Problems of Accuracy in the Prediction of Software Quality from Directed Tests.

Christoph C. Michael Jeffrey M. Voas

{ccmich,jmvoas}@rstcorp.com
RST Research
Suite #250, 21515 Ridgetop Circle
Reston, VA 20166 USA

Abstract

When software quality is to be evaluated from test data, the evaluation method of choice is usually to construct a reliability model. These models use empirical data and assumptions about the software development process, and they usually result in estimates of the software's proneness to failure. But another paradigm — one that seeks to establish a certain confidence that the software has high quality — also appears occasionally. In this paper, we attempt to give a unifying description of that paradigm, and we discuss its usefulness in software testing.

This analysis method, which we call probable approximate correctness analysis after a similar paradigm in machine learning, emphasizes the accuracy of quality estimates. We will discuss its use in two situations where it has not previously been applied: the case where software is being developed or maintained, and the case where software testing is not representative of the environment where the program will eventually be used. Our goal is to explicate the present usefulness, limitations, and future promise of this developing paradigm.

1. Introduction and overview

The quantitative evaluation of software quality from test results is almost always accomplished with reliability growth models. With such an approach, one estimates the function that relates time and software quality; this is done with a combination of empirical data and model assumptions describing the way faults are detected and repaired. Another, qualitatively different approach has also appeared scattered in the literature on software engineering — one that we will call *probable approximate correctness analysis* after a similar paradigm used in machine learning. This paper is a discussion of probable approximate correctness analysis (or *pac* analysis) in software quality evaluation.

In this type of analysis, the emphasis is not only on estimating the quality of software but also on establishing a confidence that the estimate is accurate. In this sense, *pac* methods differ from those employing reliability models, for although the latter methods sometimes incorporate confidence intervals, those confidence intervals traditionally do not account for the affects of incorrect model assumptions. In contrast, *pac* models seek to establish confidence levels that are less dependent on model assumptions.

This paper begins with a brief description of *pac* methods for software analysis and their relationship to reliability modeling methods. We will then discuss the advantages and shortcomings of both approaches. A particular emphasis will be on areas where *pac* analysis has not been applied extensively in software engineering: the case where software is under development and the case where its quality is evaluated by means of directed testing.

2. Two paradigms for software quality analysis

When software quality is to be evaluated from tests, reliability modeling is arguably the standard approach. This technique is the subject of a large body of literature, [Musa 87] being the standard reference. There is also a large number of specific reliability modeling *techniques*, using different assumptions about the way software changes during development, and using a number of methods to analyze the consequences of those assumptions.

There has not been as much literature on the paradigm we are calling probable approximate correctness analysis, at least in the field of software testing. To the authors' knowledge, this kind of analysis has only been used in settings

where software is no longer subject to repair or modification. Related techniques are often used to analyze partition testing or directed testing, but often those analyses do not result in a software quality measurement. We will argue that the potential of this type of analysis has not been fully exploited.

In this section, we will discuss the details of this second paradigm, and describe how it differs from approaches that use reliability modeling.

2.1. Software reliability models

Traditionally, when it has been necessary to evaluate software quality from test results, model-based methods have been used. Their goal is to derive an abstract mathematical model that distills the essential features of software development, and then use this model to find a function that relates software quality to software development time.

Usually the software-quality function is not known exactly beforehand, but contains several parameters that are to be estimated from empirical data. That is, the parameters are estimated by observing how fast the software's reliability seems to grow in practice. For example, the well-known Jelinski-Moranda model of reliability growth assumes that each fault in the program has the same probability of detection, that the time between failures is exponentially distributed, that faults are removed when they are detected, and that the initial number of faults in the program is finite. With this information, one can get a precise picture of the function that relates software reliability to development time, though to construct the right function, one also needs to know the number of faults in the program and the probability of detecting a fault. These last two parameters are estimated from the *observed* growth in software reliability.

Besides the Jelinski-Moranda model, there are many other models for the growth of software reliability. Most also assume that the function relating software reliability to development time is known up to a few parameters. Thus general shape of this function is obtained from the assumptions made in the model, and after the unknown parameters are estimated from empirical data, the actual function can be constructed.

2.2. Probable approximate correctness

Probable approximate correctness is a paradigm usually associated with machine learning, but on occasion it has also been used for estimating software quality. "Approximate correctness" usually means that the software has a low probability of failure. "Probable approximate correctness" means that one does not know for certain that the probability of failure is low, but that there is statistically a high likelihood of it. Therefore, probable approximate correctness involves two probabilities: the probability of failure, and the confidence that the probability of failure is low (this is also expressed as a probability). The probability of failure is a measure of software quality, while the confidence has to do with the quality of development and testing process.

For example, one can often say "if the software's probability of failure is high, testing is likely to uncover a fault." This statement characterizes a form of *pac* analysis, and it, too, contains two probabilities: the probability of failure, and the probability that testing will fail to uncover a fault.

To make the interplay between these two probabilities more concrete, it is useful to look at the sample spaces used to define them. Recall that probabilities are defined in terms of a *sample space*, which is just a set of objects that can be selected at random. When a program is executed, statistical frameworks assume that the input is randomly selected. Certain inputs cause the program to fail, and the probability of failure is just the probability of selecting such an input. Therefore, when one defines a program's probability of failure, one uses the set of program inputs as a sample space. On the other hand, a program's development and maintenance depends on factors like programming skill and how the program was tested. It is as though a set of tests, a programming team, and numerous other factors were being selected at random to determine how likely it is that the program will be correct. The confidence of getting a high-quality program is defined in terms of a sample space that can contain any of these nondeterministic factors, all playing a role in the development process.

2.3. Example: simple *pac* analysis.

One of the simplest forms of *pac* analysis involves a single program whose quality is to be evaluated by random testing. The technique presented in this example has appeared often in software testing literature, notably in [Duran 84]

and [Hamlet 87].[1] We assume that the program behaves deterministically, and that there are certain inputs for which it fails. The probability that the user will execute the program with one of these failure-causing inputs is the program's *probability of failure*, and we use it as our measure of software quality — a low probability of failure constitutes high quality.

Suppose the tester has the same probability as the user of selecting a failure causing input, and let θ denote this probability. Then the probability that the tester will select a test that does *not* cause a failure is $(1-\theta)$. The probability that the tester will select a series of ℓ independent tests, none of which results in a failure, is $(1-\theta)^{\ell}$.

According to this reasoning, the tester ought to be fairly sure of getting some failures when testing a program that has a high probability of failure. Specifically, if the probability of failure is higher than some constant ϵ, then the probability of not seeing any failures in ℓ tests is no larger than $(1-\epsilon)^{\ell}$. To put it another way, at most one out of $1/(1-\epsilon)^{\ell}$ test suites will fail to reveal a fault. In this sense, if the tester sees *no* faults during the performance of ℓ tests, then we can state with confidence $1-(1-\epsilon)^{\ell}$ that the probability of failure is below ϵ — this is the confidence that testing will reveal faults in a program whose probability of failure is greater than ϵ.

In reading this example, it is important to note that there are two kinds of probabilities involved. There is the probability of failure, which is defined on the set of possible program inputs — it is the probability of selecting a single input and having that input cause a program failure. There is also the probability of seeing no failures in ℓ tests. It is defined on the set of possible *test suites* of size ℓ. It is the probability of selecting such a test suite of size ℓ, executing the program on each test, and never having the program fail.

This example illustrates a recurring paradigm in *pac* analysis, which will appear again below. It was shown that if the program's quality were low (e.g., if its probability of failure were greater than ϵ), then this fact ought to be revealed within a certain number of tests. Therefore, if the stated number of tests can be performed without observing a failure, one can conclude with a certain confidence that the probability of failure is actually less than ϵ.

2.4. The relationship between reliability modeling and probable approximate correctness analysis.

The fact that the *pac* paradigm involves a *probability* of getting high quality software distinguishes it from the traditional reliability modeling paradigm. In reliability modeling, one obtains an expected value for the software quality at a certain time. Under the *pac* paradigm, one obtains a confidence that the quality will not be below some threshold. This distinction is illustrated in figure 1, which shows a hypothetical software quality curve over time. The solid line represents the *expected* software quality as a function of time, as might be obtained from a reliability model. The true reliability curve fluctuates around it. On the other hand, the dashed line represents a lower bound on the software reliability, as obtained from a *pac* model. The true reliability curve is expected to remain *above* the *pac*-derived curve, though there is a small probability that it will be lower at any given time.[2] In short, reliability modeling seeks to provide an average case analysis, while *pac* analysis provides somewhat of a worst case.

There is an important conceptual difference between *pac* analysis and reliability modeling, other than this difference in the respective goals of the two. Reliability models tend to be parametric, meaning that the reliability growth curve is assumed known up to a few parameters. If these assumptions are wrong, it will lead to inaccurate results, but reliability modeling does not usually account for the affects of incorrect model assumptions. On the other hand, *pac* analysis is nonparametric, meaning that it *does* seek to account for the possibility of incorrect assumptions, and, in the case of software analysis, to quantify their affects on the accuracy of the tester's software quality estimate.

But while *pac* analysis is in this sense more robust than reliability modeling, it is also more difficult. A particularly salient problem is that one cannot easily use it to estimate parameters from data: the curve it provides is only a lower bound on the program's reliability, and such a bound is an abstraction not reflected in empirical data. Therefore, the observed data cannot be used to estimate unknown parameters of the curve. Therefore, there is a tradeoff between reliability modeling and *pac* analysis. Reliability modeling makes it easy to estimate unknown parameters from data,

---

[1] The title of this paper was "Probable Correctness Theory," but "probable approximate correctness" more accurately describes its content. We will use "probable correctness" later for a paradigm that more closely matches that description.

[2] In practice, these curves would not be completely straightforward to derive, because most reliability models predict a time-related quality measure like reliability or failure intensity, while *pac* methods predict a probability of failure. To convert between the two requires adding some notion of the probability of failure or abstracting time out of software reliability. However, there are straightforward ways to do this, one of which is simply to say that "time" is the number of executions that have been performed.

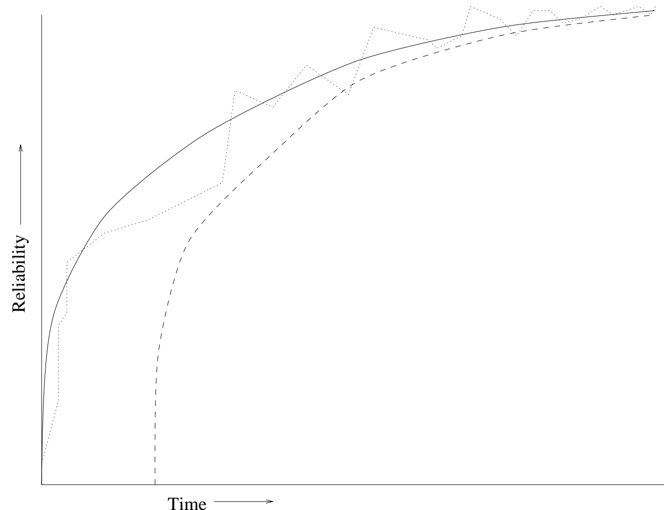

Figure 1: An illustration of the reliability curves obtained by reliability modeling and by the *pac* paradigm. The curve obtained from reliability modeling, shown as a solid line, shows the expected reliability at any given time, and so the true reliability curve fluctuates around it. The curve obtained from *pac* analysis is a probabilistic lower bound on the reliability, and the true reliability curve remains above it most of the time.

but it can provide inaccurate estimates of software quality, and it has difficulty quantifying its own inaccuracy. *Pac* analysis emphasizes accuracy but can present problems in the empirical estimation of unknown parameters.

2.5. Probable correctness analysis in software engineering.

There are two paradigms in the software engineering literature that relate to *pac* analysis. In some papers, like [Hamlet 87] and [Voas 95], the goal is to get a high confidence that the program being tested has high quality. This is what was illustrated by the example in section 2.3. Another approach, used for example in [Duran 84] and [Weyuker 91], is to try to assess the probability that testing will miss faults in a program. These approaches are closely related to *pac* analysis because the probability of *not* missing a fault is just the confidence factor that a *pac* framework seeks to provide.

Although the papers just mentioned assume that the software under test is not changing (it is not being maintained or debugged, for example), that is not a requirement for *pac* analysis. Indeed, machine learning, for which *pac* analysis was developed, involves a situation quite similar to software development. In both cases, there is a program or algorithm that is supposed to perform a certain task but makes mistakes. In both cases the mistakes are uncovered empirically and result in an adjustment of the program. The difference between machine learning and software development is simply that in the former case, a computer program makes adjustments to the erroneous program, while in the latter case, a human software developer makes the adjustment.

This difference is significant because a machine learning algorithm has predictable behavior and restricted power. A human software developer has unpredictable behavior and is much more powerful than an algorithm. Therefore, when software development is modeled with a machine learning paradigm — as in [Chern. 86], [Romanik 95], and [Michael 97c] for example — the approach used is to restrict the developer's behavior, making him or her act like a machine.

In a sense, probable approximate correctness analysis and related approaches are not yet powerful enough to analyze software development in its full generality. However, they can be useful in special cases (such as the case where the software is unchanging), and they can provide important insights into the nature of software development. Nonetheless, there are some advantages to the use of *pac* analysis in software quality assessment. This paper will

discuss its strengths and its current limitations, and attempt to give the reader a picture of its present utility and future promise.

3. Some issues in the analysis of changing software.

The software engineering literature on *pac* and *pac*-like paradigms is almost exclusively devoted to the case where software is not changing. (The past literature relating machine learning to software testing, such as [Chern. 86], did not use the *pac* paradigm, while [Romanik 95], which to the knowledge of the authors is the only paper presenting a framework that allows software to change, did not relate *pac* analysis explicitly to the problem of software maintenance.) Therefore, this section will expand on the use of *pac* analysis during a software development process. Indeed, this is the case that fuels the analogy between software development and machine learning, which was discussed above. The issues that arise in *pac* analysis due to changing software are different from the ones that arise with reliability models, and we will discuss some of them here.

When a software module is under development, *pac* analysis consists of continually estimating the software's quality (or at least having the capability to do so), while maintaining a certain confidence that the quality estimates are close to correct. In other words, we may want to estimate its quality anew many times during its development or maintenance.

Since we already know how to perform a *pac* analysis for software that is not being modified (as in the example of section 2.3), one might think of simply repeating this analysis each time the software is modified during development or maintenance. Unfortunately, this simple approach gets us into a quandary. It turns out that one cannot simply repeat an analysis like that in section 2.3, using the same tests each time. The reason is that if we did so, faults that went undetected by chance during one stage of the analysis would remain undetected at subsequent stages as well, and such faults would accumulate as more and more software modifications were performed. It is also uneconomical to select new tests each time the software is modified, simply because there can be many modifications, and many tests may be needed each time.

However, it turns out that under some conditions one *can* reuse the same tests at each analysis stage; one simply cannot do it the way it was done in section 2.3. The usual approach, used in machine learning, is to adopt some way of measuring how much the program can change during development, for such a measure can be used to find the number of tests that must be in the test suite. A simple measure is just the number of forms that the program is allowed to take during development, and this is what we will use to illustrate this process in the next example.

3.1. Example: a program required to emulate a finite automaton.

For this example, let us assume the program under development is restricted to implementing a finite automaton with k states, an input alphabet no larger than m, a fixed initial state, and a fixed set of final states. For a finite automaton fitting this description, there are at most $k^2(2^m + 1)$ transition functions: there are k^2 possible transitions in the FA, and each one can either be labeled with some subset of the symbols in the input alphabet or it can be omitted altogether. Because of the restrictions on the number of states, and the fixed initial and final states, the number of transition functions determines the number of automata that satisfy the constraint. Thus there are at most $k^2(2^m + 1)$ unique permissible FA's.

The most common *pac* analysis method is to estimate the probability of failure of all permissible programs — in this case all those implementing one of the specified finite automata — at the same time. We will do this too, and we will assume that the program is regression-tested each time it is modified, so that failures uncovered by previous tests do not reappear (that is, when the program is modified, all the old tests are re-executed). Our goal is to ask: "among the programs that are not rejected, what is the probability that one will have a probability of failure greater than ϵ?" If such a program is regression-tested ℓ times before we accept it, then the probability it will survive testing with no failures is $(1 - \epsilon)^\ell$ or less, unless its probability of failure is, in fact, less than ϵ. Therefore, the probability that *some* program whose probability of failure is greater than ϵ will survive regression testing is no greater than the sum of the probabilities that the $k^2(2^m + 1)$ individual programs will survive the tests: it is at most

$$k^2(2^m + 1)(1 - \epsilon)^\ell. \tag{1}$$

With probability $1 - k^2(2^m + 1)(1 - \epsilon)^\ell$, *no* program whose probability of failure is greater than ϵ will survive regression testing, so any choice we make among these programs will be a safe one.

The utility of this approach can be seen by calculating the number of tests needed to achieve some fixed confidence level $1 - \delta$. For example, if we wish to have a confidence of .999 that no program has a probability of failure greater than ϵ, we set δ to $1 - 0.999 = 0.00001$. In other words, δ should be the probability of a program surviving the tests even though its probability of failure is greater than ϵ. Since (1) tells us this probability in the case where there are ℓ regression tests, we can equate this expression with δ and solve for the number of tests. We obtain

$$\ell \approx \frac{2\ln(k) + m\ln(2) - \ln(\delta)}{\ln(1-\epsilon)}.$$

Thus, if we limit the number of permissible programs, and use this limit as a measure of software diversity during maintenance, we find that the test effort only grows logarithmically in the number of permitted programs.[3]

We can use the above technique to find a function that has much of the same utility as a reliability growth curve if we fix some "target confidence," say 0.99. For a given number of tests, we ask how small we can make ϵ in (1), and still maintain a confidence of 0.99 that no program with a probability of failure greater than ϵ will escape testing. If we plot this "smallest ϵ" against the number of tests, we get a curve showing how the probability of failure decays as more and more tests are employed, and as the program is debugged according to the results of those tests. If we were to convert this into a reliability curve, we would find something resembling the curve in figure 1: it would depict a probabilistic lower bound on the true reliability.

More sophisticated measures of diversity during maintenance can be adopted as well. For example, using techniques described in [Goldberg 93], one can use a bound on the time-complexity of the software to determine how many regression tests must be used to keep track of software quality using *pac* analysis.

3.2. Advantages and limitations of *pac* analysis during software maintenance.

In the example above, certain assumptions had to be adhered to for *pac* analysis to work during software development and maintenance. In this case, the assumption was that the program would always implement a finite automaton having no more than k states and m input symbols. Using bound like those of [Goldberg 93], the assumption would have been that the time complexity of the program would never exceed a certain bound.

The assumptions made during reliability modeling are of a fundamentally different nature. The Jelinski-Moranda model assumes that all faults have the same probability of detection and that no new faults are introduced during program repair. More complex models assume that the detection probabilities of the faults are governed by a simple distribution. In general, reliability models make assumptions about how faults in the software behave as well as assumptions about the affects of program repair on these faults.

In contrast, *pac* analysis methods of the kind discussed in the last example make assumptions about how maintenance affects the software itself — they involve the tangible outcome of software maintenance, instead of intangible abstractions such as software faults. In this respect, they have a great advantage over reliability models — their assumptions can be empirically verified (at least in principle). For example, one could restrict developers to a simple language that can only implement finite automata of a certain size, and this would automatically enforce the restriction suggested in our example. In contrast one cannot, even in principle, order developers to write programs with only a certain number of faults, and to *verify* that such a dictum were being followed would amount to solving much of the software maintenance problem.

Unfortunately, the disadvantages of *pac* analysis somewhat outweigh the advantages, at least in the current state of research. The finite-automaton restriction of our example is not reasonable for most software development projects. The time-complexity restriction is more reasonable, but can be expected to lead to an enormous regression test suite for even moderately complex programs. There are classes of programs for which *pac* analysis cannot be done at all; some of them are discussed in [Romanik 95]. For such classes, the regression test suite would have to be infinitely large (or, to put it another way, no finite test suite can ever guarantee any sort of quality for such classes of programs).

In short, *pac* methods give worst-case analyses, and in some cases the worst case is so bad that the technique is useless. The key to solving this problem is that the worst case is not often realized in practice, for if it were we would

[3] More than one program can obviously implement the same finite automaton, but for the purposes of our analysis all programs implementing the same function are counted as one.

continuously encounter development projects where the software was always hopelessly faulty, no matter how much maintenance was performed. Since this is not usually what happens, we can safely assume that for many projects, there is a better way to describe what developers do than saying "the developers will always implement a finite automaton with k states" or "the developers will always write a program that executes $O(n^2)$ time." However, for *pac* analysis to be generally useful, such programmer behavior has to be quantified, and this has not currently been achieved.

4. Some issues in the analysis of directed test results

During software development, testing mainly serves as a way of uncovering software defects. Testing therefore often involves exercising all parts of a program specification, exercising all features of the actual program, or creating unusual conditions that are believed likely to cause a failure. This type of testing is often called *directed testing*.

But in addition to detecting faults, testing should help ensure high software quality, and it can also be used to estimate the software's probability of failure. Tests that work well for these purposes have a different character from those that are good at detecting faults because they must simulate typical usage conditions. Above all, they must provide a representative picture of how often the program will fail in practice. Tests that do so are often called *representative tests*. (Sometimes such tests are also called *operational tests*. Sometimes they are simply called *random tests*, though this characterization is imprecise at best: it glosses over many significant assumptions that go beyond mere randomness, and that will be discussed in detail below.)

Since they will be used to estimate statistical software properties, representative tests are usually assumed to have certain characteristics that make them useful for that purpose. They are usually assumed to be selected independently at random, with replacement, according to the same probability density that will govern the selection of inputs when the program is used in the field.

However, directed tests are rarely selected according to the probability density that will be in place after the program is deployed. After all, they are meant to reveal faults as efficiently as possible, and not to simulate usage conditions. In fact, it has been argued that such a density may not exist in many cases, and even when it does, the tester may not know what it is. Testing according to a representative probability density may not be a high priority in any event, because the tester may believe other test methods are better at uncovering faults.

There are also other less obvious, but equally important reasons why directed tests are hard to use when making statistical software quality estimates: most statistical methods of software quality assessment assume that tests are identically distributed and independent. Identical distribution means that all the tests are selected according to the same probability density, and independence means that the selection of each test does not to depend on any of the other tests. Although these two requirements have not received as much attention as the need for representative tests, one cannot expect directed tests to be independent or identically distributed. The reason for this is that in directed testing, different tests are often meant to exercise different features of the program. To see why directed tests may not be identically distributed, assume that first test in a test suite is supposed to make the flow of control take the "true" branch at a certain conditional statement, while the second test is supposed to make the "false" branch be taken. None of the inputs that might be selected for the first test can also be selected for the second, so the selection of the two test inputs is not done according to the same distribution. The argument for independence is similar: if the second test in a test suite is supposed to exercise whatever branch was not exercised during the first test, then the choice of the second test will depend on what input was chosen for the first test.

Two approaches to the problem of directed testing will be discussed below. One is probable correctness analysis, which seeks to establish that software is probably correct, and not just that it is probably approximately correct. Since this approach has limited applicability, we will also discuss how a general *pac* analysis can be performed when directed testing is used.

4.1. Probable correctness analysis

In probable approximate correctness analysis, one asks how likely it is that a program has high quality. A logical extension is what we will call *probable correctness* analysis, where one asks about the probability that the program is actually *correct*.

Probable correctness analysis can take the form of a hypothesis test, as in [Laski 95]. Here the null hypothesis is

that the program contains a fault, and successful test results are used together with other information to disprove this hypothesis with a certain confidence.

A second approach, used in [Voas 95] and discussed in [Michael 97a] involves a proof by contradiction. Here, *pac* analysis is used in the usual way, to establish an upper bound on the program's probability of failure. Then, some other method — in [Voas 95] it is testability analysis — is used to establish a lower bound on the probability of failure based on the assumption that the software is faulty. If this lower bound is higher than the upper bound, then a contradiction ensues, and the contradiction is resolved by concluding that the assumption of a faulty program, which was used to establish the lower bound, must have been false. However, both the upper and lower bounds are valid only with a certain confidence, and therefore the same is true of the ultimate conclusion that the program has no faults.

In [Bertolino 96], a Bayesian approach was used to find the probability that a program is correct given that it is not rejected during testing, based on a prior belief about the program's correctness and a testability measure similar to that used in [Voas 95].

What distinguishes these methods is that they seek to establish the *correctness* of a program; they do not simply assert that the program has a low probability of failure or a high reliability. Since correctness is not a function of the operating environment, the assertion holds for *all* operating environments, and not just for some.

Upon reflection, this seemingly obvious assertion can lead to some confusion. After all, the program's correctness is supposedly established by tests, and we are using a successful series of tests to conclude that the program is fault free. It seems as though there will be difficulties if the tests are not representative of the operational environment. The solution lies in examining the two types of probabilities involved in *pac* analysis. The first probability is the probability of failure, or some other probabilistic measure of software quality. This probability depends on the operational environment, because if the software has some faulty components, its apparent quality will depend on how often those components are exercised. The second probability is that of selecting tests that lead to a poor software quality estimate. But test selection is obviously a part of testing software, and not a part of using it after it has been deployed. The event of selecting a poor set of tests is an event that takes place in the *test* environment, and not one that takes place in the *usage* environment. In this sense, it does not depend on the usage environment directly, although there can be an indirect dependence if the definition of a "poor test set" depends on the usage environment itself.

If a "good" test set is one that allows us to accurately estimate a program's probability of failure, as in *pac* analysis, then the definition of a poor test set does, indeed, depend on the usage environment. This is because the probability of failure depends on that environment, as noted above. However, if a "good" test set is one that allows us to show the program to be correct, then the dependency on the usage environment vanishes, because correctness itself is not a property of the environment. Therefore, since probable correctness analysis seeks to establish software correctness, it is only the adequacy of the *test* environment that determines the probability of misestimating the program's quality. Thus, if one could always do probable correctness analysis, one could always get quantitative results on software quality from directed testing as easily as one can get them from representative testing.

4.2. A concrete example of probable correctness analysis

Assume that a program is tested with some directed test method that is known to work well; in fact it works so well that if there are any faults in the program, the probability of uncovering them during any given test is at least 0.1. If we make a series of 100 such tests independently and there is a fault, then the probability of failing to uncover a fault is only $(1 - 0.1)^{100}$ or about 2.7×10^{-5}. In other words, if the program contains a fault, there is only about one chance in fifty thousand that a directed test suite will cause no failures. In this sense, we have a confidence of about $1 - 2.7 \times 10^{-5}$ that if there are no failures during testing, then there are no failures to be found.

Note the similarity between this approach and the one used in the previous example, in section 2.3. In the latter case, we saw how to get a high confidence that a program's probability of failure was less than ϵ. In the current example, we additionally assumed it was impossible to get a probability of failure greater than zero but less than ϵ: when we said that our tests were guaranteed to uncover any fault with probability of at least 0.1, we essentially said that any probability of failure greater than 0 but less than 0.1 was impossible. If the tests show that a probability of failure 0.1 or higher is also unlikely, then we can conclude the real probability of failure must be zero.

Another difference between the current example and the one in section 2.3 is that in the first example, we assumed that the tester experienced the same probability of failure as the user. This allowed us to say that if the probability of failure was less than ϵ for the tester, it would also be less than ϵ for the user. In this example, such an assumption is

unnecessary as long as the tester has a chance of selecting any of the inputs. If any input can be selected by the tester and yet there is no chance of selecting one that will cause a failure, it means that no input causes a failure, and so the user cannot select a failure-causing input regardless of the operational environment.

4.3. Limitations of probable correctness analysis.

The problem with probable correctness analysis (as the reader may suppose) is that it cannot always be done. In fact, the existing approaches make assumptions that may not always hold. In [Voas 95], certain assumptions are needed to make testability analysis work, and [Laski 95] uses *dynamic impact analysis*, which is also subject to some assumptions. [Bertolino 96] does not discuss where testability measurements come from, but besides the assumption that a valid testability measurement is available, an accurate prior belief about program correctness is required.

The use of extra assumptions leads to especially serious problems with probable correctness analysis. Those problems are most visible if one interprets the analysis as a proof by contradiction, as [Voas 95] does. In this case, it is simply not clear which assumption is disproved when the lower bound on the probability of failure is higher than the upper bound. Is it the assumption that the program contains a fault, as desired, or is it one of the other assumptions needed to make the analysis work? The same problem exists in the hypothesis-testing approach of [Laski 95], although it is less obvious. The hypothesis that the program is faulty is disproved if there are many successful tests *and* the other assumptions of the framework hold. Therefore, in a sense, proving these other assumptions is as important as having a series of successful tests.

In addition to assuming a testability measurement is available, the Bayesian approach of [Bertolino 96] assumes that the tester can come up with an accurate *prior* estimate of the probability that the program is correct. If this prior belief is wrong, the ultimate conclusion about the program's correctness can be wrong too. Therefore, if this technique predicts that a program is correct with high probability it may indicate that the program is really likely to be correct, but it may also indicate that the prior belief was incorrect.

The problem with all of the probable-correctness approaches can be summarized with a simple thought experiment. Suppose a developer on a large, complex software project was told that the program was correct with probability 0.9999 according to one of the aforementioned models, assuming its assumptions were right. The developer can conclude that the program is almost certainly correct, or conclude that the assumptions of the model are wrong. Practical experience with large, complicated pieces of software suggests that they are almost never fault-free, so the developer would probably make the second conclusion: the model's assumptions are at least partially wrong.

In summary, although probable correctness analysis demonstrates one way of estimating software quality with directed tests, it appears to be an unrealistic way. Because of this, we ought to investigate other avenues as well, and we do this below.

4.4. *pac* analysis of directed tests.

Most quantitative analyses of directed testing have described it using *subdomains*, which are simply subsets of the set of possible program inputs. They are a useful tool because directed testing is almost always guided by some set of adequacy criteria, and these criteria can be mapped to subdomains. Some examples of adequacy criteria were given in section 4: such a criterion might require a certain statement in the code to be executed or a certain conditional branch to be taken. For any given criterion, one can divide the inputs into those that satisfy the criterion and those that do not. The ones that satisfy it comprise the subdomain associated with that criterion. Subdomains were used for directed test analysis in [Duran 84], [Weyuker 91], [Frankl 93], and elsewhere.

The technique we introduce in this section is similar to those that make use of subdomains. In subdomain analysis, it is usually assumed that the tester selects some inputs at random from the first subdomain, then selects some from the second subdomain, etc., thereby performing directed testing. Our analysis will be similar, but instead of assuming that tests are selected at random from each of several subdomains in turn, we will assume that they are selected according to several probability density functions in turn. This is a generalization of the subdomain-based view, for we can represent a subdomain by a density that assigns a zero probability to any inputs outside of the subdomain.

Our analysis will consist of two components. First, we ask what we can conclude about the program *in the test environment* if the tester sees no failures during testing. Second, we convert this information into a form that is

meaningful to the software user, who judges software quality by its probability of failure in a different environment not known at the time of testing.

We can proceed as we did in the example of section 2.3, but instead of trying to show that the probability of failure is below some constant ϵ, we try to show that the expected proportion of failing tests is less than some constant ξ when directed testing is used. If we assume tests are selected independently (though they are not identically distributed), it can be shown that if this proportion is greater than some constant ξ, then the probability of selecting a directed test suite that causes *no* failures is at most

$$\frac{2^{-\ell\xi/2}}{1-e^{-2}}, \tag{2}$$

where ℓ is the number of directed tests [Michael 97b]. A similar but more complicated result can be obtained when tests are not independent, but we omit that result here.

Therefore, if we perform directed testing as above, and see no program failures, we use the same arguments as in section 2.3 to conclude that the expected proportion of failures was less than ξ with confidence $1 - 2^{-\ell\xi/2}/(1-e^{-2})$. That is, if the tester repeats the directed test process, selecting new inputs but using the same probability densities and selecting the same number of tests from each of them, we can confidently expect that the proportion of failures will be less than ξ.

It is now necessary to translate this result into a meaningful quality estimate. The tester selects tests according to one or more probability densities, but the user selects inputs according to yet another density, and we must adjust our results to account for this fact. In the software engineering literature, this process has become known as *profile adjustment*, because probability densities used to select program inputs are often called *operational profiles*.

In the software engineering literature, profile adjustment has been accomplished by a process very similar to the one used in directed testing. The inputs are once again divided into subdomains, but now the subdomains correspond to *atomic operations*. These are operations whose probability of causing a failure only depends on how often they are used. If a change in the operational profile does not change the frequency of usage for such an operation, that operation's contribution to the overall probability of failure will not be affected, while if the frequency of use *does* change, the operation's contribution to the overall probability of failure is just the former contribution under the old operational profile, multiplied by an appropriate constant. An example of this is the seat-selection algorithm of an airline reservation system. It may be that the seat-selection algorithm always fails exactly one percent of the time, so that it's contribution to the system's overall probability of failure only depends on how often seat-selection is invoked in the first place. If the seat selection algorithm is invoked with probability 0.3, we can conclude there is a probability of 0.03 that seat-selection will lead to a failure: there is a 0.3 probability that the seat-selection operation is invoked, and 0.1 probability that it will cause a failure after being invoked. If, at some later time, the usage conditions change so that seat-selection is only invoked five percent of the time, we can adjust our model easily: the probability of seat-selection is now 0.05, but the probability that seat selection will cause a failure once it is invoked remains 0.1 by assumption. Thus the overall probability that seat selection will cause a failure is just 0.005. This type of approach was employed by [Miller 92], [Musa 94], and [Adams 96], for example (although the process is more complicated in the latter two papers because they use failure intensity, a time-related quality measure, instead of the probability of failure).

However, we may need many subdomains to ensure that each one really represents an atomic transaction. If there are too many inputs in a subdomain, then a change in the operating environment may change the probability that inputs in the subdomain will lead to a failure. For example, it may be that the seat-selection algorithm tends to fail when the desired seat has a high seat number (perhaps the seat number overflows into the sign bit). If customers one day decide they like flying in the back of the airplane, then the seat-selection algorithm will fail a greater proportion of the time. The atomicity we need will then be destroyed: the probability of a seat selection failure can no longer be obtained by merely compensating for increases and decreases in the frequency of seat-selection requests. To avoid problems like this, there may have to be few inputs in each subdomain, and so the number of subdomains may have to be large.

Furthermore, the operational profile may initially be unknown. If this is the case, the probability of each subdomain will have to be estimated from data, and the more subdomains there are, the more data will be required to get an accurate estimate for each probability.

Since a *pac*-style analysis emphasizes accuracy, it may be preferable to circumvent this problem by using a single parameter that quantifies the largest deviation between the usage density (the probability density governing the user's selection of inputs) and the densities that were used in testing. Although this approach leads to a loss of information, it may be preferable because a single parameter is usually easier to estimate than an ensemble of many parameters. Indeed, by comparing information about the usage density with the pattern of software usage during directed testing,

one can get a single constant that relates the user's probability of failure to the tester's expected proportion of failures. For example, if θ is the probability of failure experienced by the user, and ν is the expected proportion of failures experienced during directed testing, then a technique of [Michael 97b] results in a constant ψ such that $\nu \geq (1-\psi)\theta$. If we have such a constant, we can use (2) to assert that if the probability of failure under the usage density is greater than ϵ, then the probability of finding no failures in ℓ *directed* tests will be at most

$$\frac{2^{-(1-\psi)\ell\epsilon/2}}{1-e^{-2}}. \tag{3}$$

That is, we have a confidence of $1-(2^{-(1-\psi)\ell\epsilon/2})/(1-e^{-2})$ that a program will not survive testing if its probability of failure is greater than ϵ.

However, a single parameter measuring the worst-case deviation between the test conditions and the usage conditions of a program is easier to estimate than many parameters describing the relative probabilities of many atomic transactions. This reflects a paradigm often encountered in *pac* analysis: one tries to simplify the problem as much as possible before solving it, for the sake of a more accurate analysis.

Note that because of the simplicity of the technique discussed previously in section 3, where we dealt with the case of software under development, (3) can be plugged into that analysis directly. Since the probability that testing will cause no failures in a single faulty program is less than $2^{-(1-\psi)\ell\epsilon/2}/(1-e^{-2})$, the probability that testing will cause no failures in any of N faulty programs is less than $N2^{-(1-\psi)\ell\epsilon/2}/(1-e^{-2})$ by the same argument that was used in section 3. Returning to the example involving the finite automaton in section 3.1, we can now assert that if any program emulating a finite automaton with k states and m input symbols has a probability of failure greater than ϵ, then the probability of getting no failures during ℓ directed tests is less than $k^2(2^m+1)2^{-(1-\psi)\ell\epsilon/2}/(1-e^{-2})$.

5. Discussion

The analogy between software development and machine learning suggests that the techniques of the former paradigm, especially that of *pac* analysis, might be useful in software assessment as well. In this paper, we have discussed the utility of this approach.

Since it is usually associated with machine learning, research in *pac* analysis has not concentrated some issues that are important in software engineering. The most important of these is the analysis of software under development, which was discussed in section 3. The *pac* technique we presented there was not a very satisfactory solution to the problem, although the main intent was only to show that it can be applied to this case in principle, rather than to develop a refined approach.

We have also discussed the use of *pac* methods in the analysis of directed tests: directed testing is another paradigm associated with software engineering and not so much with machine learning. Indeed, one approach to this problem, probable correctness, has already been suggested several times in software engineering literature (but never in the literature on machine learning, as far as we are aware). Since this approach has severe limitations, we also discussed a more general technique.

Modularity is a salient feature of *pac* analysis, at least in the forms it takes in the current paper. It was quite straightforward to adjust the results for directed testing to account for differences between the test environment and the usage environment. The result was easily swapped into the analysis of software under development, replacing a simpler sub-formula that originally came from the example in section 2.3.

We believe there are situations where *pac* analysis is more appropriate than reliability modeling for software quality assessment. One example is the case where software is evolving for reasons other than debugging, making it difficult to model the software modification process (see [Michael 97c]). Indeed, there may be many cases where a *pac*-style analysis may provide useful results, but where its utility is still unknown because of the early stage of research in *pac* software analysis.

References

[Adams 96] Adams, T., "Total Variance Approach to Software Reliability Estimation," *IEEE Transactions on Software Engineering* Vol. 22, No. 9, pp. 687-688, September, 1996.

[Bertolino 96] Bertolino, A., Strigini, L., "On the use of testability measures for dependability assessment.," *IEEE transactions on Software Engineering* Vol. 22, No. 2, pp. 97-108, February, 1996.

[Chern. 86] Cherniavsky, J. C., Smith, C. H., "A Theory of Program Testing with Applications," *Proceedings of the Workshop on Software Testing* pp. 110-121, July, 1986.

[Duran 84] Duran, J., Ntafos, S., "An Evaluation of random testing," *IEEE Transactions on Software Engineering* Vol. SE-10, pp. 438-444, July, 1984.

[Frankl 93] Frankl, P. G., Weyuker, E. J., "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Transactions on Software Engineering* Vol. 19, No. 3, pp. 202-213, 1993.

[Goldberg 93] Goldberg, P., Jerrum, M., "Bounding the Vapnik-Chervonenkis Dimension of Concept Classes Parameterized by Real Numbers," in *Proceedings of the 1993 Workshop on Computational Learning Theory.* New York, N.Y.:ACM Press, pp. 361-369, 1993.

[Hamlet 87] Hamlet, R., "Probable Correctness Theory," *Information Processing Letters* Vol. 25, pp. 17-25, April 1987.

[Laski 95] Laski, J. W., Szermer, W., Luczycki, P., "Error Masking in Computer Programs," *Software Testing, Verification, and Reliability* Vol. 5, pp. 81-105, 1995.

[Michael 97a] Michael, C. C., Voas, J. M., "The Ability of Directed Tests to Predict Software Quality," Technical report RSTR-96-003-03, RST Corporation, 1996.
To appear in "Annals of Software Engineering," Aug. 1997

[Michael 97b] Michael, C. C., "Reusing Tests of Reusable Software Components," in *Proc. COMPASS '97.* : 1997.
To appear.

[Michael 97c] Michael, C. C., "Using evolution constraints to assess the failure-proneness of evolving software," in *Proc. 1st Euromicro Working Conference on Software Maintenence and Reenginerring.* : 1997.
To appear.

[Miller 92] Miller, K., Morell, L., Noonan, R., Park, S., Nicol, D., Murrill, B., Voas, J., "Estimating the Probability of Failure when Testing reveals no failures," *IEEE TSE* Vol. 18, No. 1, pp. 33-43, 1992.

[Musa 87] Musa, J. D., Iannino, A., Okumoto, K., *Software Reliability Measurement, Prediction, Application.* New York:McGraw-Hill, 1987.

[Musa 94] Musa, J. D., "Adjusting Measured Field Failure Intensity for Operational Profile Variation," in *Proc. 5th International Symposium on Software Reliability Engineering.* Monterey, CA.: pp. 330-333, Nov. 1994.

[Romanik 95] Romanik, K., Vitter, J. S., "Using Vapnik-Chervonenkis Dimension to Analyze the Testing Complexity of Program Segments," *Information and Computation* to appear.

[Voas 95] Voas, J., Michael, C., Miller, K., "Confidently Assessing a Zero Probability of Software Failure," *High Integrity Systems Journal* Vol. 1, No. 3, 1995.

[Weyuker 91] Weyuker, E. J., Jeng, B., "Analyzing Partition Testing Strategies," *IEEE Trans. on Software Engineering* Vol. 17, No. 7, pp. 703-711, Jul. 1991.