Software Component Dependability — a Subdomain-based Theory [*]

Dick Hamlet

Draft of September 5, 1996

Abstract

A profile-dependent "dependability" for software can be defined using a functional profile (a set of input-space subdomains breaking down the behavior of a component according to its specification – these are the subdomains used in the usual "functional testing"). It is argued that measurements of dependability using the squeeze play (1) are profile dependent in general; but, (2) when the profile is defined by weighted subdomains, the squeeze-play predictions depend only on the subdomains, not on their weighting, hence this dependability is profile *independent.* A component developer can define functional subdomains for the component and measure the dependability, then market the component with its dependability description. A system developer could select components with attached dependabilities, combine the components in a verified structure, and measure – from the component dependabilities alone – a system dependability. However, the system dependability is *not* profile independent. Measurements for both component- and system developer can be supported by tools.

[*]Work supported by Reliable Software Technologies Corp., Sterling, VA, under NIST/ATP Cooperative Agreement 50-DKNA-4-00119. The author is solely responsible for the contents.

1 Introduction

In most mature engineering disciplines, "quality" is a quantitative notion defined by testing of components (whatever 'components' are in the particular discipline). When large, one-of-a-kind objects are engineered, quality of this object can be calculated from measurements made on the components that make it up. Such calculations then become an intrinsic part of design: the quality needed in the composite object is used to specify that of the components, and they are assembled in such a way that the desired result should be obtained. If, contrary to expectations, the composite object fails, detailed analysis can pinpoint the reason, whether it be a component that did not realize its specified quality, a mistake in the design, or even a basic flaw in the theory on which the calculations rest. This post-mortem analysis is all-important, leading to better components, safer designs, or improved theory, and ultimately, engineering that by and large can be trusted.

A well documented failure of this kind in civil/mechanical engineering is the collapse of the walkway in the Hyatt-Regency Hotel in Kansas City [Pet85]. The apparent cause was component failure – threaded rods suspending the walkway from the ceiling appeared to have given way under load. However, further analysis revealed a design flaw: the rods had been assembled differently than originally specified, and although it was less than obvious (the professional engineer in charge of the project signed off on the design change), the result was to double the stress on the components, which then failed, even though they were adequate in the original design. In this analysis the role of components whose qualities are specified, and which realize specifications, is obvious and crucial.

Software development does not follow the paradigm of engineering. While there are reasonably well-defined software 'components' (the objects of object-oriented design, for example), they are not assessed for quality in any scientific way, and the system designer has no means to analyze the quality of a composite design in terms of its parts.

It is our goal to define a scientific notion of quality (we call it *dependability*), and provide a theory that allows it to be measured for components and composite systems. These measurements can be supported by tools, which we hope will provide a *de facto* standard for the development of high-quality software.

1.1 The Notion of Software 'Dependability'

The intuition behind "dependability" is a confidence probability. In the simplest case, this is what happens:

> Some testing-like experiments or measurements will be performed on a piece of software, it will be observed that the program does not fail, and a confidence can be calculated therefrom that the program will never fail.

The shorthand for this is that the software is "probably correct," but the probability is partly a confidence that the measurements are accurate assessments of that software, in addition to being a property of the software itself.

It is possible for dependability theory to provide for software failures during testing, but in the sequel it is assumed that measurements do not expose any failures. The reason is two-fold: (1) Quality software should be purged of known failures; and (2) In practice, even a single failure seen in testing severely limits the quality that can be predicted.

1.2 Dependability and the User Profile

An ideal measure of dependability would be independent of the usage to which software will be put. In engineering with physical materials, this ideal can never be attained simply because materials are imperfect. However small the imperfection, it is impossible to guarantee that the stress of usage will not exceed what the materials can bear, and the engineered object fail. Thus for example, a dam may be designed to withstand water levels expected within the 100-year flood plain, but it may fail if subjected to a 200-year flood. It could perhaps have been engineered for the 200-year flood plain, but at a prohibitive cost, and with no guarantee should it experience a 300-year flood, etc. Expected usage is an essential part of the design specification.

The promise of software is that as a non-physical medium, it *can* in principle be perfect; in pursuit of this ideal many sins of computing are committed. For an embedded system, perfect software seems irrelevant, since the overall system includes physical components, and may be subjected to usage in which the latter fail. However, if software could be specified, designed, and implemented perfectly, it would be an advantage in that an engineer would have some ability to confine potential failures to where they were least damaging, least expensive to deal with, etc. Thus software perfection is potentially valuable, even in embedded systems. The danger lies in designing as if the software were perfect, only to discover that it is not. This design mode is responsible for many of the horror stories of computer systems gone wrong. And usage profiles are often at fault: the software *was* perfect, except for some unexpected usage, which was either not foreseen in specification, or was foreseen but improperly realized and the imperfection unnoticed because the unusual circumstances were not tried.

It is thus an important question whether or not the notion of dependability should (or can) be profile independent.

The role played by a usage profile for complete software systems, and for their components, is not symmetric. There are certainly cases in which system profiles cannot be eliminated (however difficult it may be to determine them, and however unstable they may be). A system profile induces a profile for each component of which it is composed, namely, the distribution of inputs supplied as parameters to the component, when the system receives inputs from its profile. This induced component profile stands in a very involved relationship to the system profile, since it depends on the system's internal state at the point of invoking the component, a point that is not always reached, and not always reached with the same internal state. A final problem is that component usage depends on the *structure* of the system code, so any change in structure will alter the profile seen by a component. Thus the component developer literally cannot know how the component will be used, and so cannot test it according to that usage. For components, there is therefore strong motivation to seek profile independence.

Furthermore, the necessity for considering profiles is dictated far more strongly for systems than for components. It is system behavior that is the concern of safety analysis, regulatory agencies, etc. But even if it makes no sense to talk of profile-independent "dependability" for systems, it does not follow that such an idea is impossible for components. The very complexity of how components are used may make the induced profile of low importance.

In this report, we present a model of profile-independent component dependability, and show how it can be used for systems that themselves have requirements depending on a profile. Of course, "component" and "system" are relative terms, especially for embedded software. We hope to distinguish a useful case of profile-independence in this continuum.

1.3 Dependability Standards for the Software Industry

It is our goal to define standard procedures – and support them with software tools for measurement and process monitoring – that software developers can use to measure the dependability of components and systems. We wish to alleviate the present unsatisfactory situation in which the presence of any software in a larger system makes it impossible to perform reliability or safety analysis, because the quality properties of the software are unknown. In many situations, particularly involving regulatory agencies and dangers to the public, developers would be pleased to adopt almost any standard, however difficult or expensive to apply, that could provide a plausible estimate of the software's quality.

2 Terminology and Underlying Theories

Although the ideas to be presented here are relatively general, they will be presented in a precise context described in Section 2.1. The reader should view the many restrictions of this context as approximations to reality, needed to obtain initial results, but intended to be relaxed whenever possible.

2.1 Programming Model

We consider only programs with "pure function" semantics. The program is given a single input, it computes a single result and terminates. The result on each input in no way depends on prior calculations. (And hence in particular, if an input is repeated, the result is always the same.) Such programs are not very realistic, but the relevant issues about reliability arise for them just as for more "real" programs, and they considerably simplify the formal description of testing.

This simple programming model abstracts reality, but it is more general than it may appear. Real programs may have complex input tuples, and produce similar outputs. But we can imagine coding each tuple into a single value, so that the simplification to one input value is not a transgression in principle. Interactive programs that accept input a bit at a time and respond to each bit, programs that read and write permanent data, and real-time programs, do not fit the pure-function model. However, it is possible to treat these more complex programs as purely functional, at the cost of some artificiality. For example, an interactive or real-time program can be thought of as having single inputs that are actually *sequences* of the real input elements, starting from some standard "reset" state. Each such sequence is one abstract input in the pure-function model.

Each program has a specification that is an input-output relation. That is, the specification S is a set of ordered input-output pairs describing allowed behavior. A program P *meets* its specification for input x iff: if $x \in dom(S)$, then P produces output y on input x such that $(x, y) \in S$. Where $x \notin dom(S)$, that is, when an input does not occur as any first element in the specification, the program may do anything it likes, including fail to terminate, yet still technically meet the specification. Thus S defines the input domain as well as behavior on that domain. Many real specifications can be recursively extended to be everywhere defined, by adding required 'ERROR' responses; but some, notably involving unbounded searches with uncertain outcome, cannot.

In the sequel it will be sometimes assumed that the programming paradigm is the imperative one, expressed in a language like C or C++. The introduction of a particular programming language is less fundamental than other assumptions, and it is done principally to permit a concrete discussion of tools.

2.2 Tests and Failures

A *test* is a single value of program input, which enables a single execution of the program. A *testset* is a finite collection of tests. Thus in our program model, a testset of size N invokes a sample of a program's behavior, an N-fold Bernoulli trial.

A program P with specification S *fails* on input x iff P does not meet S at x. When a program fails, the event is called a *failure*, and the input responsible is a *failure point*. The program's *failure set* is the collection of all failure points. Hence a program that meets its specification has an empty failure set. The opposite of fails is *succeeds*; the opposite of a failure is a *success*; the complement of the failure set is the *success set*.

2.3 So-called "Faults"

[1] Although there is an IEEE standard term for the idea of "bug" (or "defect," or "error"), this idea is not precise, and is difficult to make precise. The IEEE glossary [IEE83] states that a *fault* is the part of a source program that causes a failure. However appealing and necessary this intuitive idea may be, it has proved difficult to capture formally. The difficulty is that "faults" have no unique characterization. In practice, software fails for some testset, and is changed so that it succeeds on that testset. The assumption is made that the change does not introduce any new failures (an assumption false in general). The "fault" is then defined by the "fix," and is characterized, e.g., "wrong expression in an assignment" by what was changed. But the change is by no means unique. Literally an infinity of other changes would have produced the same effect.

Some fixes do appear to be unique and easily localized (e.g., a wrong operand – perhaps a typo – in an expression). But the most common 'fault' is of omission, and for missing code it is difficult for even reasonable programmers (a rare breed when it comes to assigning blame!) to agree on a fix. In addition, two changes may have quite different overall effects, yet both may fix some intersection collection of failure points. The complications of a "partial fix" that removes fewer failure points than it might have done, and a "least fix" that is in some textual way minimal for the effect it has, are extremely difficult to capture.

So "the fault" is not today a precise idea.

On the other hand, 'failure' is well defined, and so is a change in failure behavior resulting from a program change. Most of what we need to say can be phrased in these terms, as follows:

> A program change may alter the failure set; that is, the changed program's failure set will in general be different from that of the original program. A change is a *fix* for a collection of failure points F (the change *fixes* F) iff: (1) the failure set of the changed program no longer includes any member of F; (2) the failure set of the changed program is a subset of the original failure set.

Thus a fix for a collection of failure points G may eliminate failure points outside G, but it may not introduce new failures.

Maintenance programmers always attempt to find fixes in this technical sense, and good maintainers know that the most important and difficult part of this is condition (2) of the definition: changes must not create failure points that were not there before. (Some argue that a massive change could lead to a completely new failure set, but one that is smaller in size than the original. Those who consider this an improvement usually want to rewrite a piece of software. But they are not maintenance programmers.)

[1] This section, and some of the exposition in Sections 2.1 and 2.2, are adapted from the author's contribution to a paper co-authored with Phyllis Frankl, Bev Littlewood, and Lorenzo Strigini, "Choosing a Testing Method to Deliver Reliability."

In these terms, the closest we can come to speaking of a "fault" is to talk of a *failure region*, a collection of failure inputs that some change fixes exactly. Every change that does not introduce new failure points of course has such a region (if no more than the empty one). It is tempting to begin thinking of such a fix as the basis for defining "fault," but this will not satisfy the intuition behind the IEEE definition. Fixes arise from failure regions, not the other way around, and one can hardly say that an elaborate change tailored to some failure region bears any relation to a mistake made by a programmer.

In this presentation, we will try to avoid using the term "fault." The reader must judge whether the gain in precision is worth the loss of intuition.

2.4 Software Reliability Theory

To define "reliability" testing requires only a few concepts, of which the *operational profile* is the most important. It is assumed that a program has an input probability density U that characterizes its usage. U maps the (assumed to be discrete) input domain (that is, $\mathrm{dom}(S)$ for specification S) into the real interval [0,1]. For input x, $U(x)$ is the probability that x will occur as input to the program. U is called the *operational profile*, (or *user profile*), although sometimes U is called a *density* and "profile" is reserved for a practical approximation to U given as a relatively crude histogram. The term "operational distribution" is also used, although care should be taken to distinguish the density U from the cumulative probability $U_c(x) = \sum_{t \leq x} U(t)$, which is also called the "distribution."

Random testing (or *reliability testing*, or *operational testing*) consists of selecting test input points according to U; for example, by repeatedly choosing a uniform pseudorandom number r in [0,1] and the largest test point t such that $U_c(t) \leq r$.

It is further assumed that there exists a constant θ called the *failure probability*, the probability that the program will fail on a random input drawn from the operational profile. Estimates of the failure probability can be obtained during random testing as the ratio of failed tests to total tests, which estimates are assumed to tend to θ as the number of tests increases. Thus θ depends on the profile U.

Estimating θ is not entirely satisfactory in practice, because estimates of the failure probability might be expected to approach $\frac{F}{|\mathrm{dom}(S)|}$, where F is the number of failure points in the input space $\mathrm{dom}(S)$. But if the operational profile $U(x_f) = 0$ at some failure point x_f, x_f will never be seen; or, if $U(x_f)$ is very small compared to other values of U, there can be an apparent, false convergence in observed values. In a pathological case that M failure points have small U probabilities, while the remaining failure points have substantially larger U values, estimates of θ will appear to converge to $\frac{F-M}{|\mathrm{dom}(S)|}$. The profile U can be such that estimates remain near the false value for an arbitrarily large sample.

In practice, failures are not observed in testing when software is of reasonable quality and close to release status. The reason is to be found in the immense size of the input domain compared to the small size of the testsets there is time to run. Hence direct assessment gives only an estimate 0 for failure probability. An upper confidence bound in a given failure probability can be predicted, however. Suppose a successful reliability test uses N points. Then the chance C that the actual failure probability is greater than θ (technically, C is an upper confidence bound) is:

$$C = (1 - \theta)^N. \tag{1}$$

(The chance that the program will fail one test is θ, or $1-\theta$ that it will not fail one test, $(1-\theta)^N$ that it will not fail any of N independent tests. Thus this is the confidence bound that the experiment is misleading, that is, that the actual failure probability is greater than θ yet the failure(s) that should have been observed were not.)

Even this somewhat dubious theory is not very helpful in practice, because a high-confidence prediction that the failure probability is below roughly $\frac{1}{n}$ requires a test of roughly $2n$ points, so that today's technology is adequate only for predicting in the neighborhood of 10^{-3} or 10^{-4} [BF93].

The *reliability* of the program over N runs is defined to be the probability that it will not fail within N executions drawn from the operational profile, that is, $R(N) = (1 - \theta)^N$.

In other engineering disciplines, there is no question but that physical objects will fail, only a matter of how long it will take them to do so. Hence although θ is the parameter describing the object, engineers insist on knowing the time scale, and expressing quality by R. When they use "reliable" in a shorthand way, they mean that $R(n)$ is near 1 for all n of interest. It may be significant that for software, the shorthand use of "reliable" usually refers to a small value of θ, as if failure were not inevitable and the number of runs of no consequence, even though $\theta > 0$.

2.5 Sensitivity and the 'Squeeze Play'

Jeff Voas has proposed [VM92] that reliability testing be combined with *sensitivity* analysis. Sensitivity is a *lower* bound probability of failure if software can fail, based on a model of the failure process. A sensitivity near 1 indicates a program that "wears its faults on its sleeve": if it can fail, it is very likely to fail under test. High sensitivity captures the intuition that "almost any test" would expose a particular failure, which is involved in the belief that well tested software is probably correct.

To define sensitivity as the conditional probability that a program will fail under test *if it can fail at all*, Voas models the failure process as localized to one program location. For there to be a failure, the location must be executed, must produce an error in the local state, and that error must then persist to affect the result. The sensitivity of a program location can then be estimated by executing the program as if it were being tested, but instead of observing the result, counting the execution (E), state-corruption "infection" (I), and propagation (P) frequencies. Voas's "PIE" model, in its simplest form, takes the sensitivity to be the product of the frequency estimates. Sensitivity analysis thus employs a testset, but not an oracle.

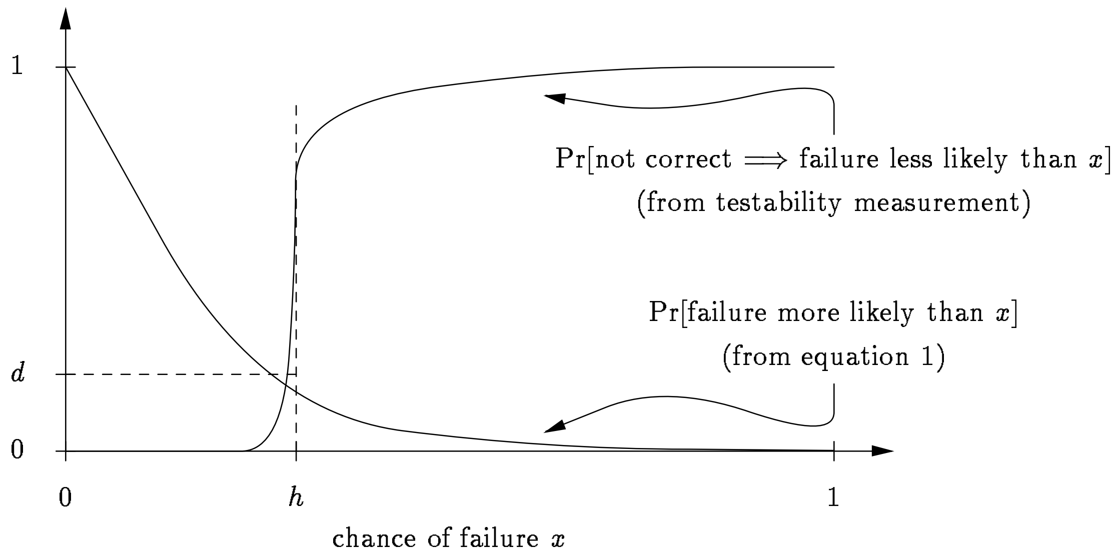

Figure 1: 'Squeeze play' between sensitivity and reliability

For a particular testset, when the sensitivity is high at a location, it means that the testset caused that location to be executed frequently; these executions had a good chance of corrupting the local state; and, an erroneous state was unlikely to be lost or corrected. The high sensitivity observed with a certain testset does not mean that the program will fail because of a particular location; it means that if the assumed failure process could occur at all (but we do not know if it could), then that testset is likely to force the failure.

Suppose that all the locations of a program are observed to have high sensitivity, using a testset that reflects the operational profile. Then suppose that this same testset is used in successful random testing. (That is, the results are observed, and no failures are seen.) The situation is then that (i) no failures *were* observed, but (ii) if failures were possible, they *would* have been observed. The conclusion is that no failures are possible, that is, the program is correct. However, this conclusion is not certain, but only probable, and the confidence to be placed in the program's correctness depends on the size of the operational sample, according to equation (1). If the sample is too small, the lack of observed failures may only reflect insufficient data.

Playing off sensitivity against reliability to gain confidence in correctness is called the *squeeze play* [VM92]. Figure refsqueeze-fig shows the quantitative analysis of the squeeze play between reliability and sensitivity [HV93]. In figure 1, the falling curve is the confidence C from equation (1); the step function comes from observing a sensitivity h. Together the curves make it unlikely that the chance of failure is large (testing), or that it is small (sensitivity). The only other possibility is that the software is correct, for which $1 - d = 1 - (1 - h)^N$ is a confidence bound, where d is the value of the confidence curve at h. Confidence that the software is correct can be made close to 1 by forcing h to the right [HV93]. For example, with a sensitivity of .001, a random testset of 20,000 points predicts the probability that the tested program is not correct to be only about 2×10^{-9}.

We define the software *dependability* as the confidence $1 - d$.

The dependability is a confidence that the software cannot fail in operation (defined by the profile used in the squeeze play). The technical relationship between dependability $1 - d$ and failure probability θ is that $1 - d$ is the confidence that $\theta = 0$. The failure probability expresses the chance of a single operational run failing – it is the parameter in which reliability is expressed, and therefore the parameter of interest to safety experts and regulatory agencies. It is tempting to confuse d and θ, because they have a similar intuitive meaning in extreme cases. When $1 - d$ is near 1, there is high confidence that $\theta = 0$, hence d near 0 makes $\theta = 0$ likely. At the extreme, $d = 0$ logically implies $\theta = 0$. At the other extreme, when θ is near 1, it means there should be little confidence in $\theta = 0$, i.e., $1 - d$ near 0, or d near 1, in agreement with θ, and $\theta = 1 \Rightarrow d = 1$. (Neither argument can be made to go the other way: $1 - d$ near 0 gives no information about θ, and θ near 0 says nothing about $1 - d$.) No quantitative relationship is established by such arguments, except for the unattainable exact zero values. Two intuitive arguments can be given in favor of d being an upper bound on the failure probability:

1. Imagine doing a series of reliability experiments. In each experiment, one operational run of the software is made, and the result observed. The ratio of the number of successful runs to total runs should be the confidence bound in $\theta = 0$, namely $1 - d$. That is, for M total runs, Md runs should fail. Thus $Md/M = d$ is an estimate of the failure probability.

2. d is a confidence bound that $\theta > 0$. Then in particular, for any failure probability b, $\theta < b$ is at least as likely as $1 - d$, since at worst all the probability mass away from 0 lies above b, and the confidence that θ lies at 0 is $1 - d$.

The weakness in these arguments is that they are not careful to distinguish confidence values from probability values. In the sequel we will try to state results in terms of dependability, and leave the connection with failure probability open. For the future, the investigation of bounds on the failure probability in terms of the dependability, is obviously important. Chris Michael has made a start [MV96].

2.6 Profiles and the Squeeze Play

In the discussion of Section 2.5 the operational profile was assumed, but an arbitrary profile can be used for both sensitivity and reliability measurements (but of course the same profile must be used for both). A cautious statement of the significance of the squeeze play would explicitly mention the profile used:

> When N successful test points are drawn from profile V, and the sensitivity is estimated to be h using V, the confidence that the software cannot fail (the dependability) *when its usage profile is* V is $1 - (1 - h)^N$.

The important question to be discussed now is whether the italicized phrase can be omitted from this cautious statement. That is, can one profile be used for the squeeze play measurements, and the dependability prediction hold good for a different profile when the software is used?

Intuitively, the answer is 'no' – the operational profile is essential. The argument in support of this negative position is the one that requires all testing activity to be "representative." If tests neglect part of the input space, how can their predictions be trusted should that part become important?

However, there is another intuitive argument that gives the opposite answer – that the squeeze play *does* estimate a correctness probability for all profiles, no matter which of them is used in its measurements. This argument examines the way in which sensitivity reacts to profile. Imagine one particular program location that might be a source of operational failure. If that location is neglected during testing, then its sensitivity will be low, and hence the prediction of probable correctness will be modest. Thus the sensitivity part of the squeeze play compensates for profile.

These two intuitive arguments do not conflict on a somewhat different question: Does the squeeze play make different predictions for different test profiles? Both agree that it can do so. But that different question is not of interest for software dependability. We wish to conduct test-like measurements, and be able to trust predictions based on those measurements. There might be an ideal profile that would give the best measurements, and it might be worth expending effort to find it. But the fundamental question is: if the test profile, be it ideal for the squeeze play or no, is not the user profile, then can the squeeze play predictions be believed?

Quantitative calculations shed some light on the validity of squeeze-play predictions under different profiles. We consider the sensitivity of a single program location, and construct examples to show how the squeeze play behaves.

First, imagine that a certain faulty location is never executed under the test profile, yet is important in the user profile. The program would be expected to fail often in use, that is, to have a failure probability near 1. The sensitivity is 0 (because the execution probability estimate is 0), hence the dependability prediction is that the software is correct with confidence at least $1-(1-0)^N = 0$, which is correct, if not very useful. At the other extreme, suppose the sensitivity at the faulty location is close to 1 for the test profile, because this location is executed by most of the tests, and changes in the state occur, and these persist. Then the program is very likely to fail under test, and no failure-free runs can be obtained, so the prediction is $1-(1-h)^0 = 0$, also correct.

Between these extremes, it is possible to falsify the prediction of the squeeze play, but not easy to make it wildly wrong. Here we must assume that the dependability can serve as a bound for the failure probability, as discussed at the end of Section 2.5. For example, suppose the software has a failure probability caused by a certain program location of .015, under the user profile. Let the user profile execute this location with frequency E near 1. The frequencies for infection I and propagation P are not measured for the user profile. The squeeze play measurements use a different test profile. In the test profile, let $E = .01, P = 1, I = .5$, so that the sensitivity $h = .005$. Then with 200 tests in the squeeze play, the dependability prediction is $(1 - .005)^{200} = .37$. Taken as an upper bound on the failure probability, the result does not contradict the assumed value of .015. For 1000 tests in the squeeze play, the dependability predicted is .0067, too low by a factor of about 2 to bound the failure probability. Evidently, by increasing the number of squeeze-play tests, the prediction can be made as small as we like, less and less accurate for the user profile. But there is an additional constraint: it must be plausible that the squeeze play could be conducted at all. The reliability part of the squeeze play requires a failure-free test – is that likely? About 10 of 1000 tests will execute the location ($E = .01$), and assuming the usage failure rate of .015, the chance that there will be a failure is 15%. Thus there is an 85% chance that the squeeze play could have been conducted as assumed. However, increasing the number of tests to make the prediction wildly inaccurate is not possible – for example, it is virtually impossible that 5000 tests could be executed without failure in the squeeze play.

The example of the previous paragraph assumed that the test points executing the faulty location in the squeeze play are in some sense the same as inputs that execute it in actual use. This assumption is the source of the constraint that the reliability part of the squeeze play cannot be conducted if the operational failure probability is too high. But the assumption is in general unjustified. Indeed, many examples of testing gone wrong (in that tests did not expose problems that later appeared in use), are examples of different inputs executing the same locations, but in such a different way that all is altered. If we relax the assumption about this "sameness" between the two profiles, then we can produce examples where the predictions of the squeeze play are arbitrarily wrong when the wrong profile is used. For example, in the last case of the previous paragraph, taking 5000 tests gives a predicted bound on the failure probability of .000000000013, off by a factor of about a billion. What makes these examples work is that the test profile and the user profile are assumed to have entirely different properties: in the test profile the P, I, and E frequencies may all be small enough so that a large number of tests can be run without failure, predicting dependability near 1; while under the user profile P, I, and E are larger, so failures *are* observed, inconsistent with high confidence that the failure probability is 0. That is, the wrong profile leads to a wrong prediction.

In the sequel we adopt the conservative view corresponding to the negative intuition about profile-independence of the squeeze play. We assume that one can only trust the squeeze-play prediction if the profile is unchanged.

3 Dependability for Components and Systems

The background work of Section 2 was developed with complete programs in mind, without regard for their possible components. Indeed, except for the squeeze play, programs play hardly any part in the theory, which is only concerned with the function that a program computes – its input-output behavior – as sampled through the input domain. However, this theory can be applied to both components and complete systems.

3.1 Subdomain-defined Profile

The probability density U over the input domain that defines a user profile cannot be realized in practice. Real users simply do not know how likely it is that each input point will be invoked. The approximation to U used in practice [Mus93] is based on functional subdomains of the input domain. The user identifies a number of "functions" (usually mutually exclusive) that the software should perform, which defines a division of the input domain into subdomains, one for each function. The inputs in the subdomain for function g are all those for which the software should compute g. Testing using these subdomains is usually called "functional testing" (or "specification-based testing," since the subdomains are defined by functions that come from the specification; or, "blackbox testing," since no use is made of the program structure). Functional testing becomes an approximation to random testing when a profile histogram is created by assigning relative weights to the subdomains. These weights take two forms: (1) if a subdomain occupies only a small part of the input domain, a fixed number of test points chosen in each subdomain give a small subdomain more weight than given to a larger subdomain; (2) the number of test points selected from each subdomain can be adjusted to apply an arbitrary weighting. Pascale Thévenod has looked at uniform sampling within functional subdomains experimentally [TF93]. She calls this "statistical testing," and finds that it is a better fault-finding technique than subdomain coverage without the additional samples.

The assumption of uniform sampling within subdomains is crucial – without it the problem of profile arises again in each subdomain. Uniform sampling can be justified as the only practical possibility. Users are unable to distinguish practically between different points in the functional subdomains, and so in terms of the profile histogram, the subdomains are "homogeneous." Should some part of some subdomain be considered different by the user, then that part should be broken off as a distinct subdomain, with its own weighting, to be uniformly sampled in turn. Such a process of subdomain refinement can in fact be used to obtain the profile histogram.

It is common practice in practical testing also to use subdomains defined by the program structure (for example, subdomains corresponding to a particular program path, or other coverage criterion). A valuable compromise is "broken-box testing," in which functional subdomains are refined by including gross program characteristics such as the size of internal data structures [Mar95]. All such program-based ideas must be excluded from the subdomains that define a profile histogram, because however valuable program structural coverage may be in uncovering problems, it has nothing to do with the user profile. The ideas to be presented here can be used for any subdomains, however defined, but the connection to operational failure probability requires that subdomains define the profile histogram.

The idea to be pursued below (Section 3.2) is to base software quality measurements on a decomposition of the input space into functional subdomains, the subdomains that are weighted to define a practical user profile. However, no weightings are used; instead, the subdomains are utilized in the squeeze play with random tests in each subdomain drawn from a uniform profile there. Thus the measurements and predictions are profile independent. However, they do depend on the choice of subdomains. If the subdomains are poorly defined – the test of this would be that they cannot be weighted to form an accurate user profile – then the quality measurement will also be inaccurate.

Subdomains could be in error in two ways: first, a subdomain may not be homogeneous, so that uniform sampling within it is not appropriate; second, a subdomain may be misdefined (the functional criterion is after all a subjective one). Both possibilities produce the same kind of problem: the measurement for a subdomain may be misleading, because the measurement process misses part of the subdomain where things are different (a local inhomogeneity, or an embedded piece that should have been a different subdomain). To quantify this misleading effect requires only that we bound the size of the hypothetical missed part. As the sample size increases in a subdomain, it is less and less likely that a part of fixed size has been missed. The situation is exactly analogous to sampling for program failures, with the hidden subdomain playing the role of the unknown failure points. That is, if N uniform samples are taken, then the confidence C that a region has not been missed is $C = 1 - (1 - f)^N$, where f is an upper bound on the fractional size of the missed region. For example, to have 90% confidence that a region of size 50 in a subdomain of size 100,000 has not been missed entirely, requires $\frac{\log(1-.9)}{\log(1-.0005)} \approx 4600$ samples.

In the theory to be presented below, it is of no consequence if the subdomains overlap; however, it is crucial that they exhaust the input space. If some part of the space is omitted, this is an ultimate subdomain error: the results of test measurements can have meaning only for user profiles in which the missing subdomain is never used. It is one of the cardinal properties of a good requirements specification that any categorization of inputs exhaust the input domain, and good techniques exist for analyzing this property, beginning with the work of Goodenough and Gerhart [GG75].

3.2 Profile-independent Dependability

Suppose then that a developer has defined functional subdomains for a software component (as described in Section 3.1), but does not know the weightings of those subdomains that would form a profile for the component. As indicated in Section 1.2, it is impossible to trust conventional reliability testing without the weightings, for it could happen that low (test) emphasis was given to an important (for the user) subdomain. Even if the developer were able to obtain a profile for the user's system, that does not practically lead to a profile for the component. The squeeze play provides a way out of this difficulty.

Let the developer proceed as follows to attain a profile-independent confidence goal $1 - d$ that the component is correct:

> Choose K uniform random tests within each of the subdomains, such that K is large enough to ensure that no part of the subdomain is badly neglected. Combine the tests for all subdomains into a composite test, and measure a confidence of correctness $1 - d_K$ with the squeeze play. If $1 - d_K > 1 - d$, stop. Otherwise, choose one of the subdomains, say S1, and choose additional uniform random tests there for a total of K_1 points in S1. Repeat the composite test, measuring $1 - d_{K_1}$. If substantial improvement over $1 - d_K$ occurs, continue to increase the testing within S1, until the goal $1 - d$ is reached, or improvement diminishes. In the latter case, add points to another subdomain S2, and continue with one subdomain after another and more uniformly chosen points in each, for a total of m steps, until the the goal is reached, or until the measured confidence $1 - d_{K_m}$ is stable without reaching the goal. In the latter case, $1 - d$ cannot be attained.

The procedure can also be used without setting a dependability goal, but rather by fixing the resources available for testing, and then attaining the best possible dependability within the resource constraints. If no goal is set and there are no constraints, then a dependability is obtained that is a property only of the software, as shown below.

This procedure is not quite mechanical, since the developer must decide when further additions to a subdomain are to be abandoned in favor of another subdomain. However, in practice it should be relatively straightforward to carry out. The procedure may also help to refine or correct the subdomain definitions, because the tester may see that the reason the goal is not being reached is that some inputs are neglected – these should then become an additional subdomain. However, it is always possible that the dependability which can be attained is limited by intrinsic qualities of the component under test.

We claim that (given that the L subdomains used are correct in the sense that the actual user profile could be obtained by weighting them) the dependability $1 - d$ obtained by the procedure above in the absence of goal or constraints, is profile independent. That is, no matter what weightings might be given to the subdomains, the dependability remains above $1 - d$, or accepting the interpretation at the end of Section 2.5, failure probability would never exceed d for any profile. Consider an arbitrary profile, defined by a normalized weighting $\{w_i\}_{i=1}^{L}$ of the subdomains. The difference between the new profile and the one used to obtain $1 - d$ is that for the new profile, in some subdomains more points will be used, while in others, fewer points will be used. Only the former need concern us, since with fewer points failure is less likely. But using more points in some subdomain cannot improve $1 - d$, since the procedure has itself tried additional tests until there is no further improvement.

3.3 Dependability for Systems and Components

Designing a system using components that come with dependability values and a collection of subdomains used to measure those values, is quite different than current practice in design.

First, the system designer must explicitly deal with the possibility of component failure. In some cases this is easy – in a non-critical batch system, for example, processing can be aborted with a call for human attention. The system and component developers can then analyze the problem, and correct the software so that the run can continue. In real-time applications things are obviously more difficult. However, fault-tolerant techniques might be used to handle failures, particularly ones resulting from concurrent non-determinism. Continually providing for failure is not easy, but the payoff is a system that is conservative in the sense that it either works properly (with high probability), or reports its own failure in such a way that correction is possible. In contrast, systems using components of unknown quality are easier to design, but can fail silently with unknown consequences.

Second, components that come with dependability values dictate a system design in which critical components are of higher quality (and correspondingly more difficult or expensive to obtain), while less important components can be of lower quality. The designer must analyze the pattern of component use to make decisions about the quality of components needed. The design process thus acquires an added dimension that requires the software engineer to determine if each component is good enough to meet the overall requirements of the system. Usage profiles for the overall system are important in this analysis, since the profile determines which components will be used.

Testing of a system with components of measured quality can substitute for system analysis. Instead of calculating the conditions under which a component will be invoked and the necessary quality, the running system can be instrumented to measure worst-case confidence in the overall result, by monitoring the values returned by the components. For example, for one system input, the system dependability might be as low as that of any component that is invoked. If multiple components are invoked, or a component invoked repeatedly on one system input, then system dependability might be a low as the sum of their dependabilities.

3.4 Example: A Communication Protocol

To illustrate the idea of subdomain-based dependability, we take a particularly simple example. Suppose a component is specified to implement a protocol defined as follows:

> Input is a string of 8-bit characters, ending with a NUL (00000000_2). The output is to be a string of fixed length 255, consisting of exactly the input characters in the input order, except that (1) the terminating NUL is not included, (2) any non-ASCII characters (that is, those with the high-order bit set) are deleted, and (3) the output string has an additional 8-bit (unsigned) first character C that is a binary count one

more than the number of characters reproduced following it, from 1 to 255; characters in the output after the first C positions are arbitrary. If the input string has more than 1023 characters before a NUL occurs, or if more than 254 ASCII characters occur before a NUL, then in the output C is 0, and the remaining characters of the output string are arbitrary.

This specification is deficient in that it fails to describe the input string *beyond* the presumed NUL terminator. Should an implementation fail to recognize the NUL, this omission might have serious consequences, because tests may not have examined many variations on this situation. In testing, these additional characters must be supplied as input.

There are many ways of defining "functional" input subdomains for this specification relation; one is given below. In the descriptions of input subdomains, the input "string" does not include the terminating NUL, if any, and the choice of "ASCII" characters does not include NUL.

S0 The string of zero length. (That is, the first input character is NUL.)

S1 All-ASCII strings of length 1-254. (This is the presumed "normal" case.)

S2 Strings of length 1-255 containing one non-ASCII character.

S3 Strings of length $N < 1024$ containing $(N - 254) > 0$ non-ASCII characters.

S4 Strings of length $N < 1024$ containing $(N - 255) > 0$ non-ASCII characters.

S5 Strings of length $N < 1024$ containing more than 255 ASCII characters.

S6 Strings of length $N > 1023$ containing fewer than 255 ASCII characters.

Any operational profile can be given by weighting these subdomains. The procedure of Section 3.2 can then be used to measure the component's dependability. Rather than setting a goal, the developer might allocate resources for the testing, and obtain the best possible dependability using those resources. During measurement of the dependability, failures may be encountered. If so, the component should be repaired, and the measurement starts over from scratch. In what follows, it is assumed that the squeeze play uncovers no (further) failures, as will eventually be the case.

S0 appears to be a special case of a finite subdomain, which any single test exhausts. However, as noted above, there are actually an infinity of inputs in which the string of zero length is followed by other characters, which may matter if the program being tested misses the NUL.

S6 is in principle an infinite subdomain, containing strings of arbitrary length. To sample it will require setting a bound on string length. If we are confident that should the program fail to stop at 1024 for a string much longer than 1024, then it will also fail to stop for a string of length 1024, then it makes more sense to approximate the subdomain with a collection of 1024-long strings.

However, we choose instead to sample from all strings of length 1024 - 2000. (Thus S6 is the best candidate for a subdomain improperly defined, in which other subdomains are hidden.)

For S1-S5, exhaustive testing is possible (with the same qualification as for S0), but impractical. (S2, for example, contains $\sum_{k=1}^{254} k(128)^k$ strings).

Suppose then that the procedure of Section 3.2 is carried out using no less than 1000 points in each subdomain, and a sensitivity of 0.01 is obtained, so that the dependability is at least $1 - (1 - .01)^{1000} = 4.3 \times 10^{-5}$.

It is instructive to consider how two different components might fare in a dependability comparison. First, the developers might choose different subdomains. For example, another choice would be to combine S3 and S4 into a single subdomain; yet another would be to split off (and exhaustively test) particular conditions such as that the first 254 characters are ASCII 'X' and character 255 is 11110000_2; and so on. The more elaborate the subdomains, and the more they intuitively correspond to possible malfunctions of the component, the more intuitive significance the component's dependability will carry. However, adding subdomains increases the testing effort, and certain subdomains may contribute little to improving the squeeze-play measurements.

Second, even if their developers take common subdomains, two components can differ in the dependability value measured. Since the sensitivity h is partly a property of the code, one component may have a larger h. The number of tests N reflects the effort that goes into the measurement. Since part of the squeeze play is a reliability test requiring an oracle, one developer may be able to employ a much larger value of N because (say) an automated oracle based on a formal specification [PP94] was employed, or an automated testing harness was utilized. Or, one developer may simply have allotted more time for testing, and been able to use it because the development process was well organized and the schedule did not slip. Of these factors, the result is most sensitive (pun intended) to the value of h.

An honest developer of the program whose hypothetical dependability is given above should believe that the protocol implementation is correct. Other correct implementations might yield different dependabilities, for the same or different subdomains, making a comparison between them possible. The developers may all believe that their software is correct; only the dependability values are available to others to judge component quality. The developer who wants to improve the dependability can always refine the subdomains to be more intuitively appealing, and can redesign for better sensitivity h (for example using assertions [VM94]), and finally can run more tests N to improve the dependability.

However, the developer of a component that is *not* correct faces real limitations. Such a component, where the sensitivity is high, cannot be tested extensively without failing. The developer must then correct the code, and try again. If the quality is really poor, the process may exhaust the resources set aside for testing, and the final resolution may have to be that N is kept small (and hence the dependability poor) so that residual failures do not show up. Or, if the sensitivity is low, the component's errors may be hidden from test, but the dependability will remain low. In either case, the developer may be able to do nothing except attempt to conceal the poor quality.

3.5 Tools for Dependability Measurement

Dependability assessment as defined in Section 3.2 requires an investment that software developers do not make today, partly because more testing effort is required, but more importantly because an operational profile must be defined and used. John Musa has done a great deal of practical work with profiles [MIO87], and has argued that they should be used in any responsible testing effort, but the substantial investment remains. Here the problem of profile definition is somewhat less severe, because our subdomains for functional testing of a software component need not be weighted – all that is required are the subdomain definitions. For the testing and sensitivity analysis that the squeeze play requires, good tool support can be provided.

3.5.1 Component Development

Imagine then that a certain component has a given list of subdomains, and that a minimum sample size is fixed in each domain by giving the fraction of the domain that the developer believes can safely remain unsampled, as discussed at the end of Section 3.1. The procedure of Section 3.2 can then be used to measure the dependability. Testsets are chosen by selecting points from a subdomain at random, using a uniform distribution. The component is executed on these testsets without failure, and the sensitivity estimated, so that the squeeze play can be used.

The existing tool PiSCES performs all the necessary calculations for C programs, except that PiSCES analyzes a stand-alone "main" program, and has no support for confining tests to a subdomain. To modify PiSCES so that it supports the calculation of dependability defined in Section 3.2 is straightforward. A driver program must be included to convert a component to a stand-alone program. This is a well understood process supported by several commercial testing systems. Provision must be made for subdomains, and for uniform random test generation confined to a subdomain. Finally, it will be useful to provide a harness for an oracle to judge test results. These features might be supported outside of the existing PiSCES, because the tool can use arbitrary files for input and output; in an experimental implementation, these files can be manipulated by other programs that provide subdomain bookkeeping, the random test generation, and the oracle.

Figure 2 gives a schematic diagram for measuring and calculating the dependability for a component, showing which parts of the process may be automated.

3.5.2 System Development

Now suppose that a component C is used in constructing some larger software system S, and suppose that C comes with a dependability measure (and associated subdomains) as in Section

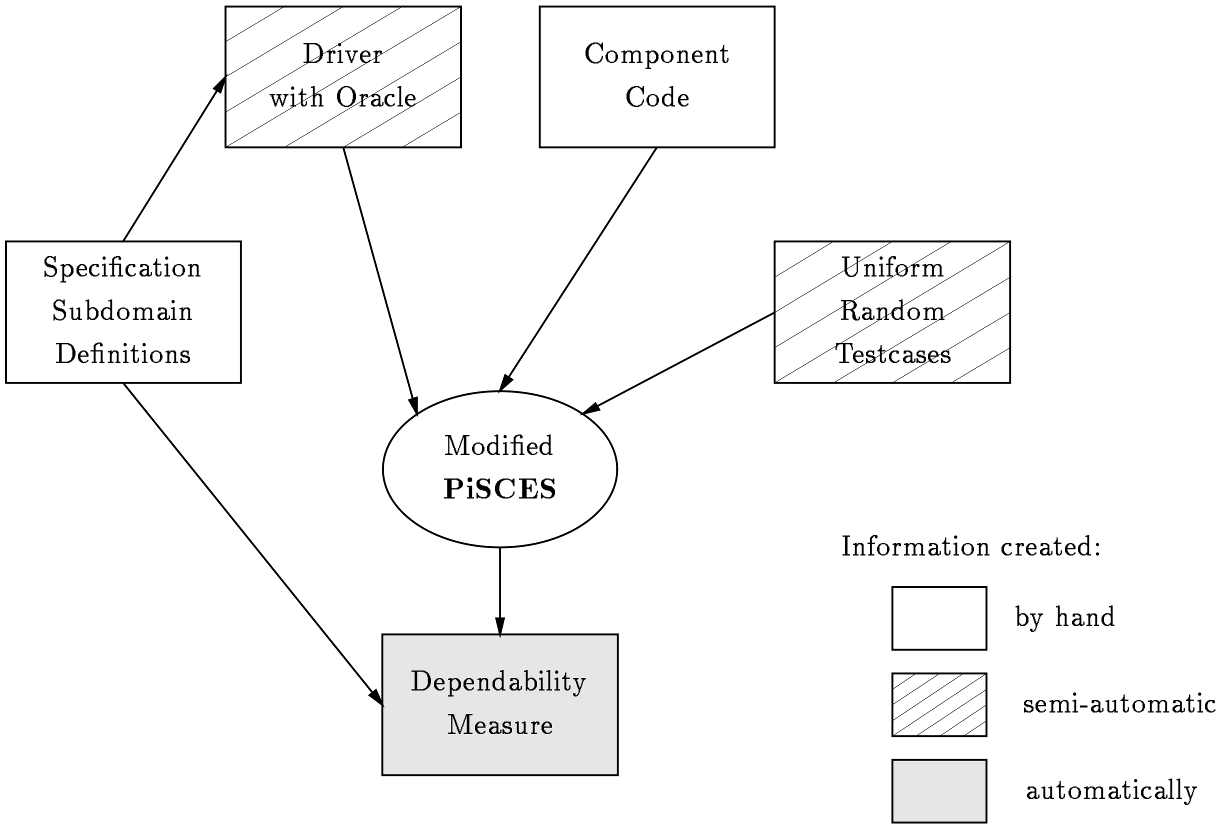

Figure 2: Assessing dependability of a software component

3.5.1, provided by C's developer. When the composite system S is executed with input Y, runtime instrumentation can determine a bound on the probability that S might fail. Each time C is used for input Y (and it might be used many times if called in a loop, or not used at all), then its known dependability comes into play. The dependability of S for input Y could be as bad as the sum of the subdomain failure probabilities encountered in C and other components. Each component might not be perfect, and the chance that the system is not perfect might be as large as that its components are not, in the way they were actually used by S on Y.

Unfortunately, a system S can fail even though its components do not fail, because S may not employ their results properly. For example, S may invoke a sorting component on a list, which correctly returns a sorted list; but, S may then mistakenly use the original list rather than the returned one. In constructing systems from components, there is a continuum of possibilities for the system structure, from implementing almost everything at the system level, to a trivial system structure in which components do all the work. It is evident that the latter is the best choice for building systems of high quality using high quality components. In the sequel, we ignore the possibility of failure outside of component failure. One way to validate this assumption is to prove that the system is correct, contingent on the correctness of its components. If the system structure is extremely simple, program proofs of this kind need not be difficult, and we recommend exactly those system structures.

If S were released to its customers with instrumentation in place to monitor component usage and predict runtime confidence in each result, it would realize what some customers want from software: with each result R, the system S would provide a worst-case probability that R could be trusted. The instrumentation would also allow S to be tested without an oracle, using instead some cutoff value of the runtime confidence to decide if each result is correct.

Given a user profile for the system S as described in Section 3.1, the system developer can also carry out further testing to provide a dependability bound for S, as follows:

- Form a testset by uniform random selection from the profile.
- Run this testset without failure, obtaining worst-case values for the confidence (as described above) for each point Y.
- Take the minimum confidence over the testset as the dependability bound for S.

This procedure produces a profile-dependent dependability bound for any composite S, be it a complete system or a component built from other components. (But recall the crucial assumption that the structure of S is correct, so S cannot fail except through component failure.) The dependability bound obtained is less accurate than if it were measured directly, because of the worst-case assumptions involved. However, as noted below, there are a number of advantages to the indirect measurement, that may make it desirable despite the loss in accuracy. One use is as a quick indicator that one component of a system is not of adequate quality. If system dependability is unsatisfactory, and this can be traced to the contribution of a particular component, then that component can be replaced, or the system design changed to use that component less frequently.

Tool support for the systems developer is also easily provided. Standard self-instrumenting techniques can be used to trap component usage, and the bookkeeping needed to calculate the composite dependability is straightforward. Figure 3 indicates the elements that go into a system measurement.

The loss of accuracy in measuring a composite dependability as above is substantial. However, there are three important advantages over direct measurement:

1. The dependability measurement can be done by the system developer or by a third party, because it does not require access to the code of the components, for which their dependabilities stand in. If the component sources are not available, there is nothing else to be done. A corollary can also be important in practice: the use of component dependabilities in place of direct component measurement saves system test time, since the component developer has done the work in advance.

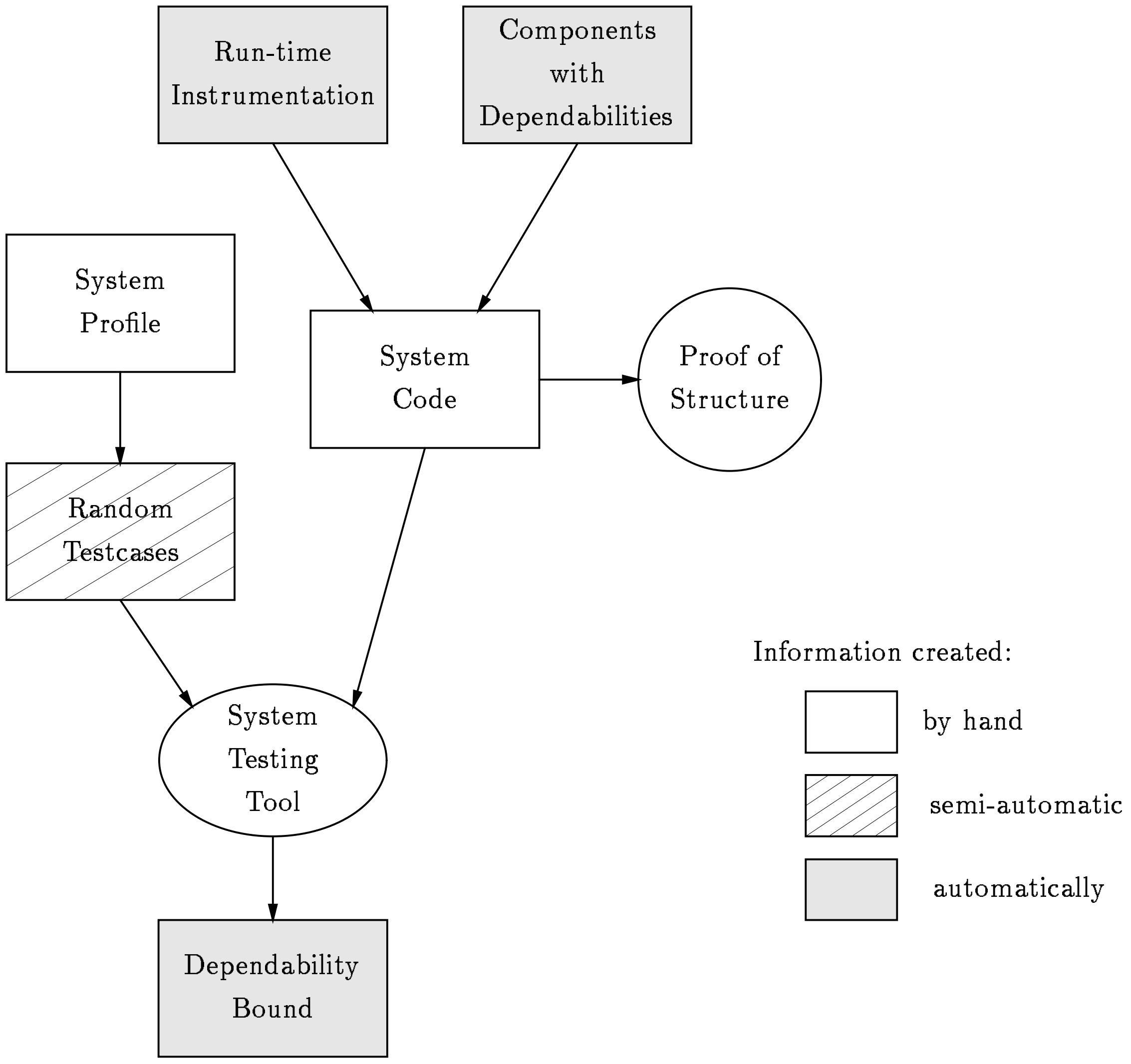

Figure 3: Assessing dependability of a software system

2. The dependability measurement requires no oracle of system behavior, since it monitors not whether the composite system meets its specification on an input, but rather correctness probabilities for that input based on component usage and quality.

3. The developer does not have to deal with any theory of the composition of components; in fact, there are no assumptions about composition. Point-by-point measurements are taken on the composite system, looking at component usage as it occurs. (It is the proof of correct system structure that underlies this advantage, however.)

A compromise is possible between measuring a system's dependability with its user profile as described in this section, and a profile-independent measurement as if the system were a component, as in Section 3.5.1. The procedure of this section can be carried out, not for tests drawn from the profile, but for uniform random tests drawn from each subdomain as in Section 3.2. The resulting bound can be shown to apply to all profiles. However, the drawback is that the developer no longer can afford to use lower-quality components in low-usage positions. Every component is likely to make a substantial contribution to the worst-case failure probability. In fact, these compromise measurements are most likely to lead to bounds that are of no practical use, since the worst case will be that the system is sure to fail.

3.6 A Standard for Developing Quality Software

The purpose of a dependability theory is to provide the theoretical foundations for software development practices to bring software quality into line with other artifacts of human construction. The theory presented in this paper is not ideal, first because it relies on a subjective definition (by the developer, who may use this subjective element to fudge results!) of an input-space decomposition into subdomains. Second, it uses the squeeze play, which is in turn based on a very simple model of software failure, to define its basic measurements. These two deficiencies aside, the theory provides component developers with a way to expend effort in quantifying the quality of what they produce. It is not certain that a high-quality component will yield good dependability under the theory, nor be distinguished from a poor-quality component, because if the sensitivity h is low, there may not be adequate testing resources to distinguish the two. However, the developer can also expend effort to avoid this situation, by increasing h.

Acknowledgments

Chris Michael read a draft of this paper, and provided many helpful comments. He is also responsible for an insightful discussion of the squeeze play and its dependence on profile. We continue to disagree on some aspects of the squeeze play, but my ideas are much clearer because of his comments. Boris Beizer reminded me that system developers are not able to perform measurements on proprietary components.

References

[BF93] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. on Soft. Eng.*, pages 3–12, 1993.

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

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

[IEE83] IEEE. *IEEE Standard Glossary of Software Engineering Terminology,*. IEEE/ANSI Std 729-1983, 1983.

[Mar95] B. Marick. *The Craft of Software Testing.* Prentice-Hall, 1995.

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

[Mus93] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, pages 14–32, 1993.

[MV96] Christoph Michael and Jeffrey Voas. Test adequacy criteria and probable correctness testing. Technical Report RSTF-003096-03, RST Corporation, Sterling, VA, 1996.

[Pet85] H. Petroski. *To Engineer is Human: The Role of Failure in Successful Design.* St. Martin's Press, New York, NY, 1985.

[PP94] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 58–65, Seattle, WA, 1994.

[TF93] P. Thévenod-Fosse. Statemate applied to statistical software testing. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 99–109, Cambridge, MA, 1993.

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

[VM94] J. M. Voas and K. W. Miller. Putting assertions in their place. In *Proc. Int. Symp. on Software Reliability Eng.*, pages 152–157, Monterey, CA, 1994.