

New Quality Estimations in Random Testing

S. Mankefors, R. Torkar and A. Boklund

Dept. of Informatics and Mathematics

University of Trollhattan/Uddevalla

P.O. Box 957, SE-461 29 Trollhattan, Sweden

{stefan.mankefors, richard.torkar, andreas.boklund}@htu.se

Abstract

By re-formulating the issue of random testing into an equivalent problem we are able to introduce a new kind of quality estimations based on Monte Carlo integration and the central limit theorem. This method also provides a limited but working “success theory” in the case of no detected failures. In an empirical evaluation using hundreds of billions of simulated tests we furthermore find a very good match between the quality estimations presented in this article and the true failure frequencies. Both simple modulus defects as well as seeded defects in two extensively employed numerical routines were subject to investigation in the empirical work.

1. Introduction

The testing and evaluation of software is of major importance for all software and has been the subject of research for some 30 years. Although exhaustive testing in principle can prove software correct, in practice most software has to be tested in some approximate way, trying to uncover most of the faults. Even highly tested systems like the NASA Space Shuttle Avionics software displays a fault density of 0.0001 defects/line of code (at a cost of 1,000 USD/LOC), which is considered to be close to what is achievable with modern methods [1].

In this respect one of the most difficult questions is the issue of to what extent a test can be trusted, i.e. the “quality” of a test. The current state of affairs is not entirely satisfactory since most methods are not possible to quantify in detail at the moment, while others do not compare easily with each other (see [2,3] for a critical review). Recent theoretical efforts have, however, produced some good results [4,5] in this field. Taking a more pragmatic point of view, a large number of experimental evaluations have been performed of test methods in order to try to compare the quality of different test approaches, see e.g. [6-12]. There are also new approaches to improve the choices of test data using data dependence analysis [13] during execution as opposed to a pure flow analysis.

Although inspiring, more exact results for many methods are lacking. Put in a different way, paraphrasing Dick Hamlet [14], a “success theory is missing”; there are no good ways to determine how good a test that found nothing was. In the area of coverage testing [11] and [14] do provide theories towards this aim, but since absolute coverage does not in itself constitute fault free software they only provide partial answers.

Given the context, statistical test methods (“random testing”) are unique in that they do provide a type of answers about possible remaining faults or execution errors [15]. The downside with this information is its probabilistic nature and interchange-ability between failure probability and test reliability. In addition to this, random testing suffers from other well known inadequacies, e.g. the inability to detect rare errors. Despite this situation, statistical testing remains one of the major testing approaches, if nothing else as a sort of scientific base line. Random testing does also provide massive quantitative data, something usually lacking otherwise. Finally random testing may very well in many circumstances outperform other test methods per *time unit* – if not per test – due to the simpler test case construction (random input). It is thus of significant importance to extend the current work on random testing to include solid quality estimations and eliminate the reliability-probability evaluation problem as far as possible.

In this paper we present distinct quality estimations and results for random testing, enabling much improved use and interpretation of a random test suite. We subsequently make this quality estimation subject of empirical scrutinizing to ensure the strength of the results using massive simulations encompassing hundreds of billions of tests. The remaining parts of this paper are consequently divided into a more general overview of random testing, fundamentals for quality estimations and empirical evaluations of the suggested estimation using massive test simulations.

2. Pro and cons of random testing

There is no absolute consensus on what statistical testing (or fault detection) is. Partly is this due to the lar-

ge number of available methods, e.g. modern methods in code inspection [16] are based on a classical statistical approach. To be clear on this point we will from now on strictly refer to test cases with randomized input and comparison of the output to a well known correct answer as random testing (this follows e.g. [15]).

Even so, the issue of the exact use of random testing still remains open. Why use random testing at all? Intuitively it is clear that by choosing input on a random basis you always come up short compared to e.g. equivalence partitioning [17] or someone inspecting the code or specification closely and constructs the test based on this information (unless a systematic error is made in the non-random approach). It should also be clear that random testing usually samples only a very small fraction of all possible input. A straightforward if-statement that compares two long integers and is executed if found equal stands a chance of 1 in 4E9 to be exercised for a single random test input.

```
a=random(1);
b=random(2);

if (a==b) then
{ some LOC...
}
```

Figure 1. The if-statement is highly unlikely to be executed by straightforward random testing.

On the other hand, similar arguments, although not as strong, can be made about coverage testing. Borrowing the following example from [2], see fig. 2, we note that branch testing with the input value of (3,2) will not result in a failure, while values (2,3) is going to result in an erroneous result.

```
int multiplier(int n, int m)
{
  int result=0;
  while (n>0)
  {
    result=result+n;
    n=n-1;
  }
  return result;
}
```

Figure 2. Example of code that gives misleading branch-coverage result.

The strongest argument in favor of random testing is that mathematical tools provide a possibility to calculate the successes as well as the shortcomings of the method. Blunt or not, random testing possess a potential industrial robustness. Given a way to fast and easily check if the output result (or operation) is correct –

something that usually is referred to as the *Oracle Assumption* [15] – random testing is both easy and straightforward to implement. This combined with its random nature (more about this later) allows for extensive, mathematical analysis. Short of AI-like testing approaches, mathematically inspired methods still remain the most powerful tools available in testing. Possibly advances in genetically programmed testing algorithms [18, 19] will render this kind of basic tools superfluous, but at the moment it would be hazardous to draw that conclusion.

An important issue at this stage of reasoning is the actual users (machine or human) input and the input used in random testing, something that has been improved on [20] and well examined [21]. Although it is well enough to consider a test suite of 1,000 test cases with correct behavior as an indication of that the routine, or similar, usually behave well (i.e. has a low probability of returning the wrong result on a general input), this has not to be true for all input subspaces. Differently put, faults in software are not any more random than, say cards in a game of poker. There is a definitive answer to whether player X has the ace of spades, and it will not change from time to next if nothing happens in between in the game (corresponding to no one re-writing the code).

This also means that a method may give completely correct results for all long integer inputs except in the range of 0-100. Given uniformly distributed non-biased input from the user, the method will behave correctly 39,999,999 times out of 40 million. In this circumstance – unless the method is executed extremely often – the software at hand can be considered as next to flawless. The mean time to failure (MTTF) will be closing in on one year of continuous operation if the software has an execution time around 1 second. On the other hand random testing will most likely – unless used massively – not uncover the fault at all. In the perspective of uniform input this is correct: the *failure frequency* is extremely low (1 in 40 millions).

Unfortunately this line of reasoning may be very misleading in certain circumstances. If the user is human, the application happens to be a computer based calculating aid and the user is free to choose input, it is intuitively clear that the probability for the user to choose low numbers as input is extremely much higher in everyday life than to choose high end numbers, e.g. between 1,907,654,1000 and 1,907,654,1100 (which has the same input range as 0-100). Hence such software, subject to a well-performed random test suite with asserted high MTTF based on uniform input will still be perceived by the user as quite flawed due to the non-uniform nature of the user profile.

It is therefore of outermost importance that the testing conditions, and especially the use of a certain distribution of random input, is clearly defined and declared for any tested software. The necessity of clear definitions and routines is a general truth in all testing activities, but

unusually important in random testing because of the complex mathematical nature.

A direct consequence of the poor sampling of rare events is that some extreme values in a method's or object's input/attributes will never, or at least very rarely, be tested using random methods. This is especially true since pseudo random number generators (PRNG) usually are used to produce input in random testing. Given the fact that PRNG:s produce output in a certain legal range, illegal values are not possible to obtain (unless the legal range is larger than the input, e.g. long integers are used to probe an input domain of short integers). The wisdom to draw from this is that extreme and illegal values should be tested in connection to, but separately from pure random testing of legal input, see e.g. [6, 18] for a practical example.

Despite these problems, random testing – given properly handled input distributions – have certain great advantages as pointed out earlier. Random test cases are usually very fast to produce due to the automatic and straightforward nature of the PRNG:s generating the input. Given a routine and corresponding oracle that perform reasonably well and execute in a few seconds, random testing may cover tens of thousand cases in 24 hours. Automatic over-night testing can give tabulated test results “en masse” for tested software units.

It has to be admitted though that the oracle assumption makes random testing much more difficult above unit-level. At the same time is random testing not, and has never been, suitable for subsystem or system testing: the possible combinations of input is so huge and almost always subject to a specialized user input that the random test suite ends up probing the wrong corner almost regardless of its size, unless it is uniquely tailored for the system.

Random testing hence offers advanced brute force testing of units where straightforward oracles are available. The results have to be carefully used, but as one of very few methods, random testing offers the possibility of genuine mathematical analysis.

3. Fundamentals of quality estimations

Quality estimations of random testing has traditionally been connected to classical statistical instruments, i.e. confidence levels and believed performance.

Given enough time we would find the true failure frequency θ_i for uniformly distributed random input:

$$\theta_i = (\text{number of test cases causing failures}) / (\text{the number of all tests performed}) \quad (1)$$

Normally θ_i is only possible to determine exactly by exhaustive testing, or approximately using massive testing. It should also be noted that we, due to the assumption of uniform distributions, do not care about in

which sub-domain of the input the failures take place – all input is equally probable, and hence the *related* failures too (with a different distribution the different inputs would have to be weighted or measured differently).

In most software it is not possible to even come close to a value of θ , determined by exhaustive testing. Given a single float number as input, there are 4E9 different possible input values (see sec. 2). By testing the software, using a limited number of randomly chosen inputs it is possible to apply statistical analysis to the results for a quantitatively controlled estimation of θ , though.

When M failures are found by running N tests, Nelson's probabilistic theory [22] gives that a guessed (estimated on the basis of the tests) failure frequency θ_g has an upper confidence bound α according to:

$$1 - \sum_{j=0}^M \binom{N}{j} \theta_g^j (1-\theta_g)^{N-j} \geq \alpha \quad (2)$$

The confidence bound determines the probability of the test suit being a representative one, while θ_g is the software engineer's guess of the true failure frequency. To be close to certain that the tests are “good” means that the software engineers become restricted in the range of failure intensity they can guess at: a “squeeze play” [14] takes place between testing and failure probabilities.

Although straightforward, the above theory fails to offer unambiguous quality estimation but leaves extensive parts of the interpretation to the testing crew. Secondly the nature of testing makes a statement like “Given the failure frequency θ_g the probability for experiencing no failures at runtime ten times in a row is $(1-\theta_g)^{10}$ ”, highly dubious if there are input domains with very high (or absolute) failure intensity. Strictly speaking the combinatorial term would be domain dependent and from a random input perspective, subject to a non-trivial probability distribution in itself. More extensive reviews of the constant failure frequency assumption and domain-partitioning problem can be found elsewhere, see e.g. [14, 15]. As we will see, there exists an alternative formalism that avoids this kind of reasoning though.

3.1. Failure Functions

Given the nature of software and the problem of the under decided solutions above, we choose to introduce the concept of “failure functions” as a mathematical tool. Intuitively it is clear that a piece of software either fails or succeeds for a specific set of input, regardless of the input type (numeric, text, logic etc.) and software at hand. This process is non-random, and very much repeatable. It also matches existing mathematical problems: Integrals.

To be able to be more exact in our reasoning we start out by defining the input space \mathbf{X} for a specific software S .

Definition 1: A software S ' regular input set \mathbf{X} is defined to be all combinations of values legal with respect to the input variable types of S .

The above definition is neither "white box" nor "black box", since the regular set (input domain) is defined by the legal input types. This means firstly that exceptions and overflows are not included in the regular domain, but should be treated separately, see [6] for an example. Secondly the domain *could* be concluded from the specifications, given a detailed enough level of description. But it could *equally well* be defined by information collected in a white box process.

Definition 2: Each software S has a failure function F_S defined on the regular set \mathbf{X} . $F_S(x) = 0$ for a given value $x \in \mathbf{X}$ if no failure occur, and $F_S(x) = 1$ if a failure is detected when the software is executed.

The function F_S match the behavior of the software exactly, although detailed knowledge of the function can only be obtained with exhaustive testing. Assuming that the regular input space is large enough to let us approximate F_S with a continuous function (this rules out e.g. pure logical input domains though) and using integration instead of summation, we find that:

$$\theta_t = \frac{\int_{\mathbf{X}} F_S dx}{\int_{\mathbf{X}} 1 dx} = \frac{\int_{\mathbf{X}} F_S dx}{|\mathbf{X}|} = \langle F_S \rangle \quad (3)$$

The bracketed term implies the mean value of F_S . Finding the true failure frequency (the mean value if exact in eq. 3) hence becomes identical to integrating (or sum up, if we let go of the continuous approximation) the failure function and dividing by the size of set \mathbf{X} . Although trivial as far as relations go, this enables a new variety of tools. It also allows us later on to go beyond the somewhat limited use of just finding θ_t . It should be noted though that this equivalence is only meaningful for input with a reasonably large legal span of values (integers, strings, floats etc.) Otherwise the small size of $|\mathbf{X}|$ will render the notion of average and continuity quite meaningless.

Turning to the integration (summation) of F_S we notice that we somehow have to integrate (sum) an unknown function: a problem identical to the evaluation of random testing (since it is just another formulation of it), see fig. 3.

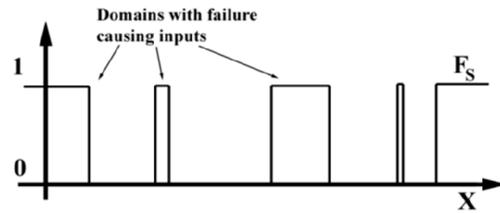


Figure 3. The general problem of integration of a failure function F_S and the relation to testing.

Unfortunately the "un-smooth" nature of the failure function limits the number of available tools for integration. One of the most straightforward methods, Monte Carlo integration [23], applies well, however. The idea is to sample the function (F_S in our case) by random input, calculate the mean and multiply it by the volume the integral is defined on. More strictly we have:

$$\int_{\mathbf{X}} F_S dx = |\mathbf{X}| \langle F_S \rangle + \text{Err. terms} = |\mathbf{X}| \left(\frac{1}{N} \sum_{i=1}^N F_S(x_i) \right) + \text{Err. terms} \quad (4)$$

The $\langle F_S \rangle$ term is the mean value of the failure function in the N randomly chosen x_i points in the regular input set $|\mathbf{X}|$. The expression is of course of limited use before the mathematical error terms are estimated. It simply indicates that if one by random choose a large number of points in the legal range of the failure function and take the average, one should come in the vicinity of the absolute average (compare eq. 3). Drawing on our established correspondence to random testing, eq. 4 just states the fact that an approximation of θ_t is the number of failures found, divided by the number of tests performed.

3.2. Quality estimations: elementa

The definitions and equations in section 3.1 merely cast the problem of random testing in a different form, although it circumvents some of the more traditional formulations and associated problems. All the more interesting is the estimation of the mathematical error terms in eq. 4. As long as each input x_i is chosen statistically independent of each other, the exact nature of how representative the associated $F_S(x_i)$ values are (i.e. the probability distribution of errors) is irrelevant if the number (N) of input values x_i is large enough relative the input space (typically a number of hundred up to a few thousand of tests).

In this case the *central limit theorem* guarantees that the resulting probability distribution of errors (mathematical mismatch) will be close to a Gaussian

distribution (see e.g. [24] for a proof and explanation of the central limit theorem). Differently stated will random testing with its randomized input ensure that the mathematical error terms in eq. 4 approximately follow a Gaussian distribution.

This approximation is valid for all cases when the number of tests is large relative to the input space, i.e. at least decently samples each input variable space. This is an extremely important point to be made since it eliminates any assumption about “evenly distributed execution failures” or similar which has troubled some of the earlier results in the literature, see e.g. [14, 15] for a discussion.

Although this could have been argued for without using the formalism of integrals, we will retain the form due to reasons revealed later in the text. In accordance with standardized statistical estimations, we identify the mathematical uncertainty (error) in eq. 4 with the number (k) of allowed standard deviations (σ):

$$\int_{\mathbf{X}} F_S dx \approx |\mathbf{X}| \langle F_S \rangle \left(1 \pm \frac{k\sigma}{N \langle F_S \rangle} \right) \quad (5)$$

Taking all values within one standard deviation σ (i.e. $k=1$) from the mean value $\langle F_S \rangle$ covers 68% of the statistical possibilities, while using two standard deviations ($k=2$) covers 95%, $k=3$ covers 99.7% and $k=4$ approximately 99.99% (see [25] or similar for tables). Although we a priori do not know the variance ($=\sigma^2$) of the failure function F_S , the sampling of F_S provides a good approximation:

$$\sigma^2 \approx N \left[\left(\frac{1}{N} \sum_{i=1}^N F_S^2(x_i) \right) - \left(\frac{1}{N} \sum_{i=1}^N F_S(x_i) \right)^2 \right] \quad (6)$$

In standard short hand notation this becomes:

$$\sigma^2 \approx N (\langle F_S^2 \rangle - \langle F_S \rangle^2) \quad (7)$$

The average is taken over the N sampled input points. It should be noted that given a very large N approaching the size of the regular input set, this relation becomes exact. Now, using eq. 7, eq. 5 transforms into:

$$\int_{\mathbf{X}} F_S dx \approx |\mathbf{X}| \left(\langle F_S \rangle \pm k \sqrt{\frac{\langle F_S^2 \rangle - \langle F_S \rangle^2}{N}} \right) \quad (8)$$

Since the failure function F_S in our case only takes on the simple values of 1 and 0, $\langle F_S^2 \rangle$ becomes degenerate with the value of $\langle F_S \rangle$. Hence for uniform input distribution we have the following result:

Result 1: If a software S with regular input set \mathbf{X} and an associated failure function F_S exists and the number of test inputs is large enough, the total number of failures for S on \mathbf{X} are given within k standard deviations by:

$$\int_{\mathbf{X}} F_S dx \approx |\mathbf{X}| \left(\langle F_S \rangle \pm k \sqrt{\frac{\langle F_S \rangle - \langle F_S \rangle^2}{N}} \right) \quad (9)$$

Knowing the average failure rate in N tests then immediately returns the quality estimation. As a practical example, using the limit of $k=4$ standard deviations (returning a coverage certainty of 99.99%) 2,300 tests and 45 found failures at runtime, we get an estimated failure rate of $1.95\% \pm 1.15\%$ (i.e. a maximum of 3.1%). The *absolute* number of faults increases linearly, however, as does the quality uncertainty with the size of the regular input set.

At this stage it is important to stop to scrutinize the validity of the approximations made so far. The central limit theorem returned the result above under the assumption of a “large enough number of input values”. Hence the results so far are mathematically valid in all cases where the statistics is good, i.e. many test cases are performed and a reasonable amount of failures are found. The “demand” for failures arise from the fact that the number of input values is not only important as an absolute number by itself but also relative the number of observed failures in eq. 9 (see the discussion below). In the case of poorer numbers, the approximation becomes less precise.

3.3. Lower end quality estimations

The approximate approach to the full-scale probabilistic problem presented here offers a great reduction of complexity in the evaluation of random testing, enabling solid hands-on quality estimations and calculations easy enough to do on a sheet of paper. It also offers straightforward tabulating of the quality of software units, as well as robustness in performance as the empirical evaluation below shows. The draw back is that the result above does not cover the lower end of the failure frequency in a strict mathematical-statistical sense

In order to extend the quality estimations to encompass the full range of frequencies we identify the limitations of the approximation and adjust it accordingly below. The necessity of this becomes self-evident considering the lower boundary of the quality estimation in eq. 9: given few enough failures the lower quality boundary will become less than zero.

This is a pure complication due to too few input values since σ is divided by the square root of N relative the average of F_S . That is, in the case of few found failures, going from 200 recorded tests to 400 tests will

in general do little to the found average, but reduce the boundary term by a factor of $\sqrt{2}$. The normal software engineer could not be expected to extend the number of test cases in order to meet the demands of an academic result though.

Instead we conclude that if the lower boundary given in eq. 9 falls below zero, a reasonable quality estimation to make within the current approach is to assume that the entire quality boundary is given by $[0, 2k\sigma]$. This ensures that the total span of the quality estimation remains intact, while the failure frequency might be overestimated, but hardly the other way around. More precisely stated we get:

Result 2: If a software S with regular input set X and an associated failure function F_S exists which has been probed by N test inputs, the total number of failures for S on X are given within (at least) k standard deviations by:

$$\int_x F_S dx \approx |X| (\langle F_S \rangle \pm C); \sigma = \sqrt{\frac{\langle F_S \rangle - (\langle F_S \rangle)^2}{N}}$$

$$\langle F_S \rangle - C = \max(0, \langle F_S \rangle - k\sigma)$$

$$\langle F_S \rangle + C = \max(\langle F_S \rangle + k\sigma, 2k\sigma)$$

(10)

For normal failure frequencies eq. 10 becomes identical to eq. 9, but in the case of very small frequencies it shifts the quality boundary upwards to avoid a “spill-over” to negative (and hence impossible) values. More strictly put eq. 10 express the statistical limits of eq. 9 in conjunction with the provided tests in each individual test suite.

This limitation is also directly connected to the quality *demands* raised by the software engineer. Obviously the “spill over limitation” will be enforced for smaller frequencies if the software engineer chooses a “2 standard deviation quality” boundary, as compared to 4 standard deviations (compare the maximum functions in eq. 10). The reason for this seemingly strange behavior is an intrinsic part of the statistical approximation used here. If we want the statistical results to be true for 3 or 4 standard deviations, the demands on the statistical quality, and hence the test suite itself, simply grow essentially larger compared to if we are satisfied with 1 or 2 standard deviations.

Although the results in eq. 10 seemingly just provides a crude cut-off in the lower end of the available failure frequency range, it in reality compresses the very same “cut-off” together with the falling failure frequency, something that is observed empirically to fit data very well (see below). This is so since σ is still estimated through the observed failures in eq. 6 and hence grows smaller together with the failure frequency.

It would now appear that the quality estimation in eq. 10 – despite corrected lower limit behavior - suffers from the same limitation as other theories. For zero detected

failures, eq. 10 returns an estimation of exactly zero due to the “variance compression” discussed above. The inability to prove a test suite successful is a trivial illusion, however, since a reasonably bad scenario is that the very next (un-executed) test in the suite would have caused a failure. Assuming that the *true* variance (i.e. what would be given by exhaustive testing) corresponds to this scenario we get a possible failure rate of:

$$\int_x F_S dx \approx |X| \left(2k \sqrt{\frac{1}{N(N+1)} - \frac{1}{N(N+1)^2}} \right) \leq \frac{2k}{N}$$

(11)

It should be pointed out at this stage that even if one assumes something along the lines of “the next two tests would have failed if we did not stop”, the quality bound still only increase by $\sqrt{2}$ due to the square root expression. Alternatively, the statistical certainty falls from 4 to slightly less than 3 standard deviations. In addition to this, one should keep in mind that statistics can only properly be used on large suites (which is not always practiced [6]), which in turn ensures highly modest deviations along the lines suggested above. Despite these objections, the simplistic approach used here is sufficient for many purposes and withstand the empirical evaluation quite well as we will see below.

A rightful question to ask now, is to what extent the results in eq. 9-11 contributes to random testing, apart from an apparent re-formulation of the theory for uniformly distributed inputs?

First of all it is a different approach based on a statistical evaluation instead of Nelson’s combinatorial approach and thus *not* a re-formulation. It also offers a great reduction of complexity as noted above – the software engineer can in eq. 10 and 11 choose the desired quality boundaries as in an ordinary engineering problem and get a solid estimation. Something practically put to the test in the empirical section of this paper with very good results. Furthermore does eq. 10 represent a definitive simplification – the example of 2,300 tests and 45 failures is far from easily assessed in eq. 2. In addition it offers both a simple “success theory” as well as the access to a vast set of existing advanced numerical methods for Monte Carlo evaluation of integrals [23]. The straightforward method presented here is the simplest possible, but makes a solid ground for further work, one being the possibility to introduce multiple user profiling based on an already conducted random test suite.

4. Empirical evaluation

No matter how powerful a theoretical result may be, it has to be empirically validated if it is going to be of practical use to engineers. This is true in all engineering disciplines, including software engineering. We therefore turn our attention to simulated testing, that is, the mimicking of real test situations where controlled

software errors are seeded. By looking at the “mutation coverage” (i.e. how many of the artificial errors that are found) it becomes possible to analyze the effectiveness of a test method, or as in this case, validate the theoretical quality estimations.

In order to use this approach for verification of the results in section 3 we have to undertake massive test simulations. Only extensive testing will give the “true” failure rates and variances. Differently put: performing 100,000 test *suites*, will the results of eq. 10 – especially the “upper quality bound” - hold to be true?

It should be pointed out that it is only by performing this kind of analysis, one really can say anything with relative certainty, about a statistically based theory. Once shown to be reliable on the other hand (which demands repeated validation beyond this article of course), the theories could readily be used in everyday software engineering in the same way as numerical integration and statistically based methods are used in other engineering disciplines daily.

4.1. Methodological framework

Using mutation analysis to estimate the absolute number of failures, failure rates etc., one should seed errors in software that is known to be defect-free and well known in the software community in order to establish a common base-line. This presents a problem since not even extremely reliable software can be said to be absolutely fault-free [1] and far from all software is “well known”. One way to solve this problem is to employ common, well-trusted software for mutation analysis. A couple of the more popular benchmarking software units are TRIANGLE [6,26,27] and FIND [6,28] which are programs to determine the type of a mathematical triangle and a highly straightforward sorting algorithm respectively.

A problem with “small and well controlled” pieces of software is that they have small bearing on real life software, however. Furthermore does a general criticism exist against random testing for performing well on small “toy programs”, but considerably worse on software used in real life, see e.g. [6]. The benefit of very small pieces of software on the other hand is the high degree of control in the experiment they allow. It becomes increasingly difficult to tune the failure frequency and behavior of the seeded mutations with the growing complexity of the code.

Trying to resolve this problem and meet the criticism in the field we have employed both a straightforward modulus error in order to allow “mass testing” as well as extensive material of more realistic cases. A number of well-known numerical routines from [23], some being more than 30 years old in their initial form, are used to host more realistic mutations/faults. The numerical routines at hand have also been continuously and extensively used in real life research and industry for a similar period of time. More extensively corrected, tested

and used software is hard to find, possibly short of embedded industrial systems.

Because the routines can be assumed to be defect free, they also provide testing Oracles in themselves. That is, we assume that the routines behave properly before mutation and replace the absolute failure frequency with the relative one where comparing the outcome from the original and the seeded code establish failure. Finally the code is available on CD, which eliminates the human factor in transferring the software to the platform to be used in the simulations (which sometimes renders a defect free software highly dysfunctional).

Turning to the simulations, we have chosen to seed artificial defects with variable failure rates, in each individual software. The combined failure rate is subsequently calculated theoretically or determined by massive testing (tens of millions of tests or more). In the second phase we perform a vast number of test suites for each software and fixed (true) failure rate. The statistically determined variance from the combined number of tests is calculated as well as the theoretical predictions for each *individual test suite*. A comparison is then made between the different theoretical predictions based on the minor test suites and the actual failure rates. Varying the failure rate, the full experiment is repeated for each rate and software.

4.2. Technical details

The random input was generated using the PRNG “Ran2” from [23]. All software subject to testing was written in Fortran and thus non-object oriented. The routines in [23] are readily available in C as well [29], but the Fortran version was chosen out of convenience. In addition to the numerical routines chosen, we implemented a simulated “modulus error” for massive testing, see fig. 4.

```

i=abs(Mod(testinteger, ErrorFrequency))
i=i+ErrorFrequency/2.0
k1=0

if (i.gt.ErrorFrequency) then
    k1=1
endif

k2=0

if (i.ge.ErrorFrequency) then
    k2=1
endif

```

Figure 4. The “modulus” software (upper) and its oracle (lower). Note the difference in the if-statements which will result in a failure (different result compared to the oracle) when $i = \text{ErrorFrequency}/2$.

This kind of fault, e.g. by mistake using “>” instead of “≥”, typically appears in type conversions, boundary controls, checksum routines or similar. Admittedly it is a type of fault that different kinds of coverage testing techniques normally would find. Still it is a type of defect that is readily found in modern software development [30]. It also holds the basic properties of most software errors, being exact (non-random), repeatable and a typical subject of human mistake. Furthermore it is a test with one of the shortest possible execution times, which allows massive testing.

To properly exploit the fast execution of the “mutation error” we performed test suites with the individual “batch size” of 10, 100 and 1,000 tests in each series. The cut-off at 1,000 tests was made out of practical considerations (see the total number of tests run below) but could in principle easily be raised to cover extremely long test series (millions of tests in each suite). Although the lower end with 10 tests hardly state a test series we included these short series for completion. For each batch size, we varied the error frequency (see fig. 4) through the values 1/10 down to 1/1500 in a 15 steps procedure. In each case, for each error frequency and batch size, 10 million suites were implemented, resulting in a total of 166.5 billion test executions. Hence we utilized the “modulus” error to severely challenge the theoretical results in terms of statistical validity.

Turning to the non-trivial software examples, we have chosen to investigate two real-life routines used in engineering disciplines over the years. This choice has been made in order to test the theoretical approach against not only a laboratory environment, but also realistic problems.

We employed the CISI and SVDCMP routines from [23], seeding well controlled mutations. The former routine calculates the cosine and sine integral for a given input value (float) and consists of 70 lines of code. The second routine is of a more complicated nature, larger (240 lines) and performs a singular decomposition of any given (mathematical) matrix. The input size of the matrix was chosen to 4 x 4, with all entries consisting of random float numbers.

In both cases we used two types of seeded defects, typical of human errors: parameters offsets and mutated if-statements (conditions shifted). In the latter case the code will end up executing the wrong part of the code, depending on the exact nature of the input. Changing a parameter value will only affect the outcome if it interferes destructively with the execution, e.g. if it is used as a multiplier or is large enough relative a key variable it is added to, in order to tip the balance in an if-statement.

Furthermore these two defect types represents two types normally found by different testing methods. While the “branch-defect” should be detected by a branch-coverage test, a parameter shift could prove intrinsically difficult to find using code coverage. Hence the two fault types not only represents common human mistakes, but

also pose a challenge for the current random approach due to the different nature of the two.

To further test the theory at hand we mixed both types of defects, resulting in a more realistic scenario with different levels and numbers of problems, see table 1.

As in the case of the “modulus error” (see fig. 4) we performed test series of 10, 100 and 1,000 tests in each. For every type of defect and suite size we performed 10 million test series, resulting in 111 billion tests of the two routines. At each stage the randomized input was fed to the un-permuted routine to create the “Oracle answer”, which subsequently was compared with the answer from the permuted routines.

Table 1. Schematic listing of the type of defects seeded in the two numerical routines used for empirical evaluation.

Type of error	
1.	Parameter error, rare execution
2.	Parameter error, executes more often
3.	Branch-defect, executes modestly often
4.	Combination of 1 and 3
5.	Combination of 2 and 3

In order to avoid recording “round-of errors” we controlled each manipulated software routine with the seeded defects “switched off” by running 100 million tests where the answers were compared in the same way as above. No failures were recorded, which strongly suggests the absence of round-of errors or similar. To establish absolute certainty with exhaustive testing was not a viable alternative due to the humongous size of the input space.

5. Empirical results and discussion

Statistical results, especially when it comes to quality estimations or similar, tend to be less “hands on” than “absolute numbers”. Neither are the implications always very intuitive or self-explanatory. The results of section 3 above are an attempt to solve some of the former ambiguity and the interpretation of our findings is very straightforward: the probability that the *true* failure frequency is within the predicted rate ($\langle F_s \rangle \pm k\sigma$) is given by a tabulated number. This on the other hand implies that if the software engineer uses e.g. two standard deviations in his/her quality estimation in eq. 10, the probability of the test suite at hand actually returning an estimation that does not match (within bars) the true failure frequency should be no more than 5%. If one is not satisfied with this (rather high) probability, one should use three or four standard deviations which would give 0.3% and less than 0.01% respectively (see section 3).

If the quality estimations in section 3 are to hold true, the portion of test suites that under- or overestimates the failure frequency may thus not exceed the given numerical limits. More strictly should the resulting distribution of

quality estimations comply with the assumed Gaussian approximation when it comes to variance and standard deviations in order to be useful.

Alternatively and more practically formulated, the extremely unlikely “1 in 10,000” (the case of $k=4$) test suite which provides an unusually bad sampling of the code at hand should - despite it being a “pathological” (bad) test suite - still result in a quality estimation that *actually encloses the true failure frequency*. That is, the upper quality boundary should coincide with the true failure frequency for the “1 in 10,000” ($k=4$) test suites. Only test suites being extremely rare are allowed to fail and actually underestimate the true failure frequency (see above).

5.1. Modulus evaluation

We now turn to the “modulus error” case described above where we compiled all performed test suites for the different batch sizes and failure frequencies (see section 4). For each fixed failure frequency we calculated the total average and variance employing all test cases in all available test suites. In all cases both the average and variance agreed with theory, confirming the theoretically tuned failure frequency.

Continuing with the quality estimations we have plotted the upper quality bound of the pathological test suites (1 in 10,000 $k=4$, 1 in 333, $k=3$ etc., see above) versus the true failure frequency for all investigated suites encompassing 1,000 tests in fig. 5.

An overall very good match is found, with the upper quality bound being very close to the true failure frequency. This directly implies that the quality estimation in eq. 10 (section 3) *does* hold the level of certainty introduced there, even when pushed to the limit in massive simulations.

As the number of observed failures goes down to zero, the estimation in eq. 11 (section 3) ensures a constant upper quality bound which properly encompass the true failure frequency. This is illustrated by the “leveling out” of the failure frequency estimations, turning into constant minimum value ($2k$, see eq. 10) “plateaus” in fig. 5. The upper “success” bound in this case is quite crude, however, even if fully functional as pointed out earlier.

The low frequency results is especially noteworthy since the data points before the minimum value plateaus sets in, corresponds to a single recorded failure in a test suite of 1,000 tests. What more is, this happens in highly un-representative test suites (where the “proper” number of failures should have been much higher), which represents the case of 1 in 10,000 (1 in 333 etc.) of all performed test suites. Still the upper bound of the quality estimation virtually coincides with the true failure frequency, something that clearly indicates the strength of the approach and the lower end adjustments in sec. 3.3.

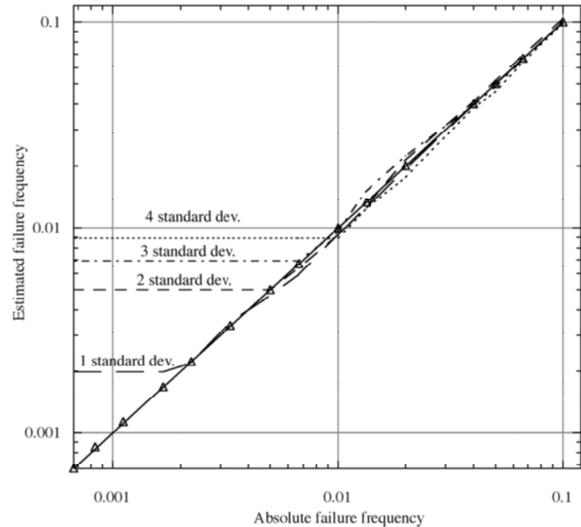


Figure 5. The upper boundary of estimated failure frequency in the case of “misfortunate” test suites (1,000 tests each) corresponding to 1, 2, 3 and 4 standard deviations (in e.g. the case of 4 deviations, the figure indicates the upper boundary value in the quality estimation made for the “1 in 10,000” poorest test suite). The triangles mark the total average found for each investigated true failure frequency (see the text).

In the case of the test suites with fewer tests (100 and 10 in each suite respectively), the agreement in the 100 case was just as good as in the case of more extensive (1,000) test suites. The minimal upper bound (the plateaus) sets in earlier though, due to the lower number of tests in each suite.

Also the “suites” of 10 test cases did conform to the theory presented in sections 3.2 and 3.3, but on the other hand does even a single failure imply an estimated upper quality bound of 0.8 failures per run (using $k=4$), which render most comparison useless. This effectively shows the absolute need for proper statistics using random testing.

5.2. CISI and SVDCMP evaluation

Turning to the evaluation of the tests of the mutated numerical routines we notice the same level of agreement. It should be noted though that in this case we do not have access to any theoretically calculated failure frequency but simply *define* the recorded average failure frequency as the “true” failure frequency. Now, performing the same kind of comparison between the upper quality boundary for the “pathological” (see above) test suites and the actual failure frequency, we find very good agreement between the theoretical results and the empirical evaluation (see fig. 6). The “plateau” phenomenon in the case of four standard

deviations (left in both panels) is recognized from the modulus case, but the over all agreement is of the same quality.

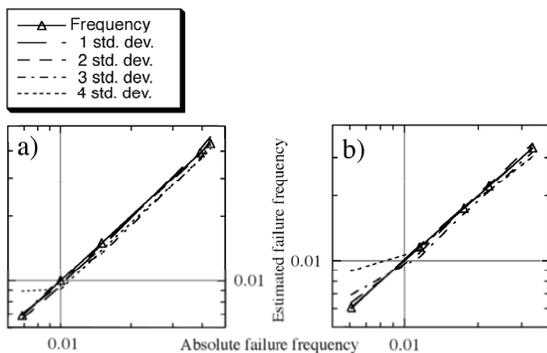


Figure 6. Match between the upper quality bound and actual failure frequency for “worst case” test suites. Triangles mark the failure frequencies for the different defects (see table 1). Panel a) is based on mutation of the CISI routine while panel b) is evaluated using SVDCMP. The scale is the same as in fig. 5.

The failure frequency is confined to a smaller interval as compared to the “modulus error”, but contains on the other hand only five defects in each case due to the considerably longer execution time. Still, there is no difference in agreement between the simple modulus failure and the mutated numerical routines. Nor did we find any differences between agreement for the branch, parameter or mixed defects for the two routines.

Perhaps even more interesting is that the accuracy of the presented method actually is rather independent of the complexity of the input as long as the statistics is good enough for the integral evaluation process underlying our results in sec. 3.2 and 3.3. Otherwise this is a much-debated issue. D. Hamlet has in several papers, see e.g. [14], compared the effectiveness of random testing with investigating the fauna of the great lakes by putting in one day’s effort of commercial fishing. From this follows the apparent logical conclusion; “the greater the lake, the poorer a method”.

In our current formulation, however, it becomes evident that the input to the failure function F_S is of less interest since we are sampling the failure profile (the integral in eq. 3 and 4, see also fig. 3). A complex input *does* complicate the use of a more realistic user profile than the uniform distribution used here though, but only in the input generation process itself.

Finally we note that the test suites with fewer tests conform well to the theoretical results in eq. 10 (section 3), but generally the minimum value plateaus sets in so early that most failure frequencies are covered by them in the same way as with the “modulus error”.

6. Conclusions

In this article we have presented a short expose of arguments against and in favor of random testing. Although outperformed per test, random testing may still perform better per time unit than coverage testing and offers a set of mathematical tools that is otherwise lacking in much testing, especially considering early phases of testing. The drawbacks have so far been evident despite this: ambiguous evaluations and no unique benefit over other approaches.

We have shown, however, that it is possible to transform the issue of random testing into an equivalent problem, formulated using integrals. Applying Monte Carlo evaluations techniques in conjunction with the central limit theorem it becomes possible to evaluate the software reliability using a given test suite in a very precise manner. Upper bounds on the true failure frequency (as given by exhaustive testing) are easily established within this framework. Furthermore our result allows us to establish an upper bound on the quality of a test suite returning *zero* failures, i.e. it provides a limited but working “success theory”.

Massive empirical evaluations with simulated testing scenarios encompassing hundreds of billions of tests have been used to verify the current theory successfully not only for small artificial failures but also seeded failures in two extensively used routines. Taken together this present a relative advancement of the state of random testing and ads a special feature in the case of the success theory. This clearly indicates that there is still room for random testing in today’s testing practice.

7. Future work

The findings presented here opens up a variety of possibilities, most notable in the area of user profiles. While previously existing theories in random testing are not easily adapted to encompass a flexible user profile (i.e. one have to specify the user profile before running the test suite), the integral formulation presented here offers new possibilities to solve this long standing problem in random testing. Furthermore, the results in section 3, leaves much to be improved on, being relatively coarse in their estimations. Finally is the connection to, and combination with, coverage testing of especial interest.

8. Acknowledgements

The European regional fund and the KK-foundation for financing parts of this project

9. References

- [1] L. Hatton, “N-version design versus one good design”, IEEE Software, pp. 71-76 (Nov./Dec. 1997).

- [2] J. Miller, M. Roper, M. Wood and A. Brooks, "Towards a benchmarking for the evaluation of software testing techniques", *Information and Software Technology*, **37**, pp. 5-13 (1997).
- [3] M. Rooper, "Software testing - searching for the missing link", *Information and Software Technology* 41, pp. 991-994 (1999).
- [4] P. G. Frankl, E. J. Weyuker, "Testing software to detect and reduce risk", *J. of. Systems and Software* 53, pp. 275-286 (2000).
- [5] T. W. Williams, M. R. Mercer, J. P. Mucha, R. Kapur, "Code coverage, what does it mean in terms of quality?", *IEEE Proc. Reliability and Maintainability* 2001.
- [6] H. Yin, Z. Lebne-Dengel and Y. K. Malaiya, "Automatic test generation using checkpoint encoding and antirandom testing", Technical report CS-97-116, Colorado State University
- [7] Y. K. Malaiya, "Anti-random testing: getting the most out of black-box testing", *Proc. International Symposium On Software Reliability Engineering*, pp. 86-95 (Oct. 1995).
- [8] S. Kuball, G. Hughes, "Scenario-based unit-testing for reliability", *IEEE Proc. Reliability and Maintainability* 2002.
- [9] N. Kobayashi, T. Tsuchiya, T. Kikuno, "Non-specification-based approaches to logic testing for software", *Information and Software Technology* 44, pp. 113-121 (2002).
- [10] D. C. Ince, S. Hekmatpour, "Empirical evaluation of random testing", *Comput. J.* 29, p. 380 (1986).
- [11] J. W. Duran, S. C. Ntphasos, "An evaluation of random testing", *IEEE Trans. Software Engineering* 10, pp. 438-444 (1984).
- [12] Y. K. Malaiya, J. Denton, "Estimating Defect Density Using Test Coverage", Technical report CS-98-104, Colorado State University.
- [13] B. Korel, "Automated test data generation for programs with procedures", *Proc. 1996 International Symposium on Software Testing and Analysis*, pp. 209-215, ACM Press.
- [14] D. Hamlet, "Foundations of software testing: dependability theory", *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, New Orleans, Louisiana, United States (1994).
- [15] D. Hamlet, "Random testing", *Encyclopedia of Software Engineering*, editor J. J. Marciniak, John Wiley & Sons (2001).
- [16] C. Wohlin and P. Runeson, "Defect content estimations from review data", 20th international conference on software engineering, (Kyoto, Japan), pp. 400-409, IEEE Computer Society (1998).
- [17] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons (1979)
- [18] G. McGraw, C. Michael, M. Schartz, "Generating software test data by evolution", RST Corporation, technical report RSTR-018-97-01 (1998).
- [19] C. Michael, G. McGraw, M. Schatz, C. Walton, "Genetic algorithms for dynamic test data generation", RST Corporation, technical report RSTR-003-97-11 (1997).
- [20] J. D. Musa, A. Iannino, K. Okumoto, "Software reliability, measurement, prediction, application", McGraw-Hill, New York (1987).
- [21] N. Li, K. Malaiya, "On input profile selection for software testing", Technical report CS-94-109, Colorado State University.
- [22] R. Thayer, M. Lipow and E. Nelson, *Software Reliability*, North-Holland (1978).
- [23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Recipes in Fortran", Cambridge Press, Cambridge (1999).
- [24] A. I. Khincin, "Mathematical foundations of statistical mechanics", p. 166, Dover publications, New York (1949).
- [25] L. Råde, B. Westergren, "Beta - Mathematics handbook", Studentlitteratur, Lund (1993).
- [26] R. A. Demillo, R. J. Lipton, F. G. Sayward, "Hints on data selection: Help for the practicing programmer", *IEEE Computer*, pp. 34-41 (1978)
- [27] P. C. Jorgensen, "Software testing: A craftsman's approach", CRC Press, New York (1995).
- [28] W. E. Wong, "On mutation and dataflow", Ph.D. thesis, Purdue University, Computer Science department (1993).
- [29] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Recipes in C", Cambridge Press, Cambridge (1999).
- [30] R. Torkar, S. Mankefors, K. Hansson, A. Jonsson, "An exploratory study of component reliability using unit testing", *ISSRE 2003* (accepted).