When Only Random Testing Will Do

Dick Hamlet*
Portland State University
Portland, OR, USA
hamlet@cs.pdx.edu

ABSTRACT

In some circumstances, random testing methods are more practical than any alternative, because information is lacking to make reasonable systematic test-point choices. This paper examines some situations in which random testing is indicated and discusses issues and difficulties with conducting the random tests.

Category and Subject Descriptor: D.2.5 Software engineering, Testing and debugging
General Terms: Verification
Keywords: Testing theory, random vs. systematic testing

1. RANDOM VS. SYSTEMATIC TESTING

The conventional view sees random testing as a second-class alternative to its antithesis, so-called systematic testing. Systematic testing is preferred because it is directed, usually toward exposing failures. The proponents of systematic testing may admit that if one cannot come up with a purposeful testing plan then random testing is a fall-back possibility. They acknowledge that automatic test input generation in random testing is attractive, but only when an operational profile [**?**] is known, and only if there is an automatic oracle [**?**, for example] to judge the many test results that arise. Since both enabling conditions are problematic in practice, random testing's advantage is discounted. Random testing's undeniable strong suit is that a successful random test can predict a reliability bound [**?**] for the software being tested. However, this advantage is compromised by the infeasible number of test points necessary to predict ultrareliability [**?**]. Thus when a test manager chooses between test methods, random testing is always problematic, not competitive with (for example) specification-based functional testing that promises to find many show-stopping failures with a few use cases.

*Supported by NSF ITR grant CCR-0112654 and by an E.T.S. Walton grant from Science Foundation Ireland. Neither institution is in any way responsible for the statements made in this paper.

RT'06, July 20, 2006, Portland, ME, USA

However, there are situations where random testing must be chosen because systematic alternatives are the impractical ones. This paper explores some of those situations and suggests ways to maximize the benefit of a random test.

1.1 Subdomain-testing Methods

Almost all systematic testing is subdomain testing. In support of some testing goal (usually uncovering failures so they may be fixed) the program input space is divided into subdomains, and one or more test points are selected in each subdomain. In practice the number of subdomains must be small, so some subdomains are almost as large as the full input domain. It is often said that a subdomain is defined by its members being 'the same' in some sense. This 'sameness' is subjective and often misleading. It might be better to say that the tester can think of nothing of consequence that distinguishes between subdomain elements. That being the case, it is sensible to make the choice of test point(s) from a subdomain according to a uniform random distribution[1]. Thévenod-Fosse [**?**] calls this mixed test-selection strategy 'statistical testing'.

At the system level, the aforementioned specification-based functional testing is almost always the right choice. The subdomains are defined by cases recognized in the specification. If the specification is precise, these cases can be defined by formal predicates and it is possible to check that the subdomains are disjoint and exhaust the input domain. However, the practical value of functional-subdomain testing arises from an *imprecise* specification for the software, the existence of important use cases, and limited time or resources for testing. Untrained testers can execute these cases, and can use common sense and intuitive knowledge of the problem domain to judge the results. A few such cases go a long way toward finding problems that users will encounter immediately. It is hard to imagine any other testing method supplanting system-level functional testing.

At the unit level, functional testing is less useful, partly because specifications are often missing, but primarily because there can be no narrow use cases. The role a unit will later play in an assembled system might in principle be known at unit-test time, but in practice the unit tester is unaware of how it will be used. In component-based software development the units are components, and at component-test time

[1]The distribution cannot even be a non-uniform profile, because if one existed it would be a way to distinguish elements and to split the subdomain.

there may be no application system even envisaged. Thus for unit testing, 'systematic' testing takes on an additional meaning: *all* invocation possibilities for the unit must be explored, including many that will never arise in a given system.

Systematic structural code coverage testing comes into its own at unit level. Because unit code is compact and well understood (often the unit tester is the coder), tests can be sought that exercise each part of the code. For example, it is common to try to cover all branches by supplying a test data set some point of which makes each branch go each way, TRUE and FALSE [**?**]. The subdomains for branch testing are indexed by branch conditional and truth value; a test point x falls in (e.g.) the TRUE subdomain for branch b iff the program when executed on x passes through b and makes the conditional there TRUE.

The best use of tools that measure test coverage is as an adequacy metric, not for test generation [**?**, Chapter 20]. Tests should be generated by some other method, then coverage examined to see if any code element was missed. The original test is deemed adequate if the structural coverage is good[2]. Functional test generation augmented by tests to probe suspected code faults is the most used generation method.

1.2 Other Systematic Methods

Exhaustive testing is a subdomain method only in the limiting sense that each domain point is a singleton subdomain. Recently attention has focused on 'bounded exhaustive testing' (BET) [**?**, for an early example] [**?**], in which a finite portion of the domain (usually an initial portion in length order) is completely tested. BET has achieved some remarkable anecdotal successes, but has not yet received much theoretical study.

Spacing test points evenly throughout an input space is 'systematic' in the most intuitive sense. It could be considered a subdomain method (the subdomains being intervals) that includes a prescribed single-test-point choice (say the midpoint) in each subdomain. Equispaced sampling could also be described as random testing using a very peculiar distribution.

1.3 Random Testing

Random testing is literally the antithesis of subdomain testing: no points are considered 'the same' and the sampling is over the entire input domain. Its application to any system or unit depends only on a knowledge of the input domain and the ability to map pseudorandom numbers into that domain as a sampling device. However, to obtain a reliability estimate from random testing, an input distribution for expected software usage must be available. This distribution, the so-called 'operational profile' [**?**], weights inputs according to how likely they are to occur in use.

[2]It is necessary to point out that good feelings resulting from achieving high coverage when one was not trying to do so are no more than that: there is no theoretical evidence that code coverage, however attained, has a necessary relationship to any quality measure. The intuition behind this sad fact is that many coding/design mistakes are omissions, and one never fails to cover what is not there. If a test set fails to cover the code something may be wrong; when it does cover there is no corresponding significance.

At the system level, an operational profile is also called a 'user profile,' since it describes what a software user will do. Although user profiles are not commonly part of a software specification, they should be, because without a profile, reliability is not a meaningful idea. At system level, the definition of input domain is necessarily arbitrary and at the whim of the software's users. It must always be a design consideration that no input value is precluded, even when the 'user' is another program or a hardware device. Mistakes and malfunctions occur, and for each software system the designers must decide how much effort will be devoted to checking input values before using them. A user profile is no more than a fine-grained quantification of domain definition. The profile constitutes a tacit promise concerning which inputs will be provided and with what frequencies.

Things are different at the unit level. What input values arrive there is partly determined by user caprice at the system level, but the system architecture is also a filter. How a unit is placed within the system control structure determines what inputs it may receive. In the simplest example, a unit called by another cannot receive parameter values of a type other than that of the interface, and if the call is within a conditional statement the Boolean condition can make some parameter values impossible. There is no probabilistic aspect to this domain modification in a statically structured system.

If the system architecture is given along with a system operational profile, it is possible in principle to deduce a distribution that the unit will see within the system. For a simple extreme example of a distribution seen by a unit, there could be just two parameter values that can reach an embedded unit, and the system operational profile could make one of them much more likely. If just one parameter value could get through, then the unit profile is one of two: a unit spike at the given value or a pathological 'profile' with zero density everywhere, depending on whether the system profile as seen at the unit call site ever assigns the one possible value a non-zero probability. In any case, calculation of the input distribution a unit will see in place in a system is never done in practice, because it goes against the whole idea of unit testing early in the development cycle. It doesn't help that the practical calculation of the profile a unit inherits is technically difficult even given full information.

1.4 System- vs. Unit Testing

Intuitively, the goals of system testing differ from those of unit testing. At the unit level the testing problem is a necessary absence of usage information, so the tester hopes to be convinced that no failures exist anywhere in the complete domain, through an intimate understanding of the code. At the system level the testing problem is code- and problem complexity, but there is usage information that allows the most-used inputs to be tested.

Random testing in its usual application to predict a reliability bound makes sense only at the system level. Paradoxically, it is at the unit level where Section 2.1 suggests its use.

2. RANDOM TESTING THE METHOD OF CHOICE

Random testing seems a better choice than systematic testing in two general situations:

Sparse sampling. For a large, unstructured input domain—i.e., when a meaningful subdomain breakdown is not evident—it seems wrong to invent subdomains, each of which may be no more tractable than the full domain. Haphazardly picking a few points in such subdomains seems no different than haphazardly picking a few points from the whole domain[3]. This position receives theoretical backing from many studies (following the seminal work of Duran and Ntafos [**?**]) comparing random testing with subdomain testing in the abstract. The most intuitive statement favoring subdomain testing for detecting failures is that it works when there is a variation in failure rates among the subdomains[4]. Section 2.1 will consider examples where random sampling of a large, unstructured space is indicated, and the practical problems that arise.

Persistent state. Most software includes permanent storage that persists from one execution of a program to the next. Testing practice takes account of persistent state by forcing a program under test into a state, and once there, conducting a number of tests from that initial condition. Testing theory has most frequently ignored state. The usual theoretical assumption is that software is 'reset' between tests, so that results are repeatable. Theory and practice come together by imagining that state is just another input to be sampled. This view is technically incorrect, since the state dimension is not independent and hence a sampled state may never actually occur. The correct view is that each test is a sequence of inputs beginning from state initialization. In response to an input sequence, the software goes through a state sequence determined by its code, with a final output that is the test result at the end of the sequence. In Section 2.2 it will be argued that such sequences are best selected randomly.

2.1 Sparse Sampling of a (Sub)domain

Subdomain testing requires information to make a division of the input space plausible. Without this information, 'systematic' efforts are no more than a subjective comfort to the human tester, who may feel better using a 'method' even if it has no rational basis. The information needed to define sensible subdomains can come from a software specification or domain knowledge of the problem the software is to solve. But it can happen that the input space is immense and specification information lacking to divide it; then subdomain testing is contraindicated. Random testing still has two competitors: Equispaced (non-random) samples and bounded exhaustive testing (BET).

Random testing and equispaced sampling have the same enabling condition: the ability to map numerical values onto the input space. The choice of sampling method intuitively depends on how complex the inputs are and on how sparse the sampling must be. When an input value does not have a complex structure and the sampling density is high, there are advantages to each method. Equispaced samples are just that: the unsampled gaps are all the same size. For example, let the input domain for a single integer parameter be $[0, 100000) \subseteq \mathbb{Z}$. 1000 equispaced samples have a spacing of 100, while 1000 uniform random choices will leave gaps of maximum size on the order of 800 and select about five duplicate points[5]. But there may be an unexpected correlation between equispacing and an implementation mistake. For example, if a buffer is allocated in chunks of length 128, those 1000 equispaced samples will not hit a boundary value: ..., 100, 200, 300, ..., 500, 600, ..., 1000, 1100, ... ; whereas, 1000 random samples have at least a small chance (nearly 1%) of hitting at least one of the 781 special values 128, 256, 384, 512, ...

On the other hand, when the sampling frequency is low and the input space is complex, equispaced samples do not seem wise. The chance that the coverage explores one aspect of inputs but neglects another, and the balance between multiple dimensions, seem too difficult for proper equispaced exploration. A sequence of random choices seems safer. For example, imagine testing a stack implementation for its storage and retrieval of values. The test points are sequences of *push* and *pop* operations. The values pushed, the length of the sequence, and most of all, the intermixing of operations, are relevant parameters for test. With only a few tests, it seems better to choose all of these parameters randomly than to attempt to work out a coverage system. (This example is a special case of the sequence issue discussed in more detail in Section 2.2.)

It is more difficult to compare random testing with BET. Intuitively, BET finds failures because human loss of intellectual control on a problem or algorithm occurs early in the tree of interacting cases. People can't hold hundreds of possible interplay situations in their heads; BET can investigate millions exhaustively. BET makes no claim to establish technical reliability: there are many anecdotal or contrived pathological cases in which a problem occurs only outside the range of BET feasibility. Random testing explores the whole of an input domain, but of course it does far less well with the part on which BET is perfect. But whether BET is more effective than random testing seems a problem that requires empirical, not theoretical investigation. BET tries cases too complex to have been explicitly imagined by human implementors, and therefore often exposes failures resulting from not thinking of those cases. But when BET does *not* expose failures, it is hard to imagine any sound theoretical argument that there are none. In mutation testing, a similar situation was swept under the rug by adopting the 'competent programmer hypothesis (CPH)' [**?**] that if a programmer has not made mistakes that can be detected by trivial mutations, then he/she has made no mistakes. The BET analogy would be that if there are no failures in the BET range, there are none at all. The two situations share the feature that human beings cannot think through the trivial mutations (*resp.* all the BET tests). Still, no sound evidence

[3]It is unfortunate that in the *Hacker's Dictionary* [**?**] sense of 'random', these would be called "a few random points".

[4]This insight originated in reference [**?**], and is elegantly investigated using Schur functions in reference [**?**].

[5]Values obtained by trial, using the Perl `rand()` function.

or argument has been presented for the CPH.

The conditions that favor random testing are more likely to arise at the unit level, because unit testing is characterized by a lack of known structure on the input domain. Furthermore, since the parameters that define the input domain of a unit are supplied not by users but by other software, they are more likely to have complex structure. These are subjective assessments of which sparse sampling is best; reasonable testers will not always agree about what should be done. For persistent state the case is more compelling.

2.2 Sampling Persistent State

To make the situation as concrete as possible, imagine that a program under test utilizes a permanent disk file to record 'state' values. The program initializes this file should it not exist when execution begins; we assume that program behavior is repeatable from this 'reset' condition. After reset, the program may read and write its permanent file as a memory of past executions. Two somewhat different uses of state are important:

Remembering a 'mode'. By writing only a little information into its permanent file, a program can on one execution parametrize its behavior on subsequent executions. Often the mode is a block of information that 'personalizes' behavior. A user identification is recorded, along with a set of 'preferences' which that user has set. Web services universally keep this kind of mode state with the IP address or machine symbolic name as the identification. The effect of recording a mode is that a finite number of decisions within the program code are made without requiring input to make them on each execution.

Looking up information. A program's permanent file can be an extensive information repository, to which the program goes for data when it needs to respond to some input and which may be updated as well. A database is such a repository with a special structure and a set of processing aids (its query language). While 'mode' information is best thought of as a small collection that might have been supplied as input, repository information is best imagined as of unlimited size. Its values, rather than controlling a few program decisions, usually enter into the program's computations as data.

The character of persistent-state data is that its form and substance are usually narrowly specified. A set of preferences is not an arbitrary pattern, but rather encodes one particular binary vector. A database format is rigid and prescribed. Furthermore, as an entity entirely under program control, persistent storage is not usually checked by code for 'errors.' It is expected that needed constraints will have been observed. Finally, the 'legal' values of a state are very sparse in all possible bit patterns. An arbitrary file almost never meets the necessary constraints for a database, for example.

These characteristics of persistent state impact the testing of programs that use it. For example, it makes sense to exhaustively test the control patterns described by 'modes,' and branch or path coverage is a partial check on whether this has been done. A repository cannot be treated exhaustively; it must be sampled, and some kind of partitioning will be needed to structure the samples.

When a program has persistent state, there are two quite different ways for a tester to establish state values. The correct way is to begin at reset and supply a sequence of input values that cause the program to transition into some state s. Applying this prefix from reset on a number of different test cases then exercises the program from state s. It is a short cut to externally set the state value to s and test without the prefix. In our simple case of a permanent state file, setting the state amounts to creating this file with the proper contents.

The second, invalid, testing procedure treats the state like an input, setting it externally for each test. Program tests are then (input, state) value pairs, and these pairs can be sampled efficiently. For ease of reference, call this second way of testing "state-sampled." State-sampled testing is invalid when a state selected is infeasible; that is, when no sequence of inputs starting from reset exists to reach it. Infeasible states can arise from a specification mistake or an implementation failure. The converse may also occur: The program may be able to reach states that are specified as infeasible. Unfortunately, trying a state that should not occur can be worse than missing one that should occur: it isn't easy to identify a state as forbidden by specification, so a great deal of testing time is wasted exploring what actually can't happen. Like all such testing problems involving infeasibility, it is unsolvable in general to decide if a particular state should be infeasible whether or not it has been reached; and also unsolvable to determine if an unreached state is in fact infeasible.

2.2.1 An Example of Mode Behavior

For a concrete example, consider the following simple specification:

> The input is a floating-point value in the range $[-5, 10)$. It is required on domain $[0, 10)$ to compute either a $\sin$ or $\cos$ function shifted vertically by +2.0. The output may be clipped to the range $[1.4, 2.6]$, or damped by multiplying by the function $\lambda x[1/(x/3 + 0.5)]$, but not both clipped and damped. The choice of trigonometric function, clipping, or damping is determined by a 'preference' which is set by negative inputs as follows:
>
> | *Input* | *Setting* |
> |---|---|
> | -1 | Clip |
> | -2 | Damp |
> | -3 | $\sin$ or $\cos$ |
>
> Initially, the function is $\cos$, neither damped nor clipped. Each negative input reverses the appropriate choice. The preference remains in effect from run to run after it is initialized or changed.

This specification is deficient as natural-language ones usually are, particularly about what happens on inputs like -4.3. It is also somewhat unusual in not specifying the exact form of the state, only what the state must accomplish. It doesn't say what is returned for negative inputs or what to do in the case of 'error' inputs. Perhaps it is too concise in the way it forbids clipping and damping together. Nevertheless, the Perl code of Fig. **??** is the way many people would implement it (correctly, it is hoped). The programmer has decided