Theory of Software Testing With Persistent State

Dick Hamlet

***Abstract*—Software testing began as an empirical activity, and remains part of engineering practice without a widely accepted theoretical foundation. The overwhelming majority of test methods are designed to find software errors, termed faults, in program source code, but not to assess software operational quality. To go beyond fault-seeking requires a theory that relates static program properties to executions. In the 1970s and 1980s, Gerhart, Howden, and others developed a sound functional theory of program testing. Then Duran and others used this theory to precisely define the notions of random testing and operational reliability. In the Gerhart-Howden-Duran theory, a program's behavior is a pure input-output mapping. This paper extends the theory to include persistent state, by adding a state space to the input space, and a state mapping to a program's output mapping. The extended theory is significantly different because test states, unlike inputs, cannot be chosen arbitrarily. The theory is used to analyze state-based testing methods, to examine the practicality of reliability assessment, and to suggest experiments that would increase understanding of the statistical properties of software.**

***Index Terms*—Software testing, fundamental theory, persistent state.**

NOTATION

$\lvert \cdot \rvert$	Cardinality of a set
$\lceil \cdot \rceil$	Ceiling function
$[P]$	Output-function semantics for P
$\langle P \rangle$	State-function semantics for P
X^{∞}	Set of all finite sequences of values in X

ABBREVIATIONS AND ACRONYMS

GHD	Gerhart-Howden-Duran
MTTF	Mean Trials To Failure
MTBF	Mean Time Between Failures
ART	Adaptive Random Testing
FSCS-ART	Fixed-size Candidate-set ART
SARTE	State Adaptive Random Testing Extension

I. INTRODUCTION

SOFTWARE testing has no widely accepted theoretical basis that can be used to precisely analyze and evaluate empirical methods. Textbooks [1], [2] name methods and theories, but do not critically compare them. When you are an experienced tester who knows a good deal about a particular program and its specification, you just select interesting test scenarios, then follow your nose through the intricacies of execution experiments, and stop when you run out of time. This individualistic intuitive testing approach is often successful at exposing problems. However, as software comes to dominate more of daily life, a point is reached at which intuitive testing isn't enough, even when asserted by the best practitioner. The testing emphasis must shift from trying to make software fail to predicting that it will not fail. In short, testing must assess reliability. Precise analysis requires a theoretical foundation.

The explanatory power of a theory is enhanced by abstraction. Programs and their behavior are very complicated things, so unless most real detail is removed, one would not expect any grand theoretical insights. The Gerhart-Howden-Duran testing (GHD) theory, reviewed in Section II, treats program behavior as nothing more than an input-output mapping. It has been very successful, for example in studying the efficacy of random testing [3]. However, programs have a dimension the GHD theory does not capture: their behavior depends not only on input, but on a persistent external state, which in turn depends on past program behavior. For example, many programs in wide use are little more than front ends to database query-update processes. To test such a program by sampling its input space is misleading; its real behavior reflects mostly past database actions. Fortunately, GHD theory can be extended to programs with state, without compromising its simple, abstract quality. The concept of state in programming theory has several disparate meanings; here we have in mind something like the contents of a disk file external to the program.

This paper presents a natural extension of the GHD theory to include persistent state, and explores the consequences of the extended theory. The most surprising result is that state values cannot be arbitrarily sampled in testing a program. This means that basic ideas of random testing and reliability require re-thinking, and measures of state coverage are suspect. The extended theory is used to analyze existing test methods, and to propose a new one.

This paper is organized as follows. Section II reviews the input-only GHD testing theory for stateless software. Section III extends the theory by considering a space of persistent state values in addition to the input space; Section III.A presents the basic definitions. Section IV describes pitfalls of testing that takes state into account, and proposes a new state-aware testing method (SARTE). Finally, Section V explores wider issues of testing that include state, and its relationship to formal methods, to specification, and to engineering.

II. BACKGROUND OF GHD THEORY

This section summarizes the GHD software-testing theory developed in 1970 through 1980 in Gerhart and Goodenough [4], Howden [5], and Duran and Ntafos [6], including subsequent results and insights. The theory's abstraction is extreme in that program input-output functions play the only role. Given

Manuscript received December 30, 2013; revised October 27, 2014 and January 22, 2015; accepted February 03, 2015. Date of publication June 19, 2015; date of current version August 28, 2015. Associate Editor: C. Smidts.

The author is with the Department of Computer Science, Portland State University, Portland, OR 97207 USA (e-mail: hamlet@cs.pdx.edu).

Digital Object Identifier 10.1109/TR.2015.2436443

the code for an arbitrary program, it is an unsolvable problem to mechanically obtain or identify its input-output function.[1] The GHD functional theory is therefore seldom helpful in analyzing particular programs. The strength of its abstraction lies in investigating the properties of *all* programs, for which very general results have been obtained.[2]

This section begins with a discussion of program failure, captured in GHD theory as a program output that does not meet specifications. Section II.A considers whether there is always a root cause of failure in some program-source fault. Section II.B presents an alternate way of viewing the cause. Section II.C introduces the concept of reliability that depends on the notion of random test selection from the input space. Section II.D gives examples of the successes of GHD theory.

A. Faults and Failures

The business of engineering is the design and construction of artifacts, which are placed in the world to serve a practical purpose. Henry Petroski makes a strong case [8] that successful design results from analyzing failures. When an artifact fails, engineers look for a root cause and try to correct it. At its inception, each engineering discipline goes through a time of little understanding when failed designs are changed haphazardly, often without effect. Developing software is certainly a kind of engineering, which has led software engineers to adopt the model of failure correction so successful in other kinds of engineering. Unfortunately, software engineering failures often seem idiosyncratic, and fixes appear haphazard.[3]

For software, a failure is an event in the input space of a program. It is natural to seek the cause in the textual space of the program, where it seems a fault must lurk. Program faults have a long intuitive history, but this idea is not well defined,[4] and is sometimes counterproductive in dealing with failures.

For this informal presentation, it suffices that there is a *specification*, a statement of what software is supposed to do, which acts as an effective *oracle*. That is, given any input value x, and the corresponding software behavior of a program's input-output function, the oracle decides whether or not that behavior meets the specification.[5] If not, the software *fails* (at x).

Following the traditional engineering model, when a piece of software fails, its source code is studied to understand the failure and eliminate it. Describing this situation is linguistically tricky in that it is natural (but as this section shows, sometimes counterproductive) to attribute the failure to a textual fault in the source code, a small programming mistake that when found and corrected will eliminate the failure. The common name for the fixing process, *debugging*, also assumes that localized faults can be found and removed. Failures *do* sometimes result from textually localized mistakes, even typos, that intuitively qualify as software faults, but sometimes this mindset gets in the way.

Two examples of failures that are not the result of textual faults are incomplete specifications, and misunderstood specifications.

> **Incomplete specifications.** When some case is left out of a specification, its implementation can be expected to fail[6] when an input lies in the omitted case. An important example of incompleteness arises from the conjunction of events. Several situations (each handled in the specification and code) can occur simultaneously as a distinct case [9] not covered in the specification. Failure results because the code incorrectly treats this case as one of the simpler component events. To search for a software fault causing the failure is counterproductive because it will be sought in that part of the source code that happened to be invoked. There is no fault there; what's wrong is that code is missing.
> **Misunderstood specifications.** Human beings may misunderstand what software is supposed to do, even when specifications are expressed in formal language. The most pernicious form that misunderstanding takes is a divergence between specification and implementation that only a complex case exposes. Misunderstanding is almost certain to arise if a specification is vague and explained by examples. Searching the source code for faults may well find them, on the assumption that only a localized change is needed. But the needed fix is to discard the code as the wrong program, and reimplement with a better understanding of the specifications.

There are two mindsets for debugging, each with merit: 1) to blame source-code faults, or 2) to ignore the code and think about failures in a program's input space. The fault view of debugging necessarily concentrates on a single failure. One failure can be traced through the control flow of the failing program's code, and if along the way someone notices a mistake like a typo, it's a good candidate for a fault. But *the* fault, a localized mistake that when corrected will eliminate the failure and others like it, may not exist; and searching for typos is the wrong way to begin. The failure view of debugging instead concentrates on the input space, looking for properties that characterize the failing case. Paradoxically, the failure view comes into its own when a program has yet to fail. This discussion will be continued in Section II.D, after our context of reliability has been defined in Section II.C.

B. Failure Regions

As illustrated at the end of the previous section, faults sometimes poorly describe software failures. Instead, failures themselves can be connected to program code through the idea of a *failure region*.

[1]Indeed, even some very short programs defeat most attempts to understand their input-output behavior [7].

[2]The evident danger in abstraction is that it may remove too much detail, and thereby create a theory that disagrees with reality. For example, GHD theory ignores state, so it cannot describe the common situation in which a program obtains different results on two runs with the same input. Pure input-output functions do not behave in that way.

[3]It can be argued that the field will never mature because software engineering is unique in its absence of an underlying empirical science; see Section V.A.

[4]A mathematical entity is *well defined* iff it does not depend on parameters outside its definition, but instead is uniquely determined. The fault that causes failure f is not well defined because there are many textual changes that fit this description for any given f. Attempts to frame a definition that singles out one change as a best fix for the *real* fault have been unsuccessful.

[5]There is an immediate technical difficulty in the case that a program fails to halt on some input. Specifications are not usually thought to require non-halting behavior, and in any case there is no effective way to discover that the behavior is non-halting. It will be assumed that specifications never have undefined cases, and although programs may fail to halt, it is not the business of the specification to detect this problem, but only to judge program output values.

[6]Here the failure is not determined according to the specification, which requires nothing in the omitted case. The failure is judged by real requirements that the incomplete specification did not capture.

Frankl *et al.* ([10], Sections 2.1 and 2.2) describe the process of changing software to eliminate an observed failure, paraphrased as follows.

> Suppose that program P_0 fails on input $x_0 \in D$, where D is its input space.[7] By probing D in light of the discovery of x_0, other failure points may be found, perhaps ones easier to understand or describe, which may or may not be related to x_0. After a time, it is sensible to seek a fix for (perhaps some of) the failures that have been discovered. The person doing the fixing may subdivide them into groups that seem likely to respond to the same change. Let a change C_0 in the code text (creating a new program P_1) eliminate the failure at x_0, and perhaps other failure points comprising a set $F_1 \subseteq D$, but C_0 must not introduce any new failures. (That is, P_1 fails only on points where P_0 also fails.) Then define F_1 as the *failure region* of C_0. If C_0 does not fix a failure at x_1, that is, x_1 is a failure point of P_1 but $x_1 \notin F_1$, then x_1 is a basis for seeking another change C_1 to P_1, with its own failure region F_2. This process continues (suppose for M steps) until all the known failure points of P_0 have been eliminated. Thus the failure regions $F_1, F_2, \ldots, F_M$ are disjoint, and P_M is not known to fail.

The textual code changes $C_i, 0 \leq i < M$, may overlap, i.e., some parts of C_i may be altered in $C_j, j > i$. The set of failure points $X = \{x_i | 0 \leq i \leq M\}$ has no significance because it is discovered only accidentally. Finally, neither the failure regions nor the accumulated changes that make up P_M are well defined, because their definitions depend on the order in which failure points are chosen, and the changes chosen to try to eliminate those failures. The advantage of failure regions over textual faults is that they lie in the input space. Let F be the set of all failure points of P_0. The process of isolating failure regions can be described as moving points from F to $F_1, F_2, \ldots, F_M$. If $F = \cup_{i=1}^{M} F_i$, then P_M does not fail.

This formal description suggests a mindset for the successful tester, because the greatest difficulty in carrying out the process is in making changes that introduce no new failures. People responsible for the corrective maintenance of large software systems are well aware of this difficulty, and they try to make changes that perturb the existing design as little as possible. Their ideal change can be described as follows. If the software were to fail with this change, then it would have failed anyway. It may be tempting to make radical program changes to attack a stubborn failure, but it is not worth the sacrifice of steady progress.

Failure regions have been the subject of a few empirical studies, mostly attempting to establish that contiguous regions exist. Peter Bishop studied real software failures [11], and found regions that are contiguous blobs. Perhaps the most pernicious aspect of studying faults in the source code is that it has kept research attention away from the program input space. Some recent surprising results are described in Section II.E.

C. Random Testing and Reliability

The reliability of a mass-produced item is often estimated by *life testing*, in which K samples are each repeatedly tried until failure, say at trial N_i for sample $i, 1 \leq i \leq K$. The *mean trials to failure (MTTF)*[8] and standard deviation S can be estimated from life-testing data, with a confidence interval whose size is proportional to $1/\sqrt{K}$.

If we accept input sampling as a source of trials, something analogous to life testing can be carried out for software. The most important difference between physical-device life testing and software testing is that, in the latter, the statistical space is created by selecting a collection of input values. In principle, any value in the input space might be selected by a user (if only by mistake); but, in a practically infinite space, there must be some limitation to likely values if a significant sample is to be of manageable size. The idea of likely inputs is captured by the notion of a *user profile*, a probability distribution d over the input space. At input x, a profile value $d(x)$ is a probability that x will occur as an input when the software is used. Profiles acknowledge that, to users, some inputs are more important than others; a profile defines *operational testing* as having test points selected according to that profile. If the profile used in operational testing is even roughly correct, operational testing is far and away the best method for detecting software failures that matter [12], [13].

Precise, accurate profiles evidently do not exist for most software, because its many users cannot be expected to agree; indeed, a single user may have a different profile from day to day. John Musa treated the problem of an unknown user profile as a practical one of approximating the profile with a low-resolution histogram, obtained empirically by adding rough input probabilities to a program's specification. [14][9]

Approximately N test points can be selected randomly according to a profile expressed as a finite histogram of B bars. For each histogram bar of height u, select $\lceil uN \rceil$ uniform random points from the input set the bar represents. The total number of points selected is the sum of the selections for all histogram bars. The sum is within B of N, as each bar's sample count has been adjusted by less than 1.

Suppose then that P is in operation, and P is given a sequence of N_1 inputs X_1 selected randomly according to a user profile, $X_1 = \langle x_1, x_2, \ldots, x_{N_1} \rangle$. Let P fail only on x_{N_1}. Repeating this procedure K times gives a multiset $M = \{N_i | 1 \leq i \leq K\}$ of run counts to failure. Estimates of MTTF and standard deviation can be obtained from M. Confidence in the MTTF can be estimated using non-parametric methods.

It should be a high priority in software engineering to study the distribution of failure runs M for different kinds of software and testing methods. However, the only explicit case study found in the literature [10] is artificial, constructed by assuming a few simple failure regions.

For most physical devices, life testing is a practical procedure, because failures occur in times on the order of a day. Enough data can be obtained for an acceptable confidence interval. When a device does not fail quickly, it may be possible to accelerate its use. For example, a vinyl hinge might have a MTTF of many years in normal use, but it can be made to fail in minutes when flexed by a motorized jig.

Software testing is easy to accelerate because multiple executions can be started in parallel, and inputs can be supplied by driver programs at high speed, giving a typical acceleration of

[7]It is assumed that D arises from the problem that P_0 tries to solve, and remains the same when P_0 is changed.

[8]For artifacts in continuous operation, the similar parameter is MTBF.

[9]His idea worked well enough to qualify as a software-development best practice at Bell Labs.

about 1000 times. Thus life testing should also be practical for software. However, the real bottleneck in testing is not the test execution time, but the time required to select test cases and evaluate results. If either of these operations is manual, acceleration is lost, and failing test sequences are too long except in cases of extreme low reliability. Safety-critical applications are required to be *ultrareliable*, referring to an MTTF of better than 10^9 trials [15]. A popular PC application might have 20 million users (and smart-phone applications far surpass that). If each user makes five runs a day, and no more than one failure among all users once a month is acceptable, the required MTTF is about 3×10^9 trials. Thus, even with acceleration, billions of runs may be required before any failure would be seen in these programs, too many for practical life testing [15].

For seldom-failing artifacts, there is a statistical alternative to life testing. The no-failure case can be described by Bernoulli trials, in which each test execution results in either success or failure. The trials must be s-independent, and the probability of failure must be the same for each trial. The latter suggests that each bar of a user-profile histogram be treated separately. Consider a bar of the profile of height $u \leq 1$, which is the probability that a user selects a point in the corresponding input set U for a trial. If the fraction of failure points in U is f, $0 \leq f \leq 1$, then in n Bernoulli trials the probability p_k of seeing k failures is given by the Binomial distribution:

$$p_k = \binom{n}{k} f^k (1-f)^{n-k},$$
$$p_0 = (1-f)^n.$$

A test sequence weighted by the user profile is then a case of multiple Bernoulli trials, each with its own f_i and fraction of the trials u_i, where i ranges over the B histogram bars. The probability of seeing the multiple event in which there are no failures in any region is

$$\prod_{i=1}^{B} (1-f_i)^{\lceil uN \rceil},$$

where N is close to the total number of trials.[10] Because there are no observed failures in the trials, the only estimate of the failure probability is zero, and there can be no confidence interval. However, an upper confidence bound can be calculated, the likelihood that if the trials were repeated, the result on one repetition would be no failures. This bound is just the complement of the probability that no failures are seen, that is, the probability that some are seen:

$$1 - \prod_{i=1}^{B} (1-f_i)^{\lceil uN \rceil}.$$

For example, for one region,[11] the confidence in a failure probability of $1/N$ (that is, a MTTF the same as the number of trials) is about 50%:

$$1 - \left(1 - \frac{1}{N}\right)^N \approx 1 - \left(1 - N/N + \frac{N(N-1)}{2N^2}\right) = 1 - \frac{N-1}{2N}$$
$$= \frac{2N - N + 1}{2N} = \frac{N+1}{2N} \approx \frac{1}{2}.$$

[10]As explained previously for histogram-based sampling.

[11]In the case of a uniform profile, and all failure rates the same, $f_i = f$, the bound is $1 - (1-f)^N$.

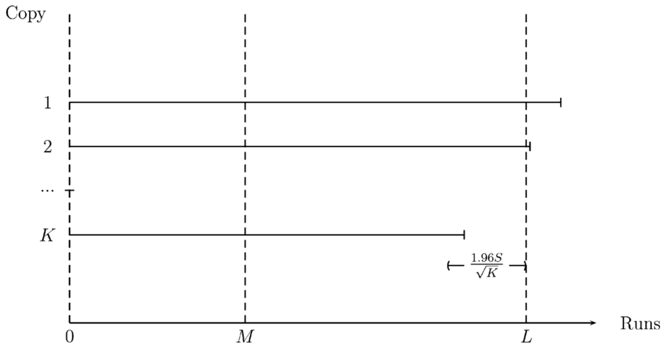

Fig. 1. Life testing vs. no-failure testing.

The approximation uses the binomial expansion, assuming $N \gg 1$, with three terms.

Life testing and testing without failure can be compared, because a life test that terminates before the first failure is observed is a no-failure test. For simplicity, consider a single-region profile. Let there be K copies of the software run in a life test, with MTTF L, and standard deviation S. The chance is about 95% that the MTTF lies in the $1.96S/\sqrt{K}$ confidence interval.

If each of the K run sequences stops at $M < L$ runs without failure, then the whole is equivalent to a no-failure run of length KM. The upper confidence bound on a MTTF of L or greater is $1-(1-1/L)^{KM}$. First, suppose that $KM \ll L$. Using the binomial expansion, the upper confidence bound is approximately $1 - 1 + KM/L \approx 0$, as would be expected. At the other extreme, suppose that $KM \geq L$. Even for large K, it may still be likely that each life test ends before failure occurs. For an upper confidence bound of 95%,

$$\begin{aligned} .95 &= 1-(1-1/L)^{KM} \\ \ln(.05) &= KM \ln(1-1/L) \\ &= KM(1/L - 1/(2L^2) + \ldots). \end{aligned}$$

Taking just the first term in the Taylor series, $KM \approx L\ln(.05) \approx 3L$.

In summary, estimates of a minimum software MTTF L can be obtained with 95% confidence in life testing by increasing the number of samples K. But if initial segments of the same runs are used as a no-failure test of M runs, $KM \approx 3L$ trials suffices for an upper confidence bound of 95%. The life test requires about KL trials, and K is usually an order of magnitude larger than 3. Fig. 1 illustrates the comparison between confidence intervals and upper-confidence bounds. In the figure, a life-test of K copies of software is shown, ending in failures distributed around MTTF L, with standard deviation S. A 95% confidence interval of size $1.96S/\sqrt{K}$ is shown below the Kth copy. M is chosen so that none of the copies has a failure in M runs, but $KM > 3L$, making the upper confidence bound that the actual MTTF exceeds L about 95%.

The role of a user profile is often dismissed by incorporating a profile d in software reliability theory but ignoring the practical fact that results depend crucially on d, yet it may be a poor approximation. For a simple example, suppose a no-failure test comprising 30,000 points is conducted for a program P using a uniform profile. The Bernoulli-trial formula gives a 95% upper-confidence bound on the failure rate being below 0.0001. But if the actual profile has two regions, one weighted 0.99, and $1/4$ the size of the other weighted 0.01, then the 24,000 uniform test points in the latter contribute almost nothing, while the 6000 in the former only give 95% confidence in a failure-rate bound five times as large (0.0005). Evidently, using different profiles can lead to arbitrarily large changes in the theory's predictions.

Probability models are mathematics, whose theorems are not subject to experimental error. However, their application to programs requires justification. Tossing a coin is believed to be a binomial process only because no mechanism has been suggested for its success rate to vary, and many experiments match a binomial distribution. Fundamental statistical properties of software are seldom studied, theoretically or experimentally. At issue is whether a uniform random selection of input points are each Bernoulli trials. In a unique study [11], Bishop observed that they are not: the failure probability changes as tests are selected.

D. Successful Applications of the GHD Theory

GHD theory is the setting for a body of research that precisely compares the efficacy of testing methods. Joe Duran and Simeon Ntafos started this research line in the 1980s in a paper [6] that called for a new appreciation of random testing. They compared uniform random test selection with selection based on covering subdivisions of the input space, and found them not very different in their ability to detect failures.[12]

Merkel and Chen went much farther, suggesting that no testing method can detect failures much better than uniform random test selection can [17]. For example, using the F-measure, the average number of tests required to detect the first failure, they showed that no other method is more than twice as good as random testing.[13]

Another promising research direction defines a class of new testing methods, motivated by searching for contiguous failure regions. So-called adaptive random testing (ART) [18] comprises methods in which random test selection is distorted to favor some kind of coverage of, or diversity in, the input space. For example, in Fixed-size Candidate-set ART (FSCS-ART) [19], the distortion favors points farther from those already selected. Many different distortion schemes have been investigated, each with an improvement in (for example) the F-measure. In fact, the improvement approaches the theoretical maximum predicted by Merkel and Chen [17], suggesting that ART may be the best-possible testing method.

Finally, GHD theory is a good setting for the discussion begun in Section II.A about when it is appropriate to consider software failure as caused by a localized fault. The Apple SSL goto bug is an instructive example [20]. Once it was discovered that there *was* a failure in applying the security protocol, conventional debugging by tracing source-code control flow found a spurious unconditional transfer statement that skipped necessary code, clearly a software fault. But why did it take so long to notice the failure? More important, what assurance is there that other failures are not waiting to happen? Here, software faults are of no use, because there is no theoretical link between the code and the input space. In terms of GHD theory, one cannot estimate a program's reliability from its source code.

It is difficult to imagine obtaining such general results in any theory less abstract than GHD.

E. A Summary of Stateless Testing Theory

The pursuit of software faults is a worthy endeavor. But almost everything that can be said about software faults can be expressed in terms of failures and failure regions.[14] A measure of testing effectiveness, with which one that can analyze and compare empirical methods, must be soundly connected to software failure. As demonstrated in Section II.A, faults and failures cannot always be put into correspondence. Nor is a count of fixes for failure regions an adequate measure; all too often a change fixes nothing, or fixes failures too infrequent to matter.

Turning to probabilistic ideas, conventional reliability engineering can be applied to software, and its MTTF parameter (also called the *F-measure* in testing theory) with confidence bounds is a good candidate for a quality measure. Life testing can be used only if the MTTF is small. Bernoulli trials require fewer test executions, and always assign a confidence bound to any MTTF, albeit a large MTTF and high confidence together are impractical. Unfortunately, for the predictions of these measures to hold, random sampling from the input space must match the operational distribution under which the software will be used. At best, the user profile can be approximated by a histogram over a handful of sets that divide the input space.

In the next section, it will be a straightforward exercise to add a state space to the definition of program execution (Section III.A). However, it is not obvious that deep results continue to hold in the extended theory. Indeed, in the presence of state, it will emerge that fundamental ideas like correctness and reliability are difficult to define. The idea of random test selection itself will require re-examination.

III. Persistent-State Theory

The GHD theory of programs as pure input-output functions will be extended to include a state space in addition to the input space. A program may use this space to write values on one run, which it may read on a subsequent run. That is, on a sequence of inputs, the program's behavior may depend not only on input values, but on state values created earlier in the sequence. In more concrete terms, a program writes to an external medium on one input, then reads what it wrote on a subsequent input. Values stored in a permanent disk file to which the program has create-, write-, and read-access are a good intuitive model of such states.

There are other, quite different ideas of state in programming, although all of them refer to some kind of memory that persists during or beyond the execution of code. In subdisciplines of computer science, the word state has its precise technical meaning, but research results are often presented as if the concept were universal, leading to confusion. The proponents

[12]Their article launched a cottage industry in refining and sharpening the theoretical comparison of random testing with other methods, which is still producing results [16].

[13]Merkel and Chen require a number of strong assumptions to carry out a complex proof based on analysis of the geometric shapes of failure regions, so their result is only suggestive.

[14]Except, of course, an ill-defined location of *the fault* in the code.

of functional programming decry use of state altogether. The state they mean is the vector of variable values (including the program counter) that describes program execution in the hardware, step by step. Our state omits all of this detail (the program counter in particular). Instead, our state values are obtained at the beginning of a run, and saved at the end, just as in an input-output function the input occurs at the beginning, and the output at the end of a run. Finite-state automata (FSMs) involve a finite number of states as a fundamental idea. An FSM stores and examines a value with a transition to different states depending on that value; then different actions can take place starting in each state. The FSM memory from input symbol to input symbol lies in patterns of its defining states. This trick works only with a finite number of input values and states. In contrast, a conventional program such as we imagine will store a state value with a disk write, and subsequently read it back for processing with conditional statements. The conventional code does not change with the number of state values, which is naturally thought of as unlimited.

In the software testing literature, our idea of state is neglected. For example, in Mathur's meticulous 697 page textbook *Fundamentals of Software Testing* [2], the word *state* appears in neither the index nor the table of contents.[15] There are, however, competing theories of programs with persistent state, extending FSM theory to unlimited state sets [21]. These theories do describe ideas similar to our idea of persistent state, but unlike the extension of Gerhart-Howden-Duran theory, they usually model computation with a detailed machine. A machine model makes them useful in specification, design, and analysis of particular programs,[16] but less useful for expressing properties of programs in general.[17]

In the remainder of this paper, the unqualified word *state* means a value in the state space S defined in Section III.A.

This section extends Gerhart-Howden-Duran theory to take account of persistent state.[18] The mathematical theory that begins with these definitions, and describes their consequences is this paper's major contribution. The extension is straightforward: state can be included by considering sequences of runs instead of single runs (Section III.A). In extending random testing to sequences, a theory would be expected to include precise definitions of state coverage and state sampling, which are extensions of important input-space ideas. However, the theory instead will show that state sampling is inherently complicated and indirect (Sections III.B and IV.A).

A. *Theory of Program Execution Including State*

Consider a program P with an input and output space D of single values, and a state space S of single values. P's behavior on a run depends on values from $D\times S$, and P may create values in S.[19] By definition, the semantics of P are described by two semantic mappings[20]: its *input-output mapping*, symbolized by $[P]$, maps inputs and states to outputs

$$[P] : D \times S \to D;$$

and its *state mapping*, symbolized by $\langle P\rangle$, maps inputs and states to states

$$\langle P\rangle : D \times S \to S.$$

$D \times S$ is not necessarily the domain of these mappings; either or both may be undefined at points of $D \times S$ for a particular P.

There is general agreement on the ways in which a program's input-output mapping can be undefined for a run: for a value $(x,s) \in (D \times S)$, P may not terminate, or may terminate without any output. Failure to terminate also makes the state mapping undefined, but capturing the proper intuition about failure to write a state requires more care. It is a good mental model to think of the state set as the contents of a permanent disk file available to the program, but with external existence. Undefined state means that no state value is available; i.e., the permanent file does not exist. This serves a crucial purpose because it is the reset condition for the state, which the program can detect, and to which it can respond by initializing, e.g., by creating the permanent file. When a program does not set the state, its state mapping is not undefined. The output state value is by definition the same as the input state value.

There is another important way in which a program's semantic functions are undefined: state values may be unreachable, as described in Section III.B to follow.

Because the undefined state acts like a state value, let each program P have an *initial state* $s_0 \in S$ that begins each sequence of runs. If $\langle P\rangle(x,s) = s_0$, we say that P has *reset* its state. The behavior of P is captured by the definition of an *execution sequence*. Let $X = x_0, x_1, \ldots, x_{N-1}$, where $x_i \in D$, for $0 \le i \le N-1$, be any sequence of input values of length N. Supplying X to P defines a sequence of runs, and a corresponding sequence of states $s_0, s_1, \ldots, s_{N-1}, s_i \in S, 0 \le i < N$, where

$$\begin{aligned} s_1 &= \langle P\rangle(x_0, s_0), \\ s_2 &= \langle P\rangle(x_1, s_1), \\ &\ldots, \\ s_{N-1} &= \langle P\rangle(x_{N-2}, s_{N-2}), \end{aligned}$$

and the execution sequence is

$$(x_0, s_0), (x_1, s_1), \ldots, (x_{N-1}, s_{N-1}), \tag{1}$$

with output $[P](x_{N-1}, s_{N-1})$. Each initial segment of an execution sequence is also an execution sequence. No execution sequence or output of a sequence can involve points of $D \times S$ where either semantic mapping of P is undefined.

[15]Mathur makes a few off-hand mentions of what might be our state. He has a definitive chapter on testing finite-state machines, but the properties of FSMs are significantly different from those of conventional programs.

[16]Section IV.A to follow uses such a model as a specification for a microwave-oven control program.

[17]The same remarks apply to the stateless case: stateless programs have many detailed models, e.g., using an imperative programming language, but the model detail is a hindrance in seeking general results like those in Section II.D.

[18]Preliminary versions of the definitions presented here have appeared in published papers [22], [23] and most recently in a monograph [24]. In the cited works, the subject is an implementation of component-based software development. The components have state, but no theory of state is developed.

[19]Multi-valued spaces and distinguishing an input from an output space would complicate the notation, but would not change the results.

[20]S. C. Kleene was the first to use this kind of notation [25]; Harlan Mills's box functions [26] are a variant.

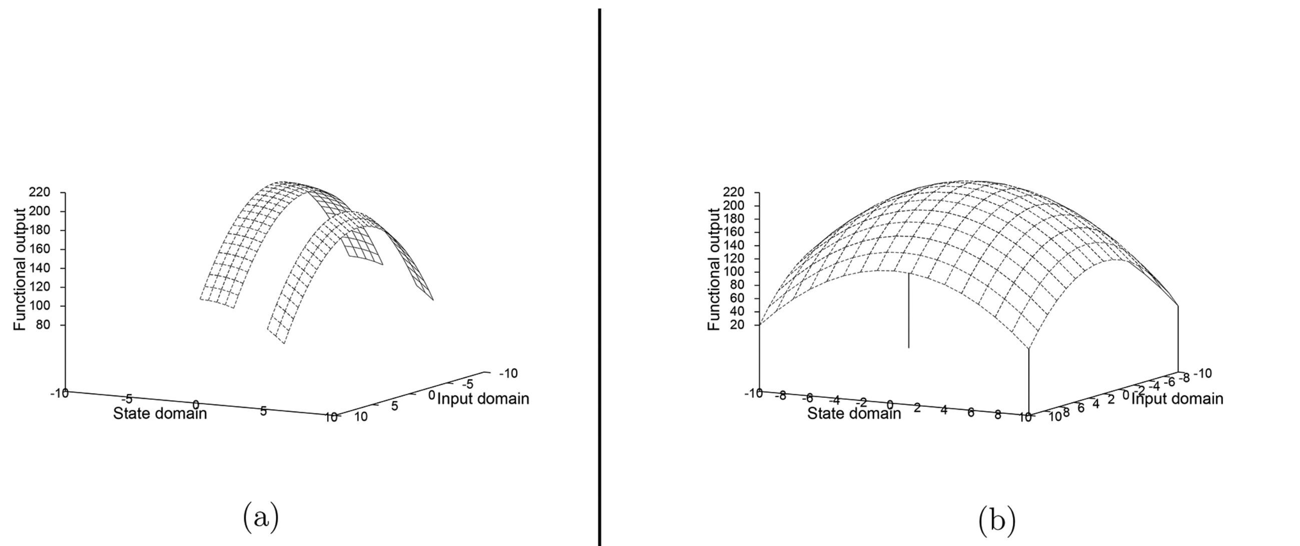

Fig. 2. Actual and erroneous behavior of a program $E1$. (a) Actual output behavior. (b) Behavior treating state as an input.

The definition (1) makes clear that the choice of inputs is arbitrary, but states are computed, not chosen by any outside agent. The definition also permits P to behave improperly. P may fail to compute some output or state on input (x, s) because $[P](x, s)$ or $\langle P\rangle(x, s)$ is undefined.

Software specification and software correctness must be defined for input sequences. A *specification relation* R is a set of pairs, $R \subseteq D^\infty \times D$. $(X, y) \in R$ means that, according to R, the output y is an acceptable response to the input sequence X. A specification R must be algorithmically defined everywhere on D^∞, so that it serves as an oracle to effectively decide if any (input, output) pair is acceptable. Specifications need not be functions; more than one output may be acceptable for the same input.

A program P is *correct* (with respect to specification R) iff the output of P on input X is acceptable for every $X \in D^\infty$. P *fails* on input sequence X iff there is an initial sequence of X whose output is not acceptable to R. According to this definition, a program cannot be correct if it fails anywhere within an execution sequence, even if the output of the whole sequence is correct.

As defined, a specification relation makes no mention of state, and failure does not involve states except indirectly. Section V.C to follow considers an alternate definition that makes more direct use of the state mapping $\langle P\rangle$.

B. Unreachable States

It is typical of programs P with persistent state values drawn from a set S that only a minuscule part of S enters into actual execution sequences of P. For $s \in S$ to occur, it must be either $s = s_0$, or must be $s = \langle P\rangle(x, s')$ for some $x \in D$ and some $s' \in S$ that itself occurs. Define the *reachable* states of P as those that occur in its execution sequences. There is no similar idea for input values because there is no way to constrain a user (or a tester) from selecting any input value in D. It is an unsolvable problem to determine the reachable states of an arbitrary program, or even to determine if some given state is reachable. The existence of unreachable states fundamentally alters program testing because the state space cannot be sampled as if it were an additional input dimension.[21] $[P]$ and $\langle P\rangle$ are defined only for reachable states. If an arbitrary state-space value is selected, it may be an unreachable state, and therefore irrelevant in testing P.

For example, Fig. 2 shows a portion of the output graph for a program $E1$ with

$$[E1](x, s) = 220 - (x^2 + s^2),$$

and

$$\langle E1\rangle(x, s) = \begin{cases} x & \text{if } 0 \le x < 2.5 \text{ or } 5 \le x < 6.25 \\ s & \text{otherwise} \end{cases}.$$

(The functions chosen have no significance other than to create Fig. 2.) Fig. 2(a) displays the actual behavior restricted to reachable state values; (b) displays what would appear to happen if the state is treated as an input. Among other spurious features, Fig. 2(b) suggests that a tester examine negative state values, which actually never occur.

Fig. 2(a) illustrates a general feature of unreachable state values: in the (input $\times$ state) space, unreachable states form bands on the output surface bounded by planes of constant state across the input dimension. In the figure, there is such a band of width 2.5 between state values 2.5 and 5. It is impossible for the output surface to have unreachable state islands or peninsulas with coastlines crossing these lines of constant state in the input dimension. That is, if a state is unreachable for one input, it is unreachable for all inputs.

C. An Example of a Program With State

To illustrate the definitions of Section III.A requires a nontrivial example, a control program for a simple microwave oven. The example will also be used to illustrate pitfalls of specification-based state testing in Section IV.A.

This oven has three buttons that signal $(\mathbf{X})$ when touched:
AddTen $(\mathbf{A})$,
Start $(\mathbf{B})$, and
Stop or Clear $(\mathbf{E})$.

[21]Direct state sampling is the accepted practice in reliability-growth modeling [27].

There is a signal $(\mathbf{D})$ when the door is opened or closed.

It has a seconds timer that can be read, set to zero, augmented, started counting down, or stopped. When the timer counts down to zero, it signals $(\mathbf{T})$.

The magnetron can be turned on or off.

There is a seconds clock that counts up from 1.

When the unit is powered up, the timer is stopped at zero and the magnetron is off. If the door is closed, the software is started and the clock is started at 1. There is no initial (D) signal if the door is open at power up, then closed.

Although real microwave ovens are not this simple, this one would be usable. The functionality has been minimized to shorten the description. Users are expected to open the door, place something inside, close the door, push the AddTen button until the necessary time has been accumulated, then push the Start button. When the magnetron stops, the operation is done. Operation can be terminated early by pushing Stop or Clear, or opening the door.

A program P to control the microwave will be given in pseudocode of an interrupt-servicing language with blocks of code labeled by the signal that invokes them. This language has global integer variables with all-cap identifiers, which are initialized to zero when the program is first run, and thereafter retain their values from one signal to the next. The special read-only variable `CLOCK` is set by the hardware to the current clock value. `OUT` is another special variable whose value is taken as the output of P. The language can issue commands to the hardware, shown as lower-case in pseudocode.

```
A: {
   add 10 to timer
   }
B: {
   start timer countdown and start
magnetron
   RUN ← CLOCK
   STOP ← 0
   }
E: {
   stop magnetron
   IF STOP
   THEN
     set timer to zero ← 0
     STOP ← 0
   ELSE
     STOP ← 1
     OUT ← OUT + CLOCK - RUN
   FI
   RUN ← 0
   stop timer and magnetron
   }
T: {
   STOP ← 0
   OUT ← OUT + CLOCK - RUN
   stop timer and magnetron RUN ← 0
   }
D: {
   IF RUN
   THEN
     stop timer and magnetron
     OUT ← OUT + CLOCK - RUN
     RUN ← 0
   FI
   }
```

The input space D for P comprises the events $\mathbf{A}\ \mathbf{B}\ \mathbf{E}\ \mathbf{D}\ \mathbf{T}$; the output is the value of `OUT`, which should be the total time the magnetron has run since the software was started.

The state space S of P is a three-tuple of values of its variables, `OUT`, `RUN`, and `STOP`, (o, r, s).

Values of $[P]$ and $\langle P\rangle$ can be obtained from the code. For example, immediately following startup, $[P](\mathbf{A}, (0,0,0)) = 0$, and $\langle P\rangle(\mathbf{A}, (0,0,0)) = (0,0,0)$.

Following power-up, the input sequence $U = \mathbf{AADDBT}$ describes the operation of cooking something (placed in the oven between the two $\mathbf{D}$ events) for 20 seconds[22]. The time between signals is arbitrary (except before $\mathbf{T}$, which the user cannot control), so it is taken to be zero. Then the corresponding state sequence is (0,0 0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,1,1), (20,0,0).

The clock is at value 1 second (as initialized by power-up) in the first six states; its value is 21 seconds in the final state. The output from U is 20, which is correct according to an informal understanding of what the microwave should do.

To describe an asynchronous event H occurring in a sequence of inputs, the time since the previous event is shown in curly brackets preceding H. The previous convention of events spaced at 0 would mean that, when times are omitted, they are 0, except for the $\mathbf{T}$ event. Following power-up, the input sequence $U' = \mathbf{D\ D\ A\ B\ \{3\}D\ D\ B\ \{2\}\ E\ D}$ might result if food is placed in the oven between the first pair of $\mathbf{D}$ events; between the second pair of $\mathbf{D}$ events it is examined, then the microwave restarted; with five seconds remaining, the microwave is stopped, and the food removed, leaving the door open. The corresponding state sequence is (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,1,0), (3,0,0), (3,0,0), (3,5,0), (5,0,1), (5,0,1). $[P](\mathbf{D}, (5,0,1)) = 5$ is the output at the end of this execution sequence.

D. Failure Regions, Random Testing, and Reliability in the Presence of State

Each idea in the stateless GHD theory of Section II has a counterpart in the extended theory of Section III, obtained by replacing the word *input* with *input sequence*, and *output* with *output of the execution sequence*. Unfortunately, the extended ideas are more difficult to grasp, and it is intuitively unsatisfactory that the state enters only indirectly as part of execution sequences. For example, the extended notion of failure region is a set of input sequences, when intuitively a set in $D \times S$ would be more natural; but the possibility of unreachable states makes a more intuitive extension impossible.

There is, however, an intuitive precursor to failure that our formal definitions can clarify: the notion of *state error* as used in the software dependability community [1]. Although the latest IEEE glossary [28] defines *error* as almost any kind of mistake, and does not mention state, the dependability community gives it the narrow meaning of an erroneous state[23] that arises in execution, which may lead to a later failure. The theory presented in Section III.A can capture and analyze this idea. Suppose that P does not fail on some input sequence X, yet X has a shortest initial subsequence Z that can be extended to a different input sequence Y on which P does fail. Intuitively, the execution sequence corresponding to Z ends in an error state that sets up a subsequent failure on input Y. Let the execution sequence

[22]The user must have forgotten to take it out, because U does not end with a $\mathbf{D}$ event.

[23]This state probably refers to a vector of values of all program variables.

that results from input sequence Z end with the (input, state) pair (x', s'). The next state will be $s = \langle P\rangle(x', s')$. Suppose as the simplest case that X adds only input x to Z, and Y adds only y. Then $[P](x, s)$ is correct, but $[P](y, s)$ is wrong; the idea of an error state would be that $s = \langle P\rangle(x', s')$ is wrong. Indeed, changing $\langle P\rangle$ might fix the failure on input Y, but so might changing $[P]$. The flaw in the state-error idea is its tacit assumption that there is a state specification, so that states like $\langle P\rangle(x', s')$ themselves can be wrong.

The probabilistic ideas of random testing and reliability also have straightforward extensions to input sequences, but now these ideas are more difficult to define so that they capture common intuition. The fundamental difficulty for random testing is that the selection space (input sequences) is multidimensional.[24]

The testing literature often describes test selection loosely using the word *random* as an antonym for the word *systematic*. The idea is to contrast a deterministic, algorithmic selection method with unsystematic choices. Duran and Ntafos [6] established the value of this approach by showing that a wide class of systematic methods are no better than pseudorandom selection. When a space X is discrete and infinite, *random* choice can be defined by mapping X one-to-one to a rational interval from which uniform pseudorandom choices are made. The mapping is left vague; of course its details are crucial in establishing whether or not these so-called random choices are intuitively without systematic bias. The meaning of the unqualified word *random* in this paper follows the somewhat imprecise testing literature.

A random input sequence in D^∞ can be obtained by making successive random choices from D, with a random choice of how many choices. But the states that arise from such input sequences are mostly of little interest; most of them describe trivial use cases of the software [30] (and these states appear over and over for different sequences). It would be more satisfactory to make random choices in $D \times S$ by selection in each space. But no direct selection from S is possible because of the unsolvable reachability problem for S (Section III.B).

It is possible to enumerate the reachable states, by any method of choosing distinct input sequences. In practice, the difficulties will be that the same trivial state values arise over and over, and there is no bound on the input-sequence length needed to reach some particular state. For each integer m, a subset $S_m \subseteq S$ of reachable states can be generated by trying all input sequences of length m or less. If D is not finite, then input values in these sequences will have to be arbitrarily selected, say randomly. Any S_m may omit important reachable states, either because m is too small, or because of poor choice(s) in an infinite D. Nevertheless, a test point with some claim to be unbiased[25] and unsystematic can be selected by randomly choosing m, then randomly choosing $x \in D$ and $s \in S_m$. This selection satisfies the intuition that the choice from $(X \times S)$ ranges over non-trivial states. Because the definition of correctness uses only input sequences, in this test selection procedure it will be necessary to retain an input sequence along with each state s in S_m, say the shortest sequence that reached s in constructing S_m. Let this sequence be c_s. Then, to decide if a point (x, s) selected from (D, S_m) has failed, add x to the sequence c_s, and use the specification for that sequence.

Reliability in the presence of state presupposes a profile, which is all the more important because of the huge sets to be sampled. Musa's empirical usage histograms enabled conventional definitions of reliability to apply to software, but estimation of a plausible profile for input sequences seems much more difficult. Empirical profiles and the definition of random selection in an (input $\times$ state) space are intertwined. It is paradoxical that, because profiles are difficult to define, precision in the definition of random selection may not be important. For example, the procedure above using an enumeration of reachable states may be intuitively wanting; but if users were able to assign usage probabilities to subsets of this space, it could be used to define operational testing, and thus a kind of intuitive reliability. Section IV explores practical schemes of this kind.

Attempts to define and use profiles with a large discrete state space, such as a database, suggest that it may be futile to seek a measurable notion of reliability. No matter which space is chosen for sampling, it will be so large and unstructured that any practical number of samples must fail to represent it.[26] Any piece of software in extensive use does have a MTTF that can be measured by keeping track of its failures in live operation. The process generating failures may not be stable, so the MTTF can poorly predict reliability. In fact, do software systems with large state spaces behave in this unpleasant way? Despite the prevalence of these systems, for example, in on-line Internet marketing, there are no published data; fundamental studies have not been done.

E. Summary of Testing Theory With State

It is straightforward to extend the Gerhart-Howden-Duran theory to include the state behavior of a program P by defining two semantic functions $[P]$ and $\langle P\rangle$ that map the (input $\times$ state) space to outputs and states, respectively. Particular state values may be unreachable; hence states cannot be selected arbitrarily for test. A sequence of input values is the starting point for program behavior; the sequence of state values follows, and finally the output value. Specifications are thus relations defined on the power set of the input space and the output space. There is no intuitively acceptable definition of random input selection for sequences, and even a crude approximation to a sequence profile is unlikely to exist.

IV. Practical Testing With Persistent State

The difficulty in testing any significant piece of software is the overwhelming size of the space(s) from which test points can be drawn. Testing is commonly thought to have two purposes: 1) to expose software failures so that they may be fixed, and 2) to

[24]In the stateless case, similar difficulties occur with non-numeric input types such as strings, but these are usually ignored in the testing literature. In a few important practical cases, tools have been developed to generate inputs from a multidimensional space. For example, the Csmith system [29] is a sophisticated test-case generator for C compilers. But testing with Csmith is selection from its arcane collection of programs devised by hand to exercise compilers. Csmith makes many random choices, aided by a partial grammar of C, but there is no critical discussion of randomness over the input space, and reliability of the software is not considered.

[25]The bias always remains that states never selected are hard to reach.

[26]Section V.D begins a discussion. A more optimistic view is presented in Section IV.C.

assess software quality using some measure of reliability. Neither purpose is served if a few samples are drawn haphazardly from a huge space; unless the MTTF is near zero, no failures are seen, and nothing is learned about the software. Debugging methods will expose some failures for correction, but to improve reliability requires connecting a software fault to the operational frequency of its failure. For stateless programs with narrow histogram profiles, approximate reliability assessment is possible, as described in Section II.C. When state is added to the theory, the testing space of input sequences is vast, and profiles have no intuitive approximation. The test engineer is left without a valid sampling strategy. This section uses the framework of extended GHD theory to analyze a common testing strategy, and suggest new strategies. These applications of extended GHD theory provide insights that are the secondary contribution of the paper. It is important to be wary of confusing testing *effort* with *effectiveness*: plausible but erroneous methods (e.g., Section IV.A) may do little more than keep test engineers busy.

A. Analysis of State-Coverage Methods

Program specifications may include the definition of state processing as the clearest way to describe requirements involving past behavior. For example, a *state-machine specification* [31] is a natural way to express requirements. In the simplest case, a state machine is a formal finite-state transducer (FSM). However, usually state-machine states[27] involve some arbitrary data storage that makes the specification-state set infinite. The expressive power of an infinite-state machine is most evident when its states can be partitioned into a small number of equivalence classes, each of which is conceptually a state with a parameter. A state machine is defined by its transition mapping T, which maps pairs (input, state) to pairs (output, state). To find a specified output value on input (sequence) $X \in D^\infty$, one takes the D values from X in order, and starting in a distinguished initial state, constructs successive (specification) states with T. The specified output value corresponding to X is the last output in the result sequence given by T. Thus the specification relation defined by a state machine is a function, very similar in form to the program semantics defined in Section III.A.[28]

T may be defined by a finite directed graph whose nodes represent states with parameter values, and edge transitions between states labeled by the inputs that cause them and the required actions when they occur. The graph can be used to construct input sequences. For example, a traversal of the graph will yield an input sequence that reaches every specified state (ignoring its parameter). When the state machine is not an FSM, the state parameter values may be used in the edge labels and nodes. For example, in a transition out of a state with a parameter value t, t may influence the transition.

The similarity of a state-machine specification to a program is both a strength and a weakness. The specification is easy to understand, but it is also easy to confuse with an implementation. The implementer of a state-machine specification may choose to ignore the details of the specified transition mapping T except insofar as it dictates what outputs are required.[29] Whatever the implementor's intentions, they may not be realized. It is a fundamental testing mistake to assume without verification that some specified or designed action actually occurs in a program under test. A tester who uses an input sequence derived from the specification can easily make this mistake as follows.

> As inputs are supplied to a program starting in its initial state, its state-machine specification mapping T supplies a sequence of specified result states, and specified output values, against which the program output can be checked. The tester's mental model of program behavior moves from state to state according to T, covering these specification states.

But in reality, nothing is learned about the actual program's execution sequences. It may use completely different states described in its design; in any case, the program may have arbitrary mistaken deviations from specification or design. Thus, although specification states can be explored to obtain input sequences, those sequences have an unknown significance, and may mislead the tester.

The specification mapping T is often used to define test-coverage metrics, such as covering all states, all state transitions, etc. None of these coverage concepts is meaningful for a program intending to implement the specification, because its behavior may be unrelated to T, either by design or because of implementation mistakes. There can be distinct specification states not distinguished by the implementation, implementation states never entered by input sequences derived from T, and unreachable implementation states even though all specification states may be reachable. It is simply wishful thinking to treat specifiation states as if they exist in the implementation.

To illustrate this discussion, a state-machine specification for the microwave control program implemented in Section III.C will be given. Refer to Section III.C for a description of the hardware and input-sequence conventions. Although microwave control is a common example, this one is unique in requiring an infinite state set in the specification, and in having an implementation created without reference to the specification.

The behavior of the software controller is specified by a state machine using four actions:

- action $\mathbf{a}$ means add 10 seconds to the timer,
- action $\mathbf{b}$ means start the magnetron and the timer countdown,
- action $\mathbf{e}$ means stop the magnetron and the timer, and
- action $\mathbf{z}$ means set the timer to zero.

There are four state classes: **Init**, **Add**, **Run**, and **Hold**.[30] Each state also includes the door position **OPEN** or **CLOSE**, and the time the magnetron has been on since power up. Including the latter makes the specification state set infinite. However, eight states defined by class name and door position, and parameterized by magnetron time, are useful. Fig. 3 defines the transition mapping. In the figure, circles are states, and arrows

[27]In this subsection the definitions of Section III.A are in abeyance. State machines are not programs, and although their state sets and mappings are similar to S and the semantic mappings on $D \times S$ in Section III.A, they are not the same.

[28]In fact, there is a direct correspondence in semantics. For a state machine M, $[M](x, s)$ is the output from $T(x, s)$, and $\langle M\rangle(x, s)$ is the output state.

[29]The opposite implementation plan, faithfully mimicking T, will be the subject of Section V.C.

[30]The elements describing the state machine are set in boldface to make them easier to distinguish within lines of text.

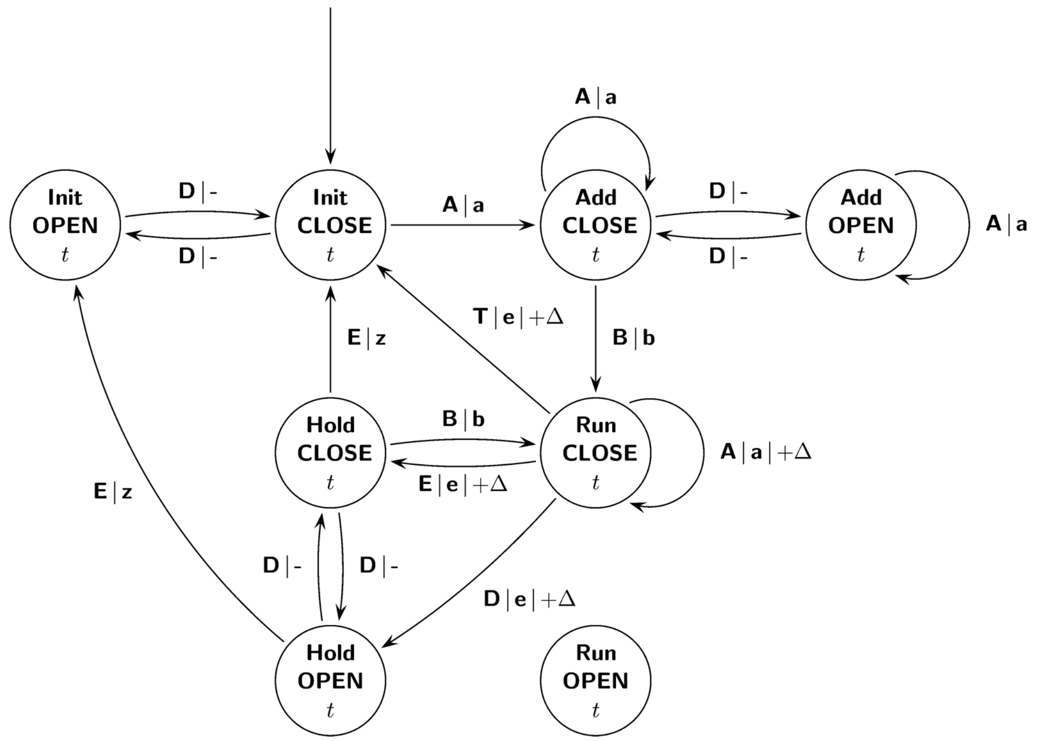

Fig. 3. Transition function of a state-machine specifying a microwave controller.

are transitions between them. A state label $(\mathbf{S}, \mathbf{D}, t)$ has state class name $\mathbf{S}$, door position $\mathbf{D}$, and magnetron time t.

A transition labeled $\mathbf{X}|y$ is specified to take place on signal $\mathbf{X}$, and action $\mathbf{y}$ should be taken. The empty action is $-$, for which the magnetron time of the new state is set to the magnetron time of the source state. Let Δ be the elapsed time since the last transition. A transition labeled $\mathbf{X}|\mathbf{y}| + \Delta$ is specified to take place on signal $\mathbf{X}$ with action $\mathbf{y}$, and a new magnetron time that is Δ greater than the magnetron time of the source state. If no transition is labeled $\mathbf{Z}|\,|$ for some signal $\mathbf{Z}$, on that signal the state machine goes to the same state, does not change the magnetron time, and takes no action.

The specified initial state is $(\mathbf{Init}, \mathbf{CLOSE}, 0)$.

The output specified for a transition is the magnetron time of the destination state. For example, with input $\mathbf{A}\ \mathbf{B}\ \{\mathbf{3}\}\mathbf{E}\ \mathbf{B}\ \{\mathbf{1}\}\mathbf{A}\ \{\mathbf{2}\}\mathbf{D}$, the output[31] should be 6, the magnetron time of the last state. For this input, the specification state-sequence is $(\mathbf{Init}, \mathbf{CLOSE}, 0)$, $(\mathbf{Add}, \mathbf{CLOSE}, 0)$, $(\mathbf{Run}, \mathbf{CLOSE}, 0)$, $(\mathbf{Hold}, \mathbf{CLOSE}, 3)$, $(\mathbf{Run}, \mathbf{CLOSE}, 3)$, $(\mathbf{Run}, \mathbf{CLOSE}, 4)$, $(\mathbf{Hold}, \mathbf{OPEN}, 6)$.

The explanatory power of Fig. 3 lies in grouping the infinite set of states into eight representative states with the same name and door position but different magnetron times.

An input sequence that covers all seven reachable specification states shown in Fig. 3 is $\mathbf{D}\ \mathbf{D}\ \mathbf{A}\ \mathbf{D}\ \mathbf{D}\ \mathbf{B}\ \{\mathbf{7}\}\mathbf{E}\ \mathbf{D}$ (output 7). To cover all transitions shown in the diagram, again ignoring magnetron times, $\mathbf{D}\ \mathbf{D}\ \mathbf{A}\ \mathbf{A}\ \mathbf{D}\ \mathbf{A}\ \mathbf{D}\ \mathbf{B}\ \mathbf{T}\ \mathbf{A}\ \mathbf{B}\ \{\mathbf{2}\}\mathbf{A}\ \mathbf{D}\ \mathbf{E}\ \mathbf{D}\ \mathbf{A}\ \mathbf{B}\ \{\mathbf{3}\}\mathbf{E}\ \mathbf{B}\ \{\mathbf{4}\}\mathbf{E}\ \mathbf{E}$ (output 39).

[31]There is a sequence of outputs, of which the output of the sequence is the final element. In this example, following the 3-second wait, the intermediate output should be 3.

Now consider the code of Section III.C as an implementation of this state-machine specification.[32] It fails to meet this specification on many sequences, including $\mathbf{A}\ \mathbf{B}\ \{\mathbf{1}\}\mathbf{E}\ \mathbf{A}\ \mathbf{A}\ \mathbf{B}\ \mathbf{T}$ (output should be 10 but is 30), and $\mathbf{A}\ \mathbf{D}\ \mathbf{B}\ \{\mathbf{9}\}\mathbf{D}$ (output should be 0, but is 9), where the magnetron has run with the door open!

However, the code gives correct output for the signal sequences above that cover specification states and transitions. Thus there is no guarantee that any sequence selected by the coverage of specification states will detect implementation failures.[33] The intuition that specification-state coverage is significant (as in Mathur's textbook [2]) is valid only for an FSM M implemented by another FSM M', where M and M' are almost the same. That is, M' differs from M by a fault.

B. A State-Aware Version of ART

This subsection describes a plausible testing method called State Adaptive Random Testing Extension (SARTE), which includes the program state. SARTE is a straw man, a simple example of how the concept of state could be brought into a stateless method. Its implementation in testing tools would be straightforward, because it uses only random choices in the input space, and algorithmic modification of those choices. It could be empirically evaluated. However, the difficulty of defining random selection in the wider context of input sequences, and the impracticality of obtaining even an approximate user profile for sequences, as suggested in Section III.D,

[32]The code was written without reference to the specification.

[33]That is, there is no guarantee except of course for coverage that exhausts the input-sequence space. Coverage algorithms that guarantee to detect FSM equivalence, although they do not terminate for infinite state spaces, usually do eventually try any given input sequence.

make theoretical analysis of SARTE or any such method problematic.

The distortions of random selection in adaptive random testing ART are intended to achieve better coverage of the input space. Creating and sampling a collection of reachable states serves the same purpose, but the separation is introduced in the state space. The point of using randomness is to avoid test-selection bias that could hide failures. We want to believe that a random test fairly represents an arbitrary use of some software. To force state spread does not seem to compromise this goal.

The idea behind SARTE is to use random test selection in the input space, but to do so for a series of reachable states, which is itself a random sample. To incorporate the adaptive aspect of ART, the random choices are replaced with ART choices.

A SARTE test sequence is controlled by two parameters: the number of input values N selected with each state, and the number of states M selected. Furthermore, multiple SARTE test sequences may be used, perhaps varying N and M. It is expected that the balance between N, M, and the number and variety of test sequences will be studied by testers using SARTE. Here are the steps of the SARTE procedure a test engineer follows, first without ART.

> Choose N and M. Beginning in the software's reset state s_0, select a random input value (as in Section 3.4), then execute the software, which goes into a state s_1. The tester counts down N.
>
> Repeat several list-adding steps so long as $N > 0$: return to the current state, make a uniform-random input selection, execute the software, count down N, and if the state s' into which the software goes is not already present in the list of next states, then add s' to the list.
>
> (Thus far, the test engineer has randomly executed the software N times starting in state s_0.)
>
> Repeat the restoration-loop steps $M - 1$ times: restore N, and repeat the above loop, but starting in a state s_A that is randomly selected from the list of next states. Make s_A the current state.

To incorporate ART, the two random choices of next input and next state are replaced by choices using ART algorithm(s). For example, when selecting a new state, the random choice could be weighted to favor states in the next-state set that are farther from the current state, as in FSCS-ART.

In one test sequence, SARTE uses $N \times M$ input values, all but the first selected using ART, and between 1 and M state values,[34] each selected using ART.

C. *Extremes in State-Set Size*

This section suggests practical testing ideas for two extreme cases: very small state spaces, and very large state spaces.

In interactive programs, users set persistent-state parameters, called preferences, which serve one important function of state: they store values that customize the software for a user. When preference possibilities are limited to a finite set, it makes sense to test each variant as a separate, stateless program. Preference values classify different users; it may happen that, once these values are given, a rough profile can be found using Musa's methods. Each stateless program in a collection described by preference values is brought into existence by a short execution sequence that sets them. Once this state is established, conventional testing methods apply. For example, random choices for subsequent tests can be made from the input space alone. To maintain the separation into distinct programs for each preference, the state must be checked after each test to see that it has not changed.

At the other extreme from a small set of preferences is software that might be called data driven: its behavior arises almost entirely from values in a large persistent-state space such as a database. The permanent file(s) comprising a database are large, but only a very sparse set of values is valid. Most databases contain discrete, clearly discontinuous values. In a clothing-store inventory, for example, there is no sensible, continuous progression from (say) red wool socks to blue nylon jackets. The input-space possibilities of data-driven programs may be limited to a small set of choices, and these often occur in a few common sequences. For example, in an inventory, items go out to sales, are queried for current supply, and come in through restocking. An order clerk using the database has a stable user profile emphasizing query-sell pairs. It appears that data-driven programs might be easier to test operationally than most complex software, but only by ignoring the database itself. Will an operational test (say a day's work by the order clerk) touch enough of the database to allow a reliability estimate?

In a programmer's naive view, a database is just a persistent file with an over-elaborate scheme for reading and writing it. But it may be that an application using a popular database can be adequately tested in practice, while a direct implementation will prove intractable to test. The essence of this argument is that a database is an encapsulated software component whose integrity can be established apart from the program using it.

A database implementation is a non-trivial program, but its specification is much better than for most programs because it operates on formalized data structures using formalized queries. Testing a database implementation is also non-trivial, because there are two inter-related dimensions: the schema design that declares the relational structure, and the queries that rely on it, both unbounded. However, a database in wide use is the subject of a huge, ongoing operational test. Even across different applications using the database, user profiles that reach the database component are similar, and can be made more similar by dividing testing into three logically independent spheres, each with its own profile: 1) structure declarations, 2) pre-loading values, and 3) production operation. It is sensible to measure reliability for the database itself, and to believe that it will hold approximately for any application.

This view of a database leads us to stipulate that the database engine executes an arbitrary query against arbitrary declared database structures with only a small probability of failure, statistically independent of the application.

The reliability of a program using the database can be statistically independently assessed by taking its output to be the query sent to the database engine. Hence, an application is just

[34]It is reasonable to suppose that the input set is very large. However, the state set might be exhausted. In the limit that there is no state, s_0 is the only choice, and SARTE reduces to stateless ART.

a filter mapping one formal language (describing the input sequences) to another (sequences of queries).[35] Jackson [33] and others have described ways to use grammar for the languages to automatically create correct filters as finite-state transducers. A pre-load application, for example, is particularly simple: an input pair (attribute, value), (a, v), is mapped to query UPDATE a to v.

V. Discussion

This section investigates some wider issues for discussion using the theoretical framework of Section III.

A. Software Has No Science

In a traditional engineering discipline, science and mathematics play crucial roles. Empirical observations of physical reality are the basis for the design rules that engineers use; the engineering is only as good as the science on which it is based. Applied mathematics is the handmaiden of science, providing the intellectual tools to apply scientific knowledge to each design. For example, civil engineering is based on the sciences of mechanics, and the strength of materials. When an engineering artifact fails in the physical world, the cause may be errors in the design rules based on weak science. In an immature branch of engineering, designs fail without apparent cause. As the discipline matures, root-cause analysis is able to pinpoint scientific mistakes. Among others, Harlan Mills noted the youth of software engineering. He believed that in 50 or 100 years[36] the science of software will catch up with (say) civil engineering.

This rosy view uncritically supposes that there can be an underlying science for software engineering, whose laws will be duly discovered and applied to software design and implementation. Everyone looks forward to this science, but in fact there can be none [34]. The laws of programming are not science, but mathematics.[37] The mathematics of software is not the applied analysis essential to engineering calculations and physics, but abstract in the sense of a body of definitions, axioms, and theorems that comprise a mathematical theory [37].

Software is by definition artificial, a human invention from whole cloth. If one chooses to use Hoare's definition, a program means nothing more than the logical predicate obtained from his axioms and proof rules. Because this mathematical definition of a program P is precise, it is possible to establish its properties by mathematical proof, for example that P satisfies a mathematical specification. What then are the root causes of software failure? There can be but one: the person developing the software made a mistake using the defining mathematics. How does that happen? One explanation is that software is called upon to solve difficult, complicated problems, so mistakes are bound to be made. But it may also be that the defining mathematics is hard to understand, inelegant, etc.; that is, bad mathematics.[38] Popper's widely accepted criterion for a scientific theory, that it be *falsifiable* [38], technically rules out mathematical theories as scientific.[39] No one searches for the largest prime number in an attempt to disprove Euclid's theorem. Objective scientific theory can be improved by observation; the subjective quality of a mathematical theory is a matter of expert opinion. Only long, successful experience using a mathematical theory proves its worth. By this standard, the mathematics defining software is certainly lacking.

Mathematical theories can die of neglect if their results are not elegant or not useful. When a theory fails to yield useful results, when its definitions are hard to frame and work with, or do not match intuition, as in Section III.D, it signals that something is wrong. The deficiency may lie in the definition of a program, or with the attempt to apply reliability. Past experience indicates that more abstraction can help. Breakthrough theories like Turing's formalization of computation [39], or Floyd's proof theory [40], succeed because their authors formalized something simple.

B. Program Proofs, Invariants, and Testing

In formally verifying that a stateless program P meets its specification, input-variable values of P are universally quantified. Sometimes local variables of P can be eliminated by symbolic substitution in favor of input variables. But when P has potentially unbounded constructions like loops or recursion, the proof requires a predicate expressed in the variables that constrains them to values that actually arise in P's execution, an *invariant*. The strongest possible invariant captures exactly what P does at the point where it is placed; however, a weak invariant may be easier to discover. In any proof of a program property, there is a balance between the difficulty of finding and establishing a strong invariant and the difficulty of proving the program with a weaker one.

The definition of correctness for programs with state in Section III.A applies to sequences of input values for a program P, with state values carried from one execution to the next in the sequence. There is nothing within P itself to describe run sequences or to restrict its use to sequences. A conventional definition of P's semantics (say using Hoare's axioms) would treat state variables as inputs, because their values are obtained outside P. As demonstrated in Section IV.A, Fig. 2, state values are not inputs, so the Hoare semantics incorrectly describe the actual behavior, instead describing behavior as in Fig. 2(b). Because P's specification may describe the narrower behavior of Fig. 2(a), it may be impossible to prove that P is correct according to the definition in Section III.A.

One way to restore the applicability of Hoare semantics to P is to create a stateless program Q that duplicates the real behavior of P. A conventional Hoare proof of Q will then prove that P is correct according to the definition in Section III.A. It is

[35]Separating a program design into physically independent components that communicate through a little language [32], which is output by one of them to be processed by the other, is a powerful problem-solving technique.

[36]If the discipline dates from the NATO conference that first used the term, 50 years are gone already.

[37]C.A.R. Hoare is one of the founders of this mathematics, yet even he speaks of it as science [35]. Juris Hartmanis is more careful: "...theories do not compete...for which better explains the fundamental nature of information. Nor are new theories developed to reconcile theory with experimental results... there are no experiments...which could resolve...problems [like P = NP?]..." [36, p. 40]. "[Computer science] is the engineering of mathematics" [36, p. 41].

[38]Certainly a case could be made that it is a mistake to define programs to have universal-Turing-machine power, because it makes many important program properties undecidable. Similarly, among many others, pointers and dynamic storage allocation are programming conveniences that we perhaps cannot afford.

[39]A mathematical theorem that is part of a theory can of course be false, not a theorem, if there is a mistake in the proof. But proofs are mechanically checkable, and the accepted definition of a theory is that there are no such mistakes.

not surprising that the construction will introduce the need for an invariant in Q. Q takes as input a list of values, and its output on a list is the same as P's output on the input sequence of those list elements. Q has a local variable S_ℓ, initialized to the reset value of P's state. Then Q enters a loop over its input list, whose body is a slightly modified P, say P'. Where P reads or writes state values, P' accesses or sets the variable S_ℓ. Where P reads input, P' accesses the Q list element that indexes its loop. In P', output values of P are not written, but after the loop, Q writes as its output value the last output from P'. In the execution of Q, each time P' terminates to end one iteration of the loop, S_ℓ will have the state value that P would have had at that point in its input sequence.[40] A conventional proof of Q with respect to P's sequence behavior will require a loop invariant J for the loop in Q, which describes the actual relationship between S_ℓ and the other variables of P', which are just those of P. That is, the strongest J describes the reachable values of S_ℓ, and hence the reachable state values of P.

A kind of invariant also arises in testing a program P, a predicate that holds at a point of P, but not universally for all inputs, as it holds merely over a strict subset of its executions. We call these formulas *test invariants*. The DAIKON system [41] uses execution instrumentation to generate a heuristically defined subset of test invariants. For any collection of test executions, DAIKON can list some predicates that hold across the collection. For example, if it should happen that for all the test executions the values of two program variables `XX` and `YY` at some point are the same, DAIKON will list a test invariant `XX` $=$ `YY` at that point. DAIKON test invariants are not predictions, and do not use a program's semantics; they merely express observations about a finite set of values seen during a test. The complexity of relationships that DAIKON tries as test invariants is severely limited: at most, it will check if two variables are related by a linear equation. For a detailed discussion of test invariants, see Hamlet [42].

Because DAIKON uses runtime instrumentation, it handles state variables like any other. Using considerable manual control, DAIKON could be presented with a collection of input sequences, from which it would produce test invariants for the state behavior of a program P, as defined in Section III.A. The same result would be obtained by giving DAIKON the program Q derived from P as described above. Each test invariant generated in this way does describe a set of reachable states, namely the ones that arose in the test executions.

C. Explicit-State Specification

In the formal-methods software verification community, there is no consensus about explicit state specification. Historically, people who write specifications have viewed state variables as implementation choices, just as are transitory local variables. There is a long-standing prejudice against the over-specification of software: the best specifications should not dictate any decisions that could be left to the software designers. However, for a complex specification, the designers may not have the expertise needed to define and manipulate state values unless the specification is prescriptive. In addition, it is usually easier to specify explicit state values and required state transformations than to describe sequence requirements implicitly. Then a case can be made for requiring a faithful implementation of this abstract state. The detailed proofs of substantial programs usually use explicit-state specifications, which are supported by specification languages like Z notation [43]. In this section, an alternate definition of program correctness is formulated using specified state values, and compared to the original definition in Section III.A. To distinguish these definitions, the new one will be called the *state definition* as opposed to the original *sequence definition*.

Formally, the connection between states of a specification and states of an implementation is described by a mapping similar to the mapping between concrete and abstract worlds in an abstract data type. Abstract (specification) state values are drawn from a set U, while in a concrete implementation a collection of program-variable values $S = \{s_1, s_2, \dots, s_K\}$ encodes a state. (As usual, we take $K = 1$ for simplicity without loss of substance.) The connection is captured by an *abstraction mapping* $A : S \to U$. $s \in S$ is called a *representative* of $A(s) \in U$.[41] It is a fundamental property of A that each element of U must have at least one representative. The definition of A is part of program design, but not part of the implementation code, which can mean that it is not carefully checked [30].

The new state definition of correctness requires a new kind of specification, two relations on $(D \times U) \times D$ (for $[\,]$), and on $(D \times U) \times U$ (for $\langle\,\rangle$). These comprise a *state specification* in contrast to the previously defined sequence specifications that are relations on $D^\infty \times D$. A state specification associates two values with each element of $D \times U$: the specified state, and the specified output. Then (*state definition of correctness*), $\langle P\rangle$ is *(state) correct* iff for all $x \in D$ (*not* sequences, now) and all concrete state values $s \in S$, $A(\langle P\rangle(x,s))$ is a specified output state for input $(x, A(s))$. $[P]$ is *(state) correct* iff for all $x \in D$ and all $s \in S$, $A([P](x,s))$ is a specified output value for input $(x, A(s))$. Program P is *correct* iff both $[P]$ and $\langle P\rangle$ are correct.

It is often stipulated that all states U of a state specification must be reachable, so that the difficulties of Section III.B do not occur at the specification level. Of course, mistakes can occur, but it is perhaps better to treat an unreachable specification state as something to be corrected, than to define correctness in the presence of the mistake. Unfortunately, it remains an unsolvable problem to recognize abstract states that can be reached according to a specification. This puts software developers who want faithful implementation of state specifications in a bad position. Either the exact set of states specified must be worked out, which is not always possible, or it is necessary to specify state results that may never arise, and furthermore to specify outputs and subsequent states that would result from being in these spurious states. Then in turn an implementer will be required to faithfully implement phantom states and outputs, and at the end of the line a tester will be given the impossible task of trying them. It should be

[40]Details of this construction depend on the particular input functions available in the programming language used to write Q, but it is evident that Q can be mechanically derived from any given P.

[41]Technically, the specification may also involve an abstract input space V, and another abstraction mapping $A' : X \to V$. The clarity of this discussion is enhanced by using only an informal notion of specification, and by pretending that A' is an identity: $X = V$.

obvious that the situation provides a wide scope for human mistakes and confusion.

Using a data structure to implement state creates a further reachability problem. The correspondence A between an implementation (concrete) state and an (abstract) specification state is usually many-to-one; that is, more than one value of the implemented structure corresponds to a single specified state. The definition requires that the semantic mappings be correct for all implementation state values, some of which may be unreachable. Implemented values cannot be selected arbitrarily for testing even if it is known that the corresponding specification state is reachable, because it may happen that only another value in the implemented values is the reachable one. Coverage of implemented states may not mean attaining all possible variants of the data structure for one abstract value. The tester has no way of knowing which variant(s) will actually occur.[42]

The result specified for an input sequence $X \in D^\infty$ can be worked out by finding the corresponding sequence of specified states, and then the output specified for (x_ℓ, u'_ℓ), where x_ℓ is the final input value in X, and $u'_\ell \in U$ is the next-to-last specified state in the sequence; the construction is very similar[43] to the definition of program semantics in Section III.A, (1). Given a state specification, it is thus possible to work out the equivalent sequence specification. Nothing similar is possible in the other direction: a sequence specification contains no information about states. Thus the sequence definition cannot imply the state definition. However, the state definition does imply the sequence definition, in the sense that any state-correct implementation returns correct (by the equivalent sequence specification) outputs on all input sequences.

Using the state definition resolves the difficulty in defining so-called state errors, encountered in Section III.D. Because a program P is defined to fail on any pair $(x, s) \in (D \times S)$ where $\langle P \rangle(x, s)$ is not correct (as well as where $[P](x, s)$ is not correct), the failure regions include cases where the output from an input sequence is correct, but along the way the state is incorrect. It may be that a bad state value can lead (in different input sequences) to incorrect output, but strictly speaking P fails by the state-correctness definition even if a state error never leads to a failure.

The practical advantage to explicit-state specification is that an important part of software development is shifted from implementation design to requirements. The implementer does not have to invent state, but merely needs to represent and manipulate it according to specification. Furthermore, the tester need not use sequences for inputs, but can choose points directly in the (input $\times$ state) space.[44] Unfortunately, a tidy specification, and a clear requirement to implement it faithfully, do not technically help to determine if the implementation is correct according to the state definition; all the usual undecidable problems of testing remain. The quality of formal methods used in specification may even be a hindrance, because it is that much easier to commit the testing sin of assuming that an implementation possesses specified properties that have not been verified, as in the microwave-controller example of Section IV.A. On balance, because the intuitive purpose of software is to map input sequences to outputs, perhaps the advantages of the state definition are dubious, which may account for the widespread belief that the sequence definition of correctness is the better.

D. Usage Profiles, and Safety Factors

If there is little agreement on specifying explicit, prescribed state, there is even less support for including usage-profile information in software specifications. Yet all probabilistic testing revolves around a profile, and the huge advantage over other methods that Cobb and Mills [12] describe for operational testing is realized only if the testing profile is similar to actual usage. If constraints on the usage profile were part of specifications, software engineering would be following almost universal engineering practice. No other branch of engineering is expected to produce artifacts that work to specification in *any* environment. Paradoxically, environmental constraints usually act to narrow the available choices and make design easier.

The most important contribution the operating environment makes to traditional engineering designs is in calculating *safety factors*. Whenever a design choice is made, values of its parameters can be adjusted to handle not only the required environment, but a more extreme one. One way to look at the resulting design is that it compensates for unavoidable flaws in construction materials. But the most compelling way to think of safety factors is suggested by Bill Addis [44]: they are the hedge against design mistakes. Engineering can be no better than the science relating environmental forces to the response of real-world objects; safety factors cover the gaps.

As a mathematical object (see Section V.A), the only environment a program faces is the input choices of its users, as reflected in user profiles. However, the analogy between user demands and forces of nature is a pretty good one. A dam must not wash out in (say) a 200-year flood; a program must not crash (say) when a flood of inputs concentrate in a small region of its input space. In building the dam, an environmental extreme dictates a great deal of the design. There is no analogous use of profile extremes in software design. Perhaps the reason is that, in principle, software can be proved correct [35]. A program proof applies to all inputs, so with a proof of correctness in hand, there is no need to consider particular extreme inputs. Unfortunately, the possibility of later proving correctness does not suggest a design, nor does it help in evaluating design choices. It may not be too extreme to say that, because the abstract mathematics of software has been so successful, software engineering is left to flounder, without the constraints that make other kinds of engineering work.

Software has no defective construction materials, and no underlying science to improve by experiment. Nevertheless, its designs can address failure.

Events thought to be impossible. As the final step in development, software testing has the dual role of seeking to exhibit failures for correction, and assessing reliability. In a safety-factor mindset, the designer should also be thinking

[42]In fact, it may be that no variants occur. One must always remember that implementers and specifiers make mistakes, and no specification-based statement, even if true, transfers to the implementation without verification. See Section IV.A.

[43]Indeed, this similarity is probably the reason the state-machine specification is confused with an implementing program, as in Section 4.1.

[44]This condition is subject to the difficulty of selecting one of many representation values as reachable.

of things that *might* go wrong, and providing for them, even though part of the design is devoted to seeing to it that they do *not* go wrong. For example, in a real-time system that dispatches tasks in a cycle by priority, under heavy load the cycle time might be inadequate, so low-priority tasks would go unaddressed.

Of course the designers will have considered the worst case and provided enough time for it. But the design can be made safer by separating out schedule requests that can tolerate an error return, and keeping them out of a busy queue.

Redundant, self-correcting code. The only sure-fire means of improving code is to detect and handle failures during a program's execution. With physical devices, statistically independent replicas are unlikely to have the same flaws, so parallel-connecting identical systems is a standard method of improving reliability. Unfortunately, it is all too easy for redundant computation to produce identical but failing results. Here are two examples. First, the initial implementation of the Airbus 330 flight-control systems claimed to achieve a MTBF of 10^9 hours by duplicating software in three computers, each with a tested 10^3-hour MTBF. Surely, no one except the Federal Aviation Authority believed that the duplicates were independent in any sense. Second, the Ariane 5 launch-vehicle control program failed in its first use. An identical backup system was started automatically, and failed within milliseconds [45]. In *N-version programming*, two or more implementation teams work information-independently (each without knowledge of the other) from the same specification. It is plausible that there will be no correlation between the failure sets of the programs they produce [46], but one study has shown common failures [47].

Sometimes redundant computation need not be instituted until trouble is detected. For example, if a data structure B should not change under operation Q, B can be copied before Q is applied, then checked after Q is complete. If B changed, an alternate version of Q can be used on the copy.

Although it has limited applicability, Manuel Blum [48] has invented a technique that is the epitome of self-correction. If there is an easy way to determine how the output of a program will be distorted when the input is changed, random input variations can serve as statistically independent trials. The same program is used redundantly, but on a range of uniformly pseudorandom inputs. The corresponding outputs are adjusted to see if they agree, and if so, even if the reliability of the program is poor, the whole can be made arbitrarily reliable. Not the least surprising property of Blum's technique is that it works for an arbitrary user profile.

Redundant methods incur extra cost in development- and execution-time; they are not routinely used even in safety-critical applications. Extremes in a user profile could suggest implementation that concentrates on the most frequent situations first. It might be objected that neglected low-frequency cases would then be more likely to fail; still, by definition, they won't arise very often, and reliability should improve.

When a persistent state is involved, it exacerbates the difficulty of identifying important failure situations. It seems unreasonable to expect a specification to assign frequencies to expected input sequences.

E. FSM States, Model-Checking States, and Testing

The most promising new technology for testing programs is *model checking*, a technique that evaluates a logical predicate throughout a program's execution. The predicate expresses a desirable or specified property of the program; it is evaluated on an *MC state*, the vector of all variable values in execution, including the program counter. MC states are not the states defined in Section III, which omit the execution detail. If the predicate fails to hold for some MC state h, an implementation failure[45] has been detected, and h is the counterexample. Using sophisticated algorithms, it has become practical to examine huge MC-state spaces, leading to the hope that something approaching exhaustive testing can be added to the verification toolbox. And indeed there have been some remarkable model-checking verification proofs in communication-protocol design [49], operating-system device-driver implementation [50], and others. However, these successes have not been extended to proofs of imperative programs in general, because of a MC-state-space explosion that overwhelms even the most capable, scalable tools.

The MC states examined by model checkers were originally those of finite-state machines. For a formal FSM M with its FSM state set S and input alphabet Z, a trial (run) from $s \in S$ on input $x \in Z$ is a state transition and output symbol. An M execution sequence results from an input string starting in the initial state of M. Thus, in the terms of Section III.A, $\langle M \rangle$ is the state-transition function of M, and $[M]$ is its alphabet-symbol output mapping.

For the finite state- and input-spaces of an FSM, all interesting problems of testing and verification are solvable in principle by exhaustive testing up to a bound related to the number of states.[46] Model-checking technology expands the size of finite MC-state sets that are tractable in practice. However, for conventional programs, the MC-state set is in general infinite, for two reasons: 1) input-variable values may be drawn from an infinite set, and 2) iteration and recursion on a single input can lead to an unbounded set of MC-state vectors. The latter increase in MC states is called the *MC state explosion*. To illustrate the difference between MC states and the states defined in Section III.A, with a finite state space and a finite input space, there can still be an MC-state explosion.

To control MC-state explosion, an abstraction (or *model*, hence the name) can be substituted for the real execution. Devising a successful model is a highly creative process. The model must have a finite set of MC-states, or its MC-state

[45]The anomaly may not be strictly a failure, if the predicate does not express a correctness property.

[46]These problems are all variations on the program-equivalence problem, which is solvable for FSMs.

set must at least grow slowly, but at the same time the model must be *safe*. If a safe model has no counterexample, then the software cannot fail. The payment for safety is a lack of *completeness*: it may happen that a counterexample in the MC-states corresponds to no real failure. The most successful model-checking is tailored to a particular problem domain, whose properties help in finding a model [51].

When the MC-state set is infinite, there is no necessary virtue in model checking, because selecting a subset of MC states is subject to the same difficulties as choosing test points from a too-large input space.

VI. Summary and Conclusions

A mathematical theory of testing in the presence of persistent state is presented, extending the 1970s and 1980s work of Gerhart, Howden, Duran, and others, based on elementary abstract ideas of software semantics. Program behavior is defined by two partial-function mappings on a (input $\times$ state) space; correctness and reliability are defined for sequences of inputs. The extended theory is used to critique existing testing methods, to propose new methods that take account of state, and to suggest relationships with formal methods and theoretical software engineering.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[2] A. P. Mathur, *Foundations of Software Testing*, 2nd ed. Delhi, India: Pearson, 2013.

[3] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 1402–1411, 1990.

[4] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.*, vol. 1, pp. 156–173, 1975.

[5] W. E. Howden, *Functional Program Testing and Analysis*. New York, NY, USA: McGraw-Hill, 1987.

[6] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, pp. 438–444, 1984.

[7] Z. Manna and J. McCarthy, "Properties of programs and partial function logic," in *Machine Intelligence*, B. Meltzer and D. Michie, Eds. Edinburgh, U.K.: Edinburgh Univ. Press, 1969, vol. 5, pp. 27–39.

[8] H. Petroski, *To Engineer Is Human: The Role of Failure in Successful Design*. New York, NY, USA: St. Martin's, 1985.

[9] H. Hecht, "Rare conditions—An important cause of failures," in *Proc. 8th Nat. Computer Assurance Conf. (COMPASS)*, Jun. 1993, pp. 81–85.

[10] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini, "Evaluating testing methods by delivered reliability," *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 586–601, Aug. 1998, correction, ibid 25, 286.

[11] P. G. Bishop, "The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail)," in *Digest of Papers, 23rd Int. Symp. Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, 1993, pp. 98–107.

[12] R. H. Cobb and H. D. Mills, "Engineering software under statistical quality control," *IEEE Softw.*, pp. 45–54, Nov. 1990.

[13] E. N. Adams, "Optimizing preventive service of software products," *IBM J. Res. Develop.*, vol. 28, pp. 2–14, Jan. 1984.

[14] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE Softw.*, vol. 10, pp. 14–32, 1993.

[15] R. W. Butler and G. B. Finelli, "The infeasibility of experimental quantification of life-critical software reliability," *IEEE Trans. Softw. Eng.*, vol. 19, no. 1, pp. 3–12, Jan. 1993.

[16] P. Boland, H. Singh, and B. Cukik, "Comparing partition and random testing via majorization and Schur functions," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 88–94, 2003.

[17] T. Chen and R. Merkel, "An upper bound on software testing effectiveness," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 1–27, Mar. 2008.

[18] T. Y. Chen, R. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *J. Syst. Softw.*, vol. 83, pp. 60–66, Jan. 2010.

[19] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proc. 9th Asian Computing Science Conf. (ASIAN'04)*, 2004, vol. 3321, pp. 320–329, Lecture Notes in Computer Science.

[20] P. Ducklin, Anatomy of a "goto fail" - Apple's SSL bug explained, plus an unofficial patch for OS X, Naked Security, Feb. 2014 [Online]. Available: http://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch

[21] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. New York, NY, USA: Springer-Verlag, 2003.

[22] D. Hamlet, "Software component composition: Subdomain-based testing-theory foundation," *J. Softw. Test., Verif., Rel.*, vol. 17, pp. 243–269, Dec. 2007.

[23] D. Hamlet, "Tools and experiments supporting a testing-based theory of component composition," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, pp. 1–41, May 2009.

[24] D. Hamlet, *Composing Software Components*. New York, NY, USA: Springer-Verlag, 2010.

[25] S. C. Kleene, *Introduction to Metamathematics*. Amsterdam, The Netherlands: North-Holland, 1952.

[26] H. Mills, V. Basili, J. Gannon, and D. Hamlet, *Principles of Computer Programming: A Mathematical Approach*. Boston, MA, USA: Allyn and Bacon, 1987.

[27] J. Musa, A. Iannino, and K. Okumoto, *Software Reliability*. New York, NY, USA: McGraw-Hill, 1990.

[28] *System and Software Engineering Vocabulary*, Std-24765-2010, 2010, ANSI/IEEE.

[29] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation, PLDI 2011*, San Jose, CA, USA, Jun. 4–8, 2011, pp. 283–294, 2011.

[30] S. Antoy and R. G. Hamlet, "Automatically checking an implementation against its formal specification," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 55–69, 2000.

[31] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, Jun. 1987.

[32] J. Bentley, *Programming Pearls*. Reading, MA, USA: Addison-Wesley, 1986.

[33] G. Dennis, F.-H. Chang, and D. Jackson, "Modular verification of code with SAT," in *Proc. Int. Symp. Testing and Analsis, 2006*, Portland, ME, USA, Jul. 2006, pp. 109–119.

[34] D. Hamlet, "Science, mathematics, computer science, software engineering," *Comput. J.*, vol. 55, pp. 99–110, Jan. 2012.

[35] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin, "Laws of programming," *Commun. ACM* vol. 30, pp. 672–686, Aug. 1987.

[36] J. Hartmanis, "On computational complexity and the nature of computer science," *Comm. ACM*, vol. 37, pp. 37–43, 1994.

[37] G. H. Hardy, *A Mathematician's Apology*. Cambridge, U.K.: Cambridge Univ. Press, 1940, reissued, 2004.

[38] K. R. Popper, *The Logic of Scientific Discovery*. London, U.K.: Hutchinson, 1959.

[39] A. Turing, "On computable numbers, with an application to the entscheidungsproblem," in *London Mathematical Society, Series 2*, 1936, vol. 42, pp. 230–265, correction, ibid 43, 544–546.

[40] R. W. Floyd, "Assigning meanings to programs," in *Proceedings Symposium Applied Mathematics*, J. T. Schwartz, Ed., Providence, RI, USA, 1967, vol. 19, pp. 19–32, Amer. Math. Soc..

[41] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 99–123, Feb. 2001.

[42] D. Hamlet, "Invariants and state in testing and formal methods," in *Proc. Workshop Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon, Portugal, Sep. 2005, pp. 48–51.

[43] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1992.

[44] W. Addis, *Structural Engineering: The Nature of Theory and Design*. Chichester, U.K.: Ellis Horwood, 1991.

[45] J. L. Lions, ARIANE 5—Flight 501 Failure—Report by the Inquiry Board, European Space Agency (ESA). Paris, France, Jul. 1996.

[46] A. Avizienis and J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, pp. 67–80, 1984.

[47] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming," *IEEE Trans. Softw. Eng.*, vol. 12, pp. 96–109, 1986.

[48] M. Blum and S. Kannan, "Designing programs that check their work," *JACM*, pp. 269–291, Jan. 1995.

[49] G. Holtzman, "The spin model checker," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[50] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft," in *Proc. 4th Int. Conf. Integrated Formal Methods*, 2004, pp. 1–20, Springer.

[51] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.* vol. 41, no. 4, pp. 21:1–21:54, Oct. 2009.

Dick Hamlet received the B.S. degree in electrical engineering from the University of Wisconsin in 1959, the M.S. degree in engineering physics from Cornell University in 1964, and the Ph.D. degree in computer science from the University of Washington in 1971.

He is Professor Emeritus in the Department of Computer Science at Portland State University. He was lead systems programmer for the Burroughs B5500 at the University of Washington Computer Center for three years, then Director of Systems Programming for the PDP-10 systems at Computer Center Corporation in Seattle. He spent two years at Shimer College as an Intern in College Teaching. He was a member of the University of Maryland software engineering faculty for 12 years, then Professor at the Oregon Graduate Center, before moving to Portland State in 1988. He retired in 2005. In addition to more than 50 refereed journal publications, he is author or co-author of four books. Beginning in 1972, his research has concentrated on mathematical theories of software testing, and on component-based software engineering.