} - 1]$, issuing a proper message otherwise. This confines the test domain to about 2^{64} values. (Without the stipulation, the test domain would be effectively unlimited, since it would include unlimited strings for inputs, e.g., "`2B or not 2B, that is the ?`". It is very often the case that programs fail to check for valid input.) Suppose also that the developer has a collection of 10^4 integer test cases for which the required results are known, and has time to execute and check these. Discount any time required to fix problems that are encountered. (Again, the situation is idealized because typically no such collection may exist, and weeks of test time may not be available; fixing problems could be a lengthy operation.) Finally let this software be intended for the PC market with about 10 million users, each of whom uses the program 100 times a week on average. Altogether the idealized situation differs from reality in giving development testing a huge favorable bias. The test domain has about 4.3 billion points, of which 10^4, about .00023%, are tested, while usage is about 10^9 points/week, about 23% of the test domain. (It is true that a great deal depends on exactly *which* elements of the test domain are tried and which used, but that important subject will be raised later in Section 3.1.) Thus in this case some user hits almost every point within a month of usage, while only a negligible fraction have been tested. Thus if there are any problems with the software, it is quite likely that they will be seen by users but not by pre-release testers.

The quality of any pre-release testing effort can be measured in terms of user-experienced deficiencies: good testing exposes problems so that they can be fixed before release; bad testing leads to damaging, expensive problem reports from the field. Unfortunately, there is the complication of software quality: if the software has very few deficiencies prior to test, then there is little distinction between good and bad testing. No test will turn up much, and few problems will arise in the field. It is the common wisdom that the best development aims for quality to be built in – that it is impossible to test quality into poorly designed or

badly constructed software. Nevertheless, from the testing viewpoint, test selection should seek to find as many problems that users might see as possible.

To attack any problem whose magnitude or complication seems as overwhelming as test selection, the best strategy is divide and conquer. Subdomain testing provides the 'divide'; whether it conquers is another issue. Subdomain testing has also been called 'partition' testing, because it divides the test domain into a finite number of classes – *subdomains*. If these classes are disjoint and together exhaust the test domain, they constitute a *partition* of it in the strict mathematical sense. Not all subdomain divisions are mathematical partitions, so "subdomain testing" is preferred over "partition testing." The essence of subdomain testing is that after the test domain is divided, sampling is required *in each subdomain*. In the extreme case, a single sample is taken in each subdomain. It is intuitively clear that if subdomains are defined haphazardly, there is likely no difference between using the same number of test points in subdomains or using them without regard for the subdomain boundaries. The attraction of subdomain testing comes in selecting the subdomains in a purposeful way that will leverage the requirement to sample each into a significant advantage over haphazard sampling of the whole test domain. Intuitively, subdomain testing makes it more difficult for the tester to stupidly miss something in selecting test cases. Unfortunately, although many clever subdomain breakdowns have been suggested, many of them intuitively valuable, it has not been rigorously established that any is actually efficacious, even in limiting stupidity. Indeed, in mathematical comparison with random sampling over the whole test domain, which is a kind of opposite of subdomain testing, there is often little difference [6], and recent results [3] indicate that there may be tight theoretical limitations on how good subdomain testing can be.

1.1 Organization

This article is organized as follows:

A brief historical and definitional introduction is given in Section 2.

A number of the most common subdomain-testing schemes are described in Section 3, each with a rationale and critique.

The important special cases of testing software that retains and uses persistent state or co-operates concurrently with other software are described in Section 4.

The results of theoretical comparison with random testing are summarized in Section 5.

Conclusions as to when to use subdomain testing and what may be expected from it in practice (Section 6) end this article.

2 History, Definitions, and Scope

Although the origin of subdomain testing is not known, it probably arose in testing practice when programmers considered which software inputs were "the same" in some sense, and therefore more than one need not be tried. For example, suppose that a program is supposed to reject negative inputs with an error message. Then it should be enough to try one negative input and check for the message. Other negative inputs are 'the same' and need not be tried. The example shows both the plausibility of subdomain testing (here the subdomain is "all negative inputs") and its flaws. If the code begins with something like

```
READ x
IF (x < 0) THEN
  print "ERROR:  Negative input."
  return
FI
```

then the single test seems safe; but if instead the code is more obscure, perhaps not so. Furthermore, the input "-1111111111111111111111111111111111111111" might very well not cause the program to do the same thing as input '-1' because of peculiarities of the library `READ` routine or hardware limitations on stored

values. In subdomain terms, these difficulties mean that the "negative inputs" subdomain is not so simple or uniform as it might seem, and so testing it requires care.

"Negative inputs" is an example of a *functional*, or *specification-based* subdomain: it is defined by what the program should do. (See SPECIFICATION-BASED TEST CASE GENERATION, BLACK-BOX TESTING.) Another viewpoint is from the program code. In the example, there are inputs that cause the error-message statement to be executed, and these define a *structural* or *code-based* subdomain, "execute the error `print`". The flaw in trusting code-based subdomains is more subtle. Indeed, every input in this subdomain does hit the `print` statement by definition. But it can still happen that inputs in the subdomain do not have the same meaning. For example, the input "ABCDE" might look '< 0' to the library `READ` routine (or cause an exception), yet should have resulted in a different error message, or even a completely different program response.

The initial response to these difficulties was to combine functional and structural subdomains. The intersection between "negative inputs" and "execute the error print" is a subdomain where both are true, and from which complications have been eliminated. It was early proposed [18] that such intersections be used for subdomain testing. If the tester manages to find an input point in the intersection, it is sufficient to try it alone, since all other points in that subdomain are intuitively the same. Unfortunately, as a practical testing technique, intersecting subdomains doesn't work any better than using functional or structural subdomains by themselves. In the intersection there are other subdomains, such as "negative input that does not execute the error print" that should be tried, and the tester has no more idea how to try them than he/she had about how to see if any "negative inputs" values were peculiar. Indeed, if the program works properly, such subdomains must be empty. But how can a tester establish that an input subset is empty?

There is one subdomain breakdown that is unexceptionable: separate the test domain into those inputs where the program succeeds in doing what it is required to do (say subdomain S) and those where it fails (subdomain F). F and S should tell us what we really want to know from testing: whether or not the program works. But given F no testing at all is needed: a program is correct iff F is empty. Once F is shown to be empty (necessarily not by sampling F or S!), there is no need to sample at all.

In 1976, Bill Howden recognized that these intuitive difficulties with subdomain testing could be made precise. His paper [14] carefully defined the process of dividing the test domain and sampling in each subdomain, and he proved that it is an unsolvable problem to succeed with this in general. That is, there is no effective subdomain breakdown (that is, one which can be algorithmically carried out) which for all programs can be sampled to show correctness. Howden illustrated his proof using the structural subdomains of executing the same path in imperative code (see Section 3.2 for a definition), and showed that for a sample of small programs from a textbook, these subdomains necessarily detected some 35% of the program failures. Howden's result shows that it is pointless to seek perfect subdomain testing, that is, subdomains whose testing cannot succeed without the program being correct. A less ambitious goal would be probabilistic: to find a subdomain breakdown such that sampling each is provably better than sampling the test domain without regard for the subdomain boundaries. Attempts to investigate the probabilistic view have been inconclusive, but beginning with a seminal paper of Duran and Ntafos in 1985 [6], many studies indicate that if subdomain testing is better, it is not much better. These results are discussed in Section 5 below. Another practical way to proceed is to define particular kinds of failure and seek subdomains that necessarily detect them. This technique has been very successful in testing electrical and mechanical systems. Their "fault modes" are described, and tests designed to preclude each particular fault. Software seems to have an infinity of "fault modes," but see SOFTWARE FAULT-BASED TESTING for more information.

If subdomain testing has limited value, why is it almost universally used, and why do testers and programmers usually believe that it is more meaningful than can be precisely established? The answer comes from human psychology rather than from computer science: people feel that it is better to do something than to do nothing; and, when people have worked hard, they tend to confuse effort with results. Howden himself said it clearly in responding to a subdomain-intersection paper at a technical meeting. The paper called its method "rigorous"; Howden suggested that this be changed to "vigorous". However, our human intuitions are not without value. If testing must be done because it can uncover even a few problems, and if only subdomains make it possible to do in practice, then subdomain testing will be used.

In discussing subdomain testing, terminology common to testing in general is needed (indeed, even the introductory remarks above had to use some technical terms).

Definitions and discussion:

The *test domain* (*input domain*) for a program is an arbitrary set of values that might be program inputs. In general this set includes every possible value of every kind; however, in special cases it may be agreed that some values are excluded as impossible, e.g., because it is stipulated that the program only accepts mouse clicks, and an operating system can only deliver these in certain ranges. One way to restrict the test domain is to adjust a program's specification to make excluded values 'don't care' inputs, as described below. However, it is always dangerous to make stipulations that exclude arbitrary values, since it may very well happen that the assumptions on which the exclusion is based prove false in practice, exposing the program to completely untried cases from its users.

A *test* (*test case, test point*) is a value selected from the test domain on which to execute a program.

A collection of tests is called a *testset*. (The common usage of 'test' for the collection or for describing the process of testing, e.g., "the test took a week," will be avoided here.)

An *operational profile* (or *user profile*, see Operational Profile Testing) is a probability distribution over the test domain that assigns to each point the probability that it will occur when the program is in use. In practice, the only profile available is a crude histogram assigning weights to a few subsets of the test domain.

A *specification* for a program P is a description of what P is required to do, hence also *requirements*. If the description is mathematically precise, it is called a *formal specification*; commonly the specification is a natural language document interpreted by human beings. In principle, a specification should include an operational profile for its program, but this is almost unheard of.

A program P *succeeds* on input value x iff the result of executing P on input x agrees with its specification; otherwise P *fails* on x. By extension, success or failure on a testset means that P succeeds on all its members, or fails on at least one. (We also say that the test or testset succeeds or fails.) In situations where P does not produce any result, and/or where its specification does not describe what the result should be, there is disagreement: some say that no result always signifies failure; others that there can be unspecified "don't care" inputs on which the program cannot by definition fail, no matter what it does or does not do. Where we cannot avoid discussing such a situation, we take the latter view. A program is *correct* with respect to its specification iff it succeeds on all values of its input domain.

An *oracle* is a procedure for deciding, given two values x and y, whether or not a program producing result y on input x has succeeded. If the specification requires human interpretation, the only possible oracle is a subjective decision. Formal specifications can by contrast be *effective* or *automatic* oracles if using them to decide is algorithmic.

A *subdomain testing technique* (*coverage method*) is a means of determining subdomains for a given test domain, selecting a testset comprised of one or more tests from each, executing those tests, and judging the results. Should the testset succeed for some program, the program is said to have passed coverage testing or testing has achieved coverage (for the particular technique).

With this terminology, it is possible to state Howden's result more precisely:

Theorem (Howden [14]): There is no general algorithmic coverage method (that is, one that can be mechanically carried out for any program) with the property that when a program passes it, that program must be correct.

There is a practical aspect to subdomain testing which is shared with any testing activity. Tools that support testing must do bookkeeping, recording and categorizing tests and reporting on their outcomes.

Subdomain testing adds a bit to the bookkeeping overhead because each test point must be associated with a subdomain, but introduces no fundamental changes. See Software Testing Tools.

Finally, a distinction is drawn between different levels of "program" that might be tested. In unit testing (see Software Unit Testing) the program tested is only part of a software system (perhaps a subroutine or an off-the-shelf component); addressing the larger piece of software is called system testing. Subdomain testing is used at all levels, and for the most part the level is not material. However, particular subdomain techniques work better at different levels. For example, most system testing uses functional subdomains; structural subdomains and tool support work best in unit testing.

3 Common Subdomain Testing Techniques

All testing has the dual purpose of uncovering failures and increasing confidence in success. It is worth separating functional- from structural subdomain testing, because their intuitive viewpoints differ. The best way to see the difference is to consider the consequences of *not* covering subdomains of each kind. In a functional breakdown of the test domain, each subdomain corresponds to some case with a specified result, e.g., "clicking the **CANCEL** button should clear the screen." Suppose no such case is tried, that is, this subdomain is not sampled in testing. Then the tester has no idea what a user of the program will see when **CANCEL** is selected, an action very likely to occur. As noted in the introduction, trying the button once successfully (and hence covering the subdomain) is no guarantee of discovering every possible failure that might result from this action—**CANCEL** might fail under different circumstances—but if the button is never pushed the tester knows nothing about what a user might see.

On the other hand, what does it mean when a structural subdomain is missed? Consider as in the Introduction subdomains corresponding to executing each statement of an imperative program. Suppose no test covers a block of code that bears the comment "`--clear the screen`". Then by definition this code will never have been tried. One does not know if it is intended to be invoked by the **CANCEL** button, or how it might be invoked. But one surely has no evidence that it does not contain mistakes. In the most benign case the code might be impossible to reach (but might not that itself indicate a programming mistake?); in the worst case it is the only code used to clear the screen in a number of different circumstances, and it might fail to do so (say because of invoking the wrong operating-system function). Again, successfully invoking the code once (covering the subdomain) is no guarantee that a different invocation will not fail, but if the subdomain remains uncovered the tester knows nothing.

In both these scenarios, subdomain testing serves the purpose of preventing the tester from missing something that should be tried. The difference is whether that 'something' is a user action or a code action. Another important difference is that functional testing requires knowledge of the specification and the program's intended use and users. Structural testing, on the other hand, requires detailed knowledge of the code and run-time tools to monitor its execution. Intuitively, the two kinds of subdomain testing complement each other; unfortunately, to use both requires extra resources.

3.1 Functional Subdomain Testing

The adjective "functional" refers to a program's 'functionality,' what it should do, not to a mathematical function. Good functional testing turns on culling from the specification a set of these functions that is a good balance between generality (abstraction) and specificity (concreteness). Too far toward abstraction and the subdomains will be too large and each will have too much variation; too specific and there will be too many subdomains, many hardly different from each other. How easy it is to find good functional subdomains depends on how complex the software is and on how well its specification is organized and presented, as well as on the skill of the test designer. In a very rigid testing plan—for example, to meet a formal test standard for a contract—the functions to be tested may be prescribed. One such standard requires that each requirement in the specification be treated as a separate function. Since even a medium-sized software system can have thousands of requirements, this standard imposes a heavy testing burden. In most cases such regimentation will be counterproductive. It will be better for the tester to imagine her/himself to be

an end user of the software, and with the specification as a guide to run through the functions a user might select.

If there is an operational profile available, it should play a large role in selecting and sampling functional subdomains. Indeed, the usual profile already provides a functional subdomain breakdown, since it is a histogram giving the probability that certain test-domain subsets will be used; those subsets are natural functional-test subdomains. It is also natural to weight the testing effort by the profile, selecting more tests in proportion to the histogram weights. Harlan Mills has given a convincing argument [4] that the quality of testing depends largely on the frequency of occurrence of the failures found (partly determined by the user profile), not the number of failures found. A counter-intuitive corollary is that for very infrequent failures, it is not worth the effort to change the code, because debugging time is wasted on a failure estimated to occur only (say) every 5000 years of operation.

Any functional testing effort will use not only isolated test points, but sequences of tests that build on one another. Users typically have problems to solve, and use software to solve them in a sequence of related executions. Test sequences will be explored in Section 4.1 to follow.

There are some general techniques for gleaning functional subdomains from informal specifications. Specifications isolate cases by describing values of input parameters under which a case holds. Even the vaguest specification describes the input parameters themselves and their intuitive meanings. Two examples will indicate how this information can be mined for functional subdomains.

Identify function by conditional requirements. Suppose a requirement reads, "If the temperature goes above 90° C for more than 5 sec, the system shall turn on the pump." The condition defines an input subdomain of values from a temperature sensor and a clock, and the requirement defines the functional result.

Guess function from input parameters. If a program involved in word processing takes a string of characters as an input parameter, it is very likely that certain values of this string are of interest. For example, an empty string results from a user failing to supply an input field. White-space characters and/or punctuation characters are likely to be significant, dividing the string into intuitive 'words' and 'lines'. Upper and lower case may or may not be significant. Such an analysis leads to identifying cases like "several lines of more than one word each, mixed upper- and lower case"; these define functional subdomains.

A breakdown of the test domain using cases defined by conditional expressions can be organized into tables that allow specified results to be read off for each case. Indeed, it has been proposed [12] that tables of this kind should form the specification itself, transforming a natural-language document into one that is semi-formal.

Although the situation is slowly changing, even rudimentary specifications for a program may not exist, beyond a few comments in its code. But even in such cases, there is a method for devising reasonable functional subdomains, making use of intuitive knowledge about the problem the code is to solve. The program's potential input values can usually be associated with variable identifiers in the code, and the values of each can be grouped into subsets using common sense. The example above of the possibilities for an input string is typical, and requires almost no knowledge of what the program should do with the string. The *category-partition* method [17] is a way to turn a list of input parameters into a functional subdomain breakdown for testing. Each input parameter is assigned a number of value 'categories,' thus 'partitioning' the multidimensional input domain into subdomains that are cross products of the categories. Each cross-product subdomain singles out an intuitive case in which input variables take those values. The category-partition method is weakest in attaching required results to these subdomains (for use as an oracle), but often choosing a particular set of values in a subdomain will call up an intuitive idea of what the program should do there.

Specification-based partitions are subject to the general deficiency described in the introduction. Subdomains that are too general include several functions, and sampling may not try them all. But no matter how well chosen the functions/subdomains, the result of subdomain testing may be misleading. Just because the

sampled points succeed is no guarantee that other points in a subdomain cannot fail. The program being tested may act differently on different parts of a subdomain and some of these actions can be correct while others are not.

3.2 Structural Subdomain Testing

In contrast to specification-based testing, which is a very subjective and people-intensive activity, structural subdomain testing can be at least partly automated. Each structural method concentrates on some syntactic aspect of the program under test [16]. The test domain is divided by whether that aspect is invoked in execution. For imperative programs, the simplest intuitive structure is the block of code, a group of statements that are executed together under all circumstances. *Statement testing* defines a subdomain for each block: those inputs that cause that block to be executed. Statement-testing subdomains are not necessarily disjoint, since the same input can cause the execution of two distinct blocks. Any structural method introduces the possibility of *infeasible subdomains*, ones that are defined for the method but which in fact contain no points of the test domain. For statement testing, a block of dead code defines such an infeasible subdomain. Unfortunately, it is an unsolvable problem to determine if a particular code block is dead; thus statement testing is not a mechanical, algorithmic method. It shares this unsolvable feasibility problem with all other common structural methods.

There is a bewildering array of structural methods, each perhaps invented to improve on the deficiencies of the others. For example, in *branch testing* each conditional statement of an imperative program is used to define the two subdomains of inputs that cause it to take one of the two possible truth values. If a conditional statement has multiple parts formed using Boolean connectives, each of its two branch subdomains can be split into ones that cause just each part to be *true* or *false*. This variant is usually called *multi-condition* coverage. Because of the infeasible-subdomain problem, it may not be possible to carry out a complete structural-subdomain test, which leads to yet another unlimited array of subdomain-based methods for partial coverage. For example, *85% statement coverage* requires that tests be selected so that less than 15% of the statement subdomains are not sampled.

The ultimate structural subdomain testing technique for imperative programs is *path testing*. Subdomains are defined to correspond to each control-flow possibility (execution path) of a program. Achieving path coverage means that each such path has been tried at least once. Path testing is considered the high-end of control-flow techniques, because if paths are covered, so are all other structural control elements. For example, there cannot be a statement or branch possibility missed if paths are covered. Howden [14] chose path testing for his example, so it is instructive to see why this ultimate structural method failed to establish correctness in 65% of the small programs to which he applied it. Its deficiency is the same as that of all other subdomain methods: Sampling every path subdomain tries all the execution paths, but falls impossibly short of trying all program possibilities. Each path subdomain is comprised of a myriad of input points which traverse that particular path. Some of these may lead to test success and some to test failure. For example, taking a path that clears the user's screen may be sometimes be the correct thing to do, but sometimes a mistake. If the mistaken invocations are few among many correct invocations, any path subdomain test is almost certain to succeed, missing the failures hidden in the subdomain(s).

Path testing has another deficiency (not shared by branch- or statement testing): Most programs have too many paths. Each conditional statement in a program doubles the path count. A program containing an indeterminate loop (`WHILE` statement) has in principle an infinity of paths, since iterating the loop body 0, 1, 2, ... times technically each creates a distinct path. Thus in practice path testing must fail to cover an infinity of paths.

Dataflow testing is sometimes motivated as a method that selects a finite subset of the potential infinity of path-testing subdomains. However, it is probably better described as an attempt to add data information to control-flow methods. Among many variants [7], the most used is called *all-uses*. An all-uses subdomain for a given variable `V` comprises those inputs on which the value of `V` is set at a particular location in the program, then subsequently this `V`-value is used at another particular location (without having been set again between). Intuitively, the programmer has stored something in `V` for later examination. Of all structural coverage methods, dataflow coverage is the closest to functional coverage, since it is often possible

to associate a functional result with each particular store and use of `V`. For example, `V` may be a flag that is used to remember a condition naturally connected to a specific functional case.

All of the structural testing ideas that have been described are based on program control flow. (Dataflow is a partial exception, since it uses variable values to pick out paths.) There is, however, an entirely different kind of structural coverage called *mutation* [10, 5]. (See SOFTWARE MUTATION TESTING.) It imagines distorting a program into variants ('mutants'), and seeking test points that distinguish each variant from the original. Mutation subdomains consist of all inputs that 'kill' each mutant, that is, witness that it produces some result different from the original. The intuition behind the method is that without a test to tell a program from its mutants, a mutant might as well be what's wanted. If there were no restrictions on the form of mutants, this method would work perfectly: Unless the original program is correct, some variant (more correct) differs from it, but the original program must fail on tests that would kill this variant. But without restrictions, there would be an infinity of mutations and mutation subdomains. In practice, the mutations allowed are very narrow indeed. It is usual to create each mutant by making only one small change at the expression/statement level, for example, to alter an original assignment statement

```
X = (X+1)*Y
```

to

```
Y = (X+1)*Y.
```

Even this drastic restriction is not enough to control the cost of mutation testing very well—checking mutation coverage is far more compute-intensive than checking other structural methods. The most difficult part of using mutation testing is its infeasible-subdomain problem: A subdomain is infeasible if the mutant from which it arises cannot be killed; that is, its mutant is a program equivalent to the original. It is much more difficult to recognize equivalence in programs than (say) to see that some control transfer is impossible.

The descriptions of structural methods in this article are necessarily brief, and many variations have been omitted.

The most seductive aspect of structural testing is its potential for automation. (See SOFTWARE AND SYSTEM AUTOMATED TESTING.) Since subdomains and their coverage are defined by the execution of some program syntactical feature, it is in principle possible to mechanically generate subdomains and test points within them, thus eliminating the labor-intensive activity of test generation. Alternately, given a testset that achieves structural coverage, it may be possible to mechanically reduce it to a smaller testset without reducing the coverage. Even without automation, structural testing seems to be more 'systematic' than functional testing, since its goal is to cover items from a finite list (the program syntax); in contrast, functional testing has an open-ended subjective feel. Engineers are happier engaging in tasks with a clear direction that can be seen to be completed by their efforts. "Find a test point that will execute statement 93," seems to fit the bill.

Unfortunately, structural test coverage is only a surrogate for what testing is supposed to accomplish. It is all too possible to busily attain coverage, yet find few failures and gain little confidence that software works. The underlying reason is that functional testing, with its end-user viewpoint, is closer to testing's goals. In contrast, when all structural subdomains have been covered, it is still possible to have missed important functionality. Concentrating on structural coverage diverts attention from the real problem in favor of details in the surrogate problem. For example, one way to achieve statement coverage is to analyze a program's pattern of conditional statements. An uncovered block of code has a series of guards in the code that lead to it, and will be executed if this chain is satisfied. A systematic attempt to execute such a block naturally concentrates on the conditions leading to it. The tester studies the code and comes up with simple cases that enable all the necessary conditions. However, such cases usually have no correspondence with the software's functionality. Zero values, empty strings, etc., make it easier to trace and control conditionals in code, but such values are unlikely to occur in actual usage. Thus the test points that an engineer contrives to achieve statement coverage are the least likely to expose problems that will arise in real usage of the program.

There is, however, a way to exploit the complementary nature of functional and structural subdomain testing, by using structural methods as an *adequacy criterion* [15] for functional testing. The procedure is this:

Functional test. The tester uses the specification to create functional subdomains and chooses test points to cover them in the usual way.

Check structural coverage. Using algorithmic tools, the functional testset is checked for structural coverage. The procedure terminates if the structural coverage is complete. Otherwise:

Add functional tests. Examine the untested structural element for clues as to what function it was intended to perform. Then using the specification, add a functional test case that should utilize it.

Repeat the structural-coverage check above.

What is essential to this procedure, what allows it to avoid the trap of getting caught up in counterproductive structural details, is the third step: Instead of seeking to improve structural coverage by studying the code, one seeks to improve the *functional* coverage, with structure as the clue to missing functions.

3.3 Comparing Subdomain Testing Methods

Because control-flow structural coverage methods were the first to be invented and are the easiest to support with tools, they have received the widest study. Among these methods there is a natural subset hierarchy called the *subsumes* ordering. Method M1 subsumes method M2 iff any covering testset of M1 is also a covering testset for M2. It was noted above that path testing (strictly) subsumes all other control-flow methods. For another example, 85% statement coverage strictly subsumes 80% statement coverage. (Generalizations of this second example are obvious.)

'Subsumes' is a much trickier idea than it seems. First, not all methods are ordered by it. In particular, there is no subsumes relationship between functional and structural methods, the ones we would most like to compare. Second, the 'better' method in the sense of subsumes is not always really better. When there is a large gap between two methods M1 and M2 and M1 subsumes M2, it means that the testsets covering M1 are a small subset of those covering M2. It can happen that what should be tested in some program is more frequently encountered by M2; that is, of all the covering M2 testsets, say half have a chance of finding some problem. The very difficulty of achieving M1 coverage may mean that only (say) a quarter of the covering M1 testsets encounter the problem. Then if the tester picks a testset without knowing of the problem being sought (always the tester's situation!), the chance that an M2 testset will find it is twice as high as the chance that an M1 testset will. Furthermore, the M2 testsets are much easier to generate.

Empirical studies aimed at comparing testing methods are very difficult to do properly [8], and those that have been done have not shown a clear advantage for any method. The choice of method then comes down to what best fits into a particular development methodology. For example, one large advantage of functional testing is that its tests can be devised as soon as the specification is available; structural testing must wait for the advent of executable code, but it has better tool support.

4 Subdomain Testing Programs with Persistent State and Concurrency

In the previous parts of this article, testing has implicitly treated programs as computing what are called 'pure functions'. It was assumed that a program has no 'memory' (or 'state') retained from one execution to the next and also that the program has no source of non-determinism such as parallel execution of its parts; if a test point should be repeated, the result will be the same each time. Some programs do behave this way, and it is easy to detect the possibility that stateless or concurrency assumptions are violated. In order to retain state from run to run or be non-deterministic, a program must make blatant calls to the operating system. Section 4.1 provides a complete discussion of subdomain testing involving state, which is seldom handled correctly in practice. When it comes to concurrent execution, testing is on shaky foundations, but perhaps the sources of difficulty are not fundamental, as described in Section 4.2.

4.1 Persistent State

Most programs use persistent storage, which is the source of their power. On one run they record information, and on another they examine it. It has been learned from sad experience that keeping state is also a major source of subtle failures that escape detection during testing. One situation is that some program function is tested successfully, but in a particular state with which the tester was unconcerned. Then the program gets into a different state, where that function was not tested and fails in use. A classic example is an aircraft control program in which the function 'retract landing gear' was tested in the state 'airborne,' but not for the state 'landed,' and was later found to work all too well in the latter state. A more subtle situation arises when one execution succeeds, but leaves the program in an erroneous state that will cause a subsequent execution to fail without apparent reason.

State values are often taken to be 'inputs,' in that a program can examine and use them in its computations just as it can use input values, and hence state partially determines what actions a program will take. If state were no more than another input, it could be added to subdomain testing easily: each input subdomain could be split into several state subdomains, each of which would thus be sampled. (It amounts to the same thing to imagine subdomains for the state, each of which is divided into several input subdomains.) Unfortunately, the simple view that state is an input is wrong. Input is not controlled by the program but by its user, and so therefore by its tester. State the program controls absolutely. A user or tester may imagine that he/she 'puts the program' in some particular state, but in fact what happens is that the user supplies input, on which the program execution establishes state. The distinction is critical for testing, because the program may malfunction in setting state, making a mockery of what the tester is trying to cover. State thus acts also like output.

There is also a pernicious interplay between state specification and state implementation. Many specifications describe abstract states and the transitions between them that a program is to make. The very intuitive and powerful notation of the *state machine* [11] is a good mechanism for giving the description. In coding a program to meet such a specification, the programmer may attempt to work with concrete analogs of the specified states, and to implement the specified transitions. But as in all programming tasks, mistakes can be made, creating a mismatch between the real states and those that should occur. To follow a sequence of specified state transitions may mean nothing at all to the real program state.

Finally, there is the 'infeasible state' problem. Since the program controls state, it may be impossible to enter a particular state, because no combination of inputs can set that state value. The specification may have such infeasible states (but they may not be known!); code may have them as well (and the ones from specification and code may be different!). It is usual that persistent storage has a rigid and peculiar format, which programs maintain as a primary duty; thus in practice almost all arbitrary state values are infeasible. For example, a database is a file, but almost all files are not databases.

The correct way to sample behaviors of a program with state is as follows:

1. Adjust the environment so that the state appears 'reset' or 'uninitialized' to the program. To make use of state, a program must know when things are just beginning, typically to create and save initial state values. All testing starts from this reset state, and from it testing is repeatable.

2. Select a test point and execute the program. Record the output values and resultant state values.

3. Continue, selecting a sequence of input test points, executing, and recording output and result state.

4. Each input selection records two pairs: the (input, input-state) pair that begins it, and the (output, result-state) pair that ends it. The input value in the first pair is the tester's arbitrary choice, but the input-state value is not—it is the result-state value from the previous selection.

5. At some point the tester chooses to end the sequence. The list of two-pair values (each having a beginning and ending pair) is then a record of the testing activity.

Each time a sequence as described is selected, it constitutes a composite 'test point' for the program with state, a 'point' composed of member (input, input-state) beginning pairs in the order they occur. Only feasible state values will occur in the sequences.

Subdomain testing of a program with state must deal with two spaces—the test domain of inputs and the state domain. These must be kept separate, because the former is within the power of the tester to sample but the latter is not—it is under program control. Each space can be independently divided into its own functional or structural subdomains as in Section 3. The (input × state) cross products, however, are not those to be sampled directly. Rather, sequences of inputs are chosen as described above. The pairs of beginning values in a sequence fall into one of the (input × state) cross-product subdomains, which has therefore been implicitly sampled by the sequence composite 'test point'.

Only one question remains: How can the cross-product subdomains be covered? That is, how should a tester choose the testing sequences? Nothing like this question arises for stateless program testing, since there it does not matter in what order test points are selected, so the tester can just systematically go through the input subdomains. To even measure subdomain coverage in both input- and state domains extensive bookkeeping is required. At each point along each testing sequence, every possible choice of input subdomain may be tried. It isn't enough to ignore the position in the sequence, because this may miss some state values that otherwise could arise. Furthermore, the particular choice of an input value from an input subdomain may alter the sampling of states. Whatever choices of sequences a tester makes, some of the (input, state) cross-product subdomains will be covered, but most will not, simply because most states are infeasible. It is an open question how to attain the widest state coverage; indeed, there is little evidence for or against the position that covering state is a good idea.

Specifications typically do not provide much guidance in choosing testing sequences. Even when they describe how state should influence the results, they may be vague about how state values are established. However, specification by state-machine diagram [11] is exceptional. State-machine specifications not only describe required states, but also the transitions among them. Of course, a programmer may choose not to implement states or transitions from the specification, perhaps because there is a more efficient way to gain the same results. By and large, however, it is safer and easier to implement a specified state machine faithfully. State-machine states form natural singleton subdomans, so there is no state-subdomain sampling problem. More important, there are mechanical rules for state-machine construction that preclude infeasible states in the specification. It is sufficient to require that each state in the diagram be reachable from the initial state. That establishes a testing sequence to bring the abstract state to any one of its values. Furthermore, if in every state a transition is defined for every possible input, it cannot happen that the result is unspecified on any testing sequence as described above.

A state-machine specification thus defines a number of coverage measures that parallel path coverage in control flow, with the states themselves analogous to control points, and the transitions of the state-machine graph analogous to execution flow of control. One could ask for coverage of all states (analogous to statement coverage in Section 3.2), or all state transitions (branch coverage), or all sequences of state transitions (path coverage)[1]. For a finite-state machine (FSM), all of these coverages are decidable—that is, there is no infeasible-state problem for an FSM. But very few specification state machines are FSMs, because the input space is not finite; then, all the coverage problems return to being unsolvable. Thus in principle all the additional difficulties of choosing state-subdomain-covering test sequences disappear when there is a state-machine specification. There are no infeasible states, and the information needed to achieve coverage can frequently be sought in the specification transition diagram. This soothing ointment has only one fly in it: the program may fail to correctly implement specification states.

Even if an attempt was made to faithfully implement a specification state machine, mistakes may have been made. Unfortunately, the possibilities for error can be spread across the input domain, so that each input subdomain is subject to the potential problem that some of its values follow the specification and some do not. Hence the testing results depend on choices made in input subdomains as each testing sequence advances. Selecting 'good' points that mimic the specification will lead to testing success; but 'bad' points will carry the program off into state never-never land. The worst of it is that 'bad' choices in a testing sequence need not fail; instead they may only put the program in a strange state where some subsequent inputs would lead to failure, but the testing sequence ends before this happens. This is precisely one way that

[1]Although it is possible to imagine somewhat far-fetched analogs for dataflow coverage and for mutation, these have not been seriously explored.

state leads to obscure failures. The best that a tester can do is to define a careful correspondence between program-state values and specification-state values, and rigorously check this correspondence at each step in each testing sequence. It can still happen that the program goes state-wrong for untried sequences, but at least the testing actually carried out will not be spurious. It is uncommon to recognize the need to match implementation states with those specified—most testers assume without any evidence that they are the same.

4.2 Concurrency

If program testing theory lags behind programming that uses state, it falls much farther behind concurrent programming. Concurrent mechanisms range from two processes using operating-system calls to share information in a single memory, to programs running in different computers and communicating across the Internet. The essence of concurrency, from the standpoint of testing a program using it, is that there is an exchange of information between the program and some outside agency that the program does not control. Information may pass both ways, so the interaction can act like an input or an output. The outside agency can store and retrieve information, so the interaction may act like persistent state. Or, because the outside agency is itself a program, the interaction may be more complicated than any of these. Furthermore, because two interacting programs cannot be sure of each others execution speed, synchronization may be used (or not used!) to control (or fail to!) when in each execution interaction takes place.

It would be highly desirable to test parts of a concurrent system independently. These parts are not strictly "units" as usually unit tested. For clarity, borrow an operating-system term and call them "processes," and their combination a "system." Unfortunately, the processes are inextricably entwined by their communication and control connections[2]. If no constraints are placed on processes executing in parallel, their system behavior can be very complicated. In general the composite execution consists of interleaved segments on different processors, exchanging information at arbitrary waypoints. For a single invocation of the system, the lengths of execution segments and the information exchanged can vary because of varying processor and communication delays. A test of such a system does not have a repeatable outcome, and infrequently occurring patterns are unlikely to be tested. In later usage when they do occur, the software can fail. Writing programs that can't be properly tested is certainly unwise, so there have been attempts to tame the dragon of concurrency through programming language design. Language constructs such as monitors [13] control process interleaving and make it easier for programmers to recognize and avoid the worst problems, such as deadlock or race conditions. Unfortunately, parallelism is useful only if two communicating processes overlap their actual executions. Once overlap is allowed, however strongly constrained, it is an unsolvable problem to decide if in fact the processes necessarily follow a safe pattern. Subdomain testing can help.

At a minimum, two processes P_1 and P_2 executing in parallel must be able to 'rendezvous' and exchange information. To capture an exchange abstractly, imagine that each process has two input domains and two output ranges. One input/output pair connects to the external (human) world and one to the other process. A representative system execution might be described by: (1) P_1 receives an external input, (2) P_1 sends an output to P_2, (3) P_2 receives this, (4) P_2 sends an output to P_1, which (5) P_1 receives, and (6) P_1 sends output to the external world. Except for the points of exchange (2)-(3) and (4)-(5), both processes are executing simultaneously. In true blackbox testing of such a system, only the external input domain can be sampled and only the external output range observed. Subdomain testing can divide the external input space, but this seems to miss the point of testing concurrency. A better decomposition might consider each process separately. P_1's execution is controlled by two inputs, one external and one from P_2, so it can be tested in isolation over the cross-product of these two input domains $E \times I_1$, each domain divided into subdomains. Sampling a two-dimensional input subdomain like $S = (S_E, S_{I_1})$ intuitively probes what P_1 does on an external input (a member of S_E) in the context of a communication from P_2 (a member of S_{I_1}). Subdomains like S allow the tester to force execution of diverse aspects of the system. For example, it might be decided to define external subdomains based on system functionality, but define subdomains for

[2]Any unit testing must deal with the problem of "stubs," stand-ins for other units not part of the test, but this problem is far more severe with concurrent processes.

communication using the structure of P_1. This discussion is suggestive rather than precise; a more detailed presentation can be found in a monograph on software component testing [9].

There are superficial similarities between concurrent, cross-product input spaces and the (input × state) test spaces for a program with persistent state. But since (in the example above) P_1 does *not* control the space S_{I_1}, it should be sampled. It is true that many I_1 test points are wasted because it is likely that S_2 does not send them, but that's the whole point of testing—the world outside P_1 cannot be trusted, whether it comes from human beings (E) or from another process (I_1 from P_2).

The discussion above barely scratches the surface of concurrent systems, suggesting how little is known about testing them in even the simplest cases. Subdomain testing is intended to isolate cases so that nothing is missed; it is hard to escape the intuition that control of that kind simply isn't possible in the face of general concurrency.

5 Comparison of Subdomain Testing with Other Testing Methods

For most practical testers, functional subdomain testing is their bread and butter; they have never considered its near-opposite, random testing. Serious consideration of random testing is not made easier by the erroneous but commonly held belief that 'random' means 'haphazard' or 'ill-defined'. Real random testing over a test domain is a perfect method with which to compare subdomain testing, since its definition is precisely that the choice of test points makes no use of any systematic relationship among points in the test domain. Furthermore, it is easy to assess the effectiveness of random testing probabilistically. Probability analysis of subdomain testing is more difficult, but some surprising results have been obtained.

Given an input domain, (uniform) random testing employs a testset (say of size N) chosen from a uniform distribution[3] over the test domain. Suppose that such a testset succeeds for some program (the usual case for software of good quality and practical values of N). Assuming the failure rate of the program (the probability that it fails on a randomly selected test point) is constant, the upper confidence bound C that the failure rate lies below F is

$C = 1 - (1 - F)^N$.

Setting C, the desired confidence, and N, the number of tests, F can be calculated, with a significance like the following:

> Random testing using about 23,000 points without failure ($N = 23,000$) establishes that the program will not fail more than one time in 10,000 ($F = 10^{-4}$), with a confidence of 90% ($C = 0.9$).

In a seminal 1985 paper, Duran and Ntafos [6] published a theoretical comparison between arbitrary subdomain testing and random testing for the stateless case. They used simulation to analyze subdomain testing's probability of finding at least one failure (among other probabilistic measures), to compare with F from random testing. Random testing did surprisingly well, or put another way, it was surprising that subdomain testing did not do as much better than random testing as expected. Their paper led to a flurry of theoretical research with similar results. Along the way this research provided some deep insights into when subdomain testing works best and why [1]. It remains to explain why there is not a greater disparity between subdomain- and random testing.

The first researchers to attempt an explanation have found a surprising bound on the effectiveness of subdomain testing, and in fact on all other forms of testing, in comparison with random testing. It seems that nothing can be done to improve very much on random testing. Chen et al. [3] derived a theoretical bound on how much random testing could be improved. The bound is different for different probabilistic measures of test effectiveness, but in all cases the possible improvement is modest (e.g., a factor of 2). Their analysis is limited to stateless programs and rests on assumptions about the description of 'failure domains,' shapes of subsets in the test domain where the program fails at every point in the subset. There is no reason to believe that their assumptions are fundamentally limiting, however.

[3]The practical difficulties of defining a distribution over non-numeric domains do not concern us here.

Granting for the sake of argument that further research will confirm these results, the proper interpretation is the one originally suggested by Duran and Ntafos: random testing deserves serious investigation. It's not that subdomain testing isn't a good idea; rather, random testing is also a good idea. The current notion of adaptive random testing (ART) is an attempt to combine the methods, and it may realize almost all of the possible gain over random testing [2].

6 When to use Subdomain Testing

Any attempt to organize and systematize a testing effort will involve some input-space subdomain breakdown, however ill-defined or subjective the subdomains may be. So given that a program is to be tested at all, advice about subdomain testing comes down to what kind of subdomains might be used, and cautions about how things can go wrong. It would take a brave software development team to declare that their efforts need no testing—any human endeavor is prey to mistakes. But it is less necessary to test software whose quality is likely high because its development process is well defined, its development team experienced, and their previous work successful. A minimal testing effort should use functional subdomains. If the functions defining them can be closely related to usage, so much the better. For example, a published tutorial on how to use the software defines functional subdomains that are irresistible: they are a minimal cover for the software's functionality, and many users will almost certainly try them.

The decision to go beyond a low-cost functional test should not be lightly taken, and should be closely monitored. The danger is that the cost-benefit ratio may be very low. When subdomain testing exposes problems at a steady rate, then it should continue; when the rate falls off it is time to stop. (And at the same time, it is of first importance to look back on development and ask why a problem is being found at all.) Structural testing is particularly subject to becoming unproductive busywork. In a choice between extending functional testing or switching to structural testing, the latter is hard to justify. However, to use structural test coverage as an adequacy measure (see Section 3.2) is unexceptionable, if tool support is in place to minimize its cost.

Whenever subdomain testing is used, those involved should be alert to its fundamental limitation: no subdomain is really homogeneous, either in the 'sameness' that defines it, or in success/failure. When a test fails in a subdomain, well and good—the problem can be fixed if its frequency merits fixing. But when a subdomain test succeeds, it almost certainly means only that the subdomains aren't very good or the choice of test points in them was unlucky or both. Imagining how a success might be accidental or atypical is a way to improve subdomain testing.

References

[1] P. J. Boland, H. Singh, and B. Cukic. Comparing partition and random testing via majorization and schur functions. *IEEE Trans. on Soft. Eng.*, 29:88–94, January 2003.

[2] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proceedings of the 9th Asian Computing Science Conference, Lecture Notes in Computer Science*, volume 3321, pages 320–329. Springer-Verlag, 2004.

[3] T. Y. Chen and R. Merkel. An upper bound on software testing effectiveness. *ACM Trans. Softw. Eng. Methodol.*, 17(3):1–27, 2008.

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

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

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

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

[8] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. on Soft. Eng.*, 19(8):774–787, August 1993.

[9] Dick Hamlet. *Composing Software Components: A Software-testing Perspective*. Springer. To be published in 2010.

[10] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. on Soft. Eng.*, pages 279–289, 1977.

[11] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, June 1987.

[12] K. L. Heninger. Specifying software requirements for complex systems: new techniques and their applications. *IEEE Trans. on Soft. Eng.*, 6:2–13, 1980.

[13] C. A. R. Hoare. Monitors: an operating system structuring concept. *Comm. of the ACM*, 17(10):549–557, 1974.

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

[15] Brian Marick. *The Craft of Software Testing*. Prentice-Hall, 1995.

[16] Simeon Ntafos. On required element testing. *IEEE Trans. on Soft. Eng.*, 10:795–803, June 1984.

[17] T. J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. of the ACM*, 16:676–686, 1988.

[18] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Trans. on Soft. Eng.*, 11(12):1477–1490, December 1985.