2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Prediction of Undetected Faults in Safety-Critical Software

Johan Sundell*, Richard Torkar†, Kristina Lundqvist*, and Håkan Forsberg*
*School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
Email: johan.sundell@mdh.se
†Computer Science and Engineering
Chalmers and the University of Gothenburg
Gothenburg, Sweden

***Abstract*—Safety-critical software systems need to meet exceptionally strict standards in terms of dependability. Best practice to achieve this is to follow and develop the software according to domain specific standards. These standards give guidelines on development and testing activities. The challenge is that even if you follow the steps of the appropriate standard you have no quantification of the amount of faults potentially still lingering in the system. This paper presents a way to statistically estimate the amount of undetected faults, based on test results.**

***Index Terms*—software, safety-critical, test, fault prediction.**

I. Introduction

More and more decisions and responsibilities are being handled by increasingly powerful computer based systems that need to be extremely reliable, this could for example be an electronic flight control system. One problem is that it is difficult to predict the failure rate of software-based systems prior to release [1]. This is of special concern when it comes to safety-critical systems, where a failure may result in severe consequences, e.g., loss of life or environmental damage. The required target failure rates of such systems are typically so low that they can not be proven [1, 2]. For example, a fly-by-wire flight control system in an aircraft is commonly required to have a failure rate of less than 10^{-9}/flight hour [3], i.e., less than 1 failure in 100,000 years, which in a complex system is hard or impossible to prove both in practice and theory.

Currently, the best practice, and the only acceptable way to claim such level of reliability is to adhere to a development standard. Provided that all activities and steps, as defined by the standard, are followed, it is assumed that the target level failure rate is reached. Examples of such standards are DO-178 (Software Considerations in Airborne Systems and Equipment Certification) [3] used in the aviation industry, and the main standard for functional safety IEC 61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems) [4]. IEC 61508 is considered an ”umbrella” and has been applied by safety-critical industries. ISO 26262 (Road vehicles—Functional safety) [5], and IEC 61513 (Nuclear power plants—Instrumentation and control important to safety—General requirements for systems) [6] can both be seen as extensions of IEC 61508.

Such standards, as the above, define a rigorous process for the development of safety-critical systems. A number of aspects are covered such as, necessary documentation, the organization of test teams, and required test coverage. However, by following the standards one has not proven that the system has actually reached the target level of failure rate. The gap between the claimed and the proven failure rate is sometimes referred to as the ‘prediction gap’ [1]. Closing this gap is difficult as exhaustive testing is almost never an option.

This paper presents a way to use results from random testing to statistically predict the total amount of both detected and undetected faults that remain in the software. The main idea is to execute tests with more than 1,000 random samples. The actual number of samples is an important input to the prediction formula. The analysis of the test result will give the total number of failures that have been triggered by the test cases. As this concerns safety-critical software all failures have to be analyzed to determine their causes (i.e., conducting root cause analysis). The root cause analysis will provide information about how many faults in the software have caused the failures that were encountered. This data, i.e., the number of test cases, number of encountered failures, and number of unique causes (faults), is sufficient to calculate an estimate of the sum of the undetected faults. Moreover, all this data are available from the test process. In the end, such an estimate can then be used as a quality metric of the software that provides input to decision-making during the development process, e.g., ready for deployment or not.

This paper is organized as follows. Next we introduce related work. This is followed by the assumptions we apply to our approach, and the rationale behind those assumptions. Thereafter (Sect IV), we present concepts and definitions that have been used. The results are summarized in section (Sect. V which is followed by an analysis and discussion (Sects. VI–VII). The paper ends with some concluding remarks.

II. Related work

Research specifically focused on testing of safety-critical software is limited and often proves its inadequateness and infeasibility. The verification of the most strict failure rate

978-1-7281-0888-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSTW.2019.00069

requirements was early on identified as very challenging, if not a futile task [2, 7–9].

Parnas et al. [2] states "Because of the large number of states (and the lack of regularity in its structure), the number of states that would have to be tested to assure that software is correct is preposterous."

The stringent requirements on reliability and failure rate for safety-critical software leads to a 'prediction gap' [5] between what is required (and subsequently claimed by the industry) and what actually can be proven during testing. Moreover, the strict requirements inevitably make many accepted methods unfeasible to apply [7–9]. Reliability growth models, size and complexity metrics etc., cannot be relied upon in this context, as they do not provide sufficient accuracy or would require too much time and resources to be of use [8, 10].

However, if testing reveals no failures the reliability can still be estimated. One example is Voas et al. [11] work, that describe methods for estimating reliability, even if the operational profile differs from the input distribution used during testing.

Concerning random testing Arcuri et al. [12] define tight lower bounds for the expected number of test cases sampled by random testing to cover predefined targets. The development standards typically mandate different coverage criteria, e.g., MC/DC, statement or branch coverage. The more critical the software is the stricter the requirements. However, the effect of such criteria on fault detection is debated [13, 14]. There are indications that there is no strong association between such criteria and effectiveness in terms of fault detection. In short, this could indicate that it is the additional testing in itself that detects more faults and subsequently results in a more reliable software, indicating that there exists a mathematical relationship between the amount of undetected faults and the number of executed test cases.

Another interesting area that is related to the work presented in this paper is combinatorial testing [15]. Aspects of this field help to explain some of the dynamics of fault detection in relation to test coverage and number of parameters. Additionally, it gives ideas on how to quantify the required variation of the test cases (this will be further elaborated on in Sect. VII).

The work presented in this paper is disjoint from and complimentary to above works. It does not require all faults to be detected and is in fact only applicable when failures have been encountered. The principle of the suggested method is to take a 'sufficiently' large number random test cases and calculate the corresponding failure densities. The method is intended to be applied on software versions throughout the software development process. The ultimate aim of this paper is to present a way to quantify and predict what remains undetected based on the results from testing, i.e., we want to predict both the current amount of faults and what remains undetected.

III. Assumptions and Rationale

The below assumptions concern development of safety-critical systems in accordance with standards, e.g., ISO 26262 and DO-178.

A. Assumptions

For this work the following assumptions are made:

1) The input and state space of the system is reachable.
2) All parameters are defined at runtime.
3) The system has a deterministic behavior.
4) Uniform sampling of the input space is used.
5) More than 1,000 test cases are generated.
6) More than 50 faults are present in the system prior to testing.

These assumptions may not hold for software systems in general and may require further elaboration. Therefore, their rationale, are further discussed below.

B. Rationale

For many practitioners in the software development field some of the assumptions might appear unrealistic. However, safety-critical software is different compared to other software in many aspects.

1) The input and state space of the system is reachable. The first assumption is the reachability (of all states). A key characteristic is to make the safety-critical software 'verifiable', meaning that it can be checked for correctness by a person or a tool. Moreover, there are requirements on removal of 'dead code' and 'deactivated code'. If the code cannot be reached during testing it should be removed. The standard DO-178C (the current DO-178 version), which is widely used in the aviation industry, uses the following definitions:
   a) Dead code. Executable Object Code (or data) which exists as a result of a software development error but cannot be executed (code) or used (data) in any operational configuration of the target computer environment. It is not traceable to a system or software requirement.
   b) Deactivated code. Executable Object Code (or data) that is traceable to a requirement and, by design, is either (i) not intended to be executed (code) or used (data), for example, a part of a previously developed software component such as unused legacy code, unused library functions, or future growth code; or (ii) is only executed (code) or used (data) in certain configurations of the target computer environment for example, code that is enabled by a hardware pin selection or software programmed options.

   Other standards applicable to safety-critical software development take a similar stance, e.g., ISO 26262 (Road vehicles—Functional safety) requires a rationale for why any dead code is acceptable. The main idea is that unreachable code, executable or not, should be removed.
2) All variables are defined at runtime. The initialization of variables when starting the system is required in order to guarantee determinism, see rationale 3 below.

3) The system has a deterministic behavior. As a rule, safety-critical software need to be deterministic, i.e., it should always behave the same way under the same circumstances. The same input must always give the same output. This is the basis for why a successful test can be used as an argument for the safety-critical aspects of the software. The deterministic behavior of the safety-critical software is strengthened by the use of so called defensive programming. This refers to techniques and practices that may be used to prevent code from executing unintended or unpredictable operations by constraining the use of structures, constructs, and practices that have the potential to introduce failures in operation and faults in the code.
4) Uniform sampling of the input space is used. The full range of all input parameters is sampled with the same probability. No consideration is given to what is seen as more likely in a real operational environment. It is important to include out-of-range values as well as other deviations like timing delays. Sensor values and other inputs are typically checked for validity before they can be used. Sampling the entire input space leads to sampling of the entire state space of the system (by Assumptions 2–3).
   It is not a question of exhaustively cover the input space but to allow a sampling of it. The uniform distribution gives predictability and a possibility to quantify the process of fault detection. Any other sampling distribution than uniform raises questions about validity. For example, if a certain operational input profile has been identified and is used during testing then this profile might change if the system is used under other environmental conditions, or if other operators use it. A real life example could be an aircraft that is deployed overseas and used in different conditions than what was originally intended.
   Note, that if only normal range values are chosen as input the estimate does not consider the (often substantial) parts of the code that handle out-of-range-values.
5) More than 1,000 test cases are generated. The focus and aim of this paper is to quantify the fault finding and failure triggering of longer test series, typically randomly generated. In this context 1,000 is a relatively modest number. The assumption of 1,000 test cases or more is a somewhat crude way of avoiding (or postponing) the question of the actual number of tests that are required to calculate a reliable estimate.
6) More than 50 faults are present in the system prior to testing. According to a study funded by the UK MoD of the transport aircraft C130J software, it was determined to have 1.4 safety-critical faults per 1,000 lines of code [16]. McDermid and Kelly [1] states that "There is a general consensus in some areas of the safety-critical systems community that a fault density of about 1 per KLoC is world class."

IV. Predicting residual undetected faults—Concepts and definitions

During the development of safety-critical software all faults that are found during testing need to be removed (or dealt with by operational limitations). For good reasons this type of software cannot be delivered with a 'known bugs list'. However, removing all detected faults does not mean the software is fault free. The danger comes from the undetected faults that remain in the system. The concept of prediction of what is undetected may require some further elaboration on the concepts and definitions.

A. Failure Density versus Failure Rate

In this paper, the *failure density* is defined as the relative part of the input space that cause failures. Or put in a more formal way, the sum of all partitions that trigger a failure at runtime, as a percentage of the entire input space. This is a strictly combinatorial approach and has nothing to do with how likely a given combination of inputs would be in operation.

A one parameter example can be a Boolean input that results in a failure of the system whenever it is `FALSE`. This would give a *failure density* of 0.5. Three input parameters, integers a, b, and c, each with a range of $[0, \ldots, 9]$, that trigger a failure if they all have the value 4, would add 0.001 to the overall *failure density*.

Unless the operational profile is equal to a random (uniform) sampling of the input space, the failure rate $\neq$ failure density. However, the following conclusions can still be drawn; a *failure density* of 0 results in no failures and consequently 1 would mean that the system always fails. Moreover, a lower *failure density* would, on average, mean a lower failure rate.

B. Distribution of Fault Probabilities

A software system is assumed to enter the testing phase with a number of faults (Assumption 6). Each individual fault is triggered by a combination of different input parameters. According to a study, 97–98% of all software failures are caused by three parameters or less. No failures are caused by more than six parameters [15].

There is a certain range of each parameter that contribute to the triggering of the fault. When these ranges are added up, the fault can be seen to constitute a partition of the input space. Each fault in the system has its own unique partition. [1]

The simulation model used in this analysis corresponds with what Littlewood and Strigini [8] propose regarding a conceptual model of the software failure process:

> Different faults contribute differently to the overall unreliability of the program: some are 'larger' than others, i.e., they would show themselves (if not removed) at a larger rate. Thus different faults have different rates of occurrence.

This concept of 'size' is used in this paper, where large means a large partition of the input space; hence, occurring

[1] Any overlap can only be counted once as the risk of encountering a fault is not increased if more than one failure is triggered, i.e., if all faults overlap the risk of triggering any fault is significantly reduced.

at a relatively large rate during random sampling. Below is a description of how the model works in the form of the classical urn problem.

C. Urn Model

In statistics, problems are often described as urn models, where balls of different colors are drawn from an urn. In this case, the way the model works can be described as follows.

Assume an urn with x white and y black marbles. Furthermore, assume that the black marbles can be bundled together, i.e., they express some attribute of stickiness, but that single black marbles can also be drawn. Now, draw a single marble from the urn and increment τ. If a black marble, or a cluster of black marbles, is drawn from the urn, color the marble (or each marble in the entire cluster) green. Increment υ and ϵ. If a green marble/cluster of marbles is drawn increment only ϵ. After each draw, put the marble(s) back into the urn.

Since $\tau \gg 1$, the ratio ϵ/τ is a good estimate of the proportion of black marbles in the urn [17]. Additionally, the relationship between υ and ϵ provides some information about the initial aggregation (if any) of black marbles. We assume that if υ/ϵ is small, say < 0.05, then we have drawn the same (large) clusters over and over. On the other hand if υ/ϵ is large (> 0.95) then the black marbles are not clustered together.

For example, assume $\tau \gg 1$ and the scenario where $\upsilon/\epsilon < 0.05$, e.g., we have drawn the same cluster a 100 times, the first time as black and the rest as green: $\upsilon = 1, \epsilon = 100, \upsilon/\epsilon = 0.01$. For the other scenario, $\upsilon/\epsilon > 0.95$, we have drawn 100 black marbles, one after another: $\upsilon = 100, \epsilon = 100, \upsilon/\epsilon = 1$. In the latter case, we have a more dispersed distribution, which is not dominated by one or a few large bundles. In the former case, ($\upsilon/\epsilon < 0.05$), the fault distribution is dominated by one or a few large bundles.

Ultimately, the above indicates that we have a multinomial logistic regression, i.e., we have > 2 outcomes (white, black, and green). If there are K types of events with probabilities $p_1, \ldots, p_K$, then the probability of observing $y_1, \ldots, y_K$ events of each type out of n trials is:

$$\Pr(y_1, \ldots, y_K | n, p_1, \ldots, p_K) = \frac{n!}{\prod_i y_i!} \prod_{i=1}^{K} p_i^{y_i}$$

However, designing and interpreting multinomial logistic regression models is hard, so in order to simplify things we disregard 'white' marbles. After all, the two entities of main interest are the black and green marbles, and the ratio υ/ϵ. This makes it more straightforward, i.e., $y \sim \text{Binomial}(n, p)$.

D. Proposed Approach

During a typical test phase in industry, tests are performed and test data are gathered. This data holds information regarding encountered failures and the number of tests performed. Identified failures are typically analyzed, classified, and the triggering faults removed. By counting the number of failures detected and by classifying the causing faults as previously found or not, it is possible to estimate the total sum of the remaining faults.

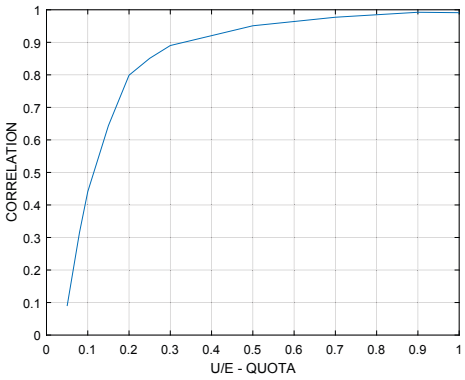

Figure 1. Correlation vs. υ/ϵ. The plot is cumulative, i.e., 0.5 on the x-axis includes the correlation for all simulations with a υ/ϵ ratio less than 0.5.

Let us define the number of tests as τ, detected failures as ϵ, and the number of uniquely identified faults as υ. The undetected failure density, ρ, is then defined as:

$$\rho = \lambda \times e^{-\lambda} \tag{1}$$

where $\lambda = \upsilon^2/(\epsilon \times \tau)$.

The rate parameter of the exponential function, λ, is reliant on the factors υ/ϵ and υ/τ. Where a high value of the first term indicates that many (in numbers) faults remain undetected.

The υ/ϵ ratio holds information about the size distribution of the faults. For a given failure density, a υ/ϵ value close to 1 indicates a granular size distribution, i.e., many small faults. A value close to 0 means that one or a few larger faults dominate in terms of size, thus triggering failures more often.

However, the υ/ϵ ratio will gradually decrease as the number of tests increase (the more faults that are found the fewer remain to detect). This means that the faults already found are encountered over and over again, i.e., $\upsilon/\epsilon \to 0$. The prediction formula renders reliable results when $\upsilon/\epsilon > 0.15$. Concerning our second ratio, $\upsilon/\tau \to 0$, decreases at an even faster rate as τ increases.

E. Method

In order to validate our approach we conducted a simulation study. The model built for this purpose needed to capture the process of random generation of test cases and validating if each of these test cases triggered failures or not. Moreover, the model also needed to allow for flexible generation of faults in the system, both in terms of fault sizes and number of faults.

Intuitively, it can be understood that larger faults are easier to find than small ones. If a given failure density, is either divided in to a few large faults or many smaller, the latter requires more test cases to detect. The factors that affect the outcome of the test process are the number of test cases and the distribution of fault sizes. The model must allow for this dynamic to be explored.

A model that corresponds to above description was implemented in MATLAB versions R2016b and 2017a. The

model generates a random distribution of independent faults. Thereafter, a given number of random test cases are generated. Each test case is checked to see if it triggers any of the faults. The individual size of each fault is randomly generated. However, the number of faults and from which distribution its size is drawn from, can vary.

The algorithm can be described as follows:

Data: input
Number of test cases (τ): this is the number of randomly generated test cases in the simulation.
The total sum of the initial faults prior to testing, in percent. Example, if 1% is chosen as input it means that 1% of the combinations of the input space trigger a fault.
Distribution parameters that specifies the granularity and distribution of the different fault sizes. The parameters differ depending on which distribution is used, e.g. $\mathcal{N}(\mu, \sigma^2)$, Beta$(\alpha, \beta)$. Additionally, in some cases the actual number of faults are specified.
while *generated test cases* ¡ τ **do**
  generate test case;
  if *fault triggered* **then**
    increment ϵ;
    if *a new fault found* **then**
      increment υ;
    end
  end
end
Result: Calculate an estimate of ρ
Algorithm 1: Algorithmic description of a simulation

The simulation model keeps track of all faults and knows if they are triggered or not. Based (only) on the outputs; number of test cases (τ), number of test cases that triggered a fault (ϵ) and number of detected faults (υ), an estimate of the sum of the remaining faults is calculated. This calculated estimate is then compared with the generated 'true' value from the simulation. This step represents the analysis of a test series, including the root cause analysis which gives υ.

The above model was used to perform a number of simulations with different number of test cases, total initial amount (size) of and number of faults. The results from the simulations were used to build a prediction model that gives an estimate of the sum of the undetected faults using only parameters τ, ϵ, and υ.

F. Distribution of Fault Sizes

Software faults are caused by many different sources. They are, by nature, unintentional and typically stems from a lack of understanding of the interaction between the code and the surrounding system, e.g., concerning requirements (incomplete, ambiguous, incorrect), logic (high complexity can lead to a lack of transparency), interfaces (timing, resolution, accuracy, sequencing), design (misinterpretation of requirements, erroneous algorithms), and implementation (divide by zero, parameter initialization, memory leaks, compilable typos, functional differences between release and debug mode).

Table I
RESULTS—GAUSSIAN FAULT SIZE. FROM LEFT TO RIGHT, NUMBER OF SIMULATIONS, NUMBER OF TEST CASES, CORRELATION COEFFICIENT, AND MEAN/VARIANCE.

Simulations	Num. test cases	Corr.	$\bar{x}/s^2$
10024	$1000 \rightarrow 100.000$	0.99	-0.28×10^{-3} /0.025×10^{-3}
102	$1000 \rightarrow 1000.000$	0.99	-0.19×10^{-3} /0.051×10^{-4}

During the development process, the code has undergone rigorous scrutiny and has been analyzed and reviewed by many experts and/or tools in accordance with the applicable standard. These practices will find and eliminate the most obvious problems. The faults that reach all the way to the test environment are thus limited in terms of size (partition of input space). The larger, easy to find faults have been removed early on. What remains are a number of independent faults, limited in size and brought on by a number of causes. The distribution of the fault sizes is not known. No empirical data, expressed in terms of fault partition size, is available. However, in this case of ignorance, a Gaussian distribution is a likely candidate. The limitation of fault size will ensure a finite variance. Moreover, the Central Limit Theorem states that sufficiently many samples will tend toward a Gaussian distribution. In this case we are considering 50 or more faults (Assumption 6).

V. RESULTS

10,024 independent simulations were performed where the fault sizes were sampled from a normal distribution. The following setup was used for the simulations:

- $0\% <$ initial failure density $\leq 2\%$
- 1,000 $\leq$ number of test cases $\leq$ 100,000
- 50 $\leq$ number of undetected faults from start $\leq$ 10,000

After each simulation the result, actual sum of undetected faults, and the estimate, of the same, were compared (see Fig. 3). The comparison shows a strong overall correlation (see Fig. 2 and Table I). In cases where the ratio $\frac{\upsilon}{\epsilon} < 0.15$ the correlation drops (see Fig. 1). However, the absolute value of the prediction error is typically low. On average, the prediction formula underestimates the result with approximately 10% (see Fig. 4 and Table I).

The blue and red areas combined, in Fig. 3, represent the sum of all faults prior to testing. The red area is what the random test cases have triggered and, thus, identified. The blue area is the remaining undetected faults after the test. This is ultimately what is being predicted.

VI. ANALYSIS

The aim of the prediction formula is to estimate the failure density of the software. The expected value is normally distributed approximately centered around the actual value.

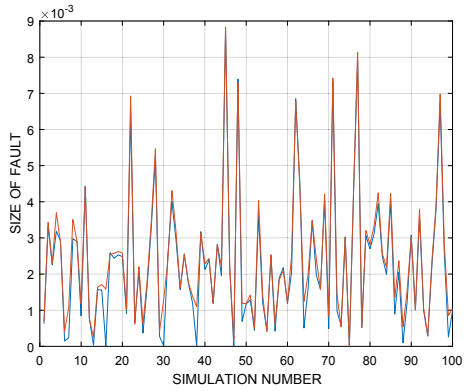

Figure 2. Estimate and result. The blue line shows the estimate and the red line shows the result for 100 independent simulations. The y-axis shows the sum of the size of the undetected faults. The x-axis shows the number of each simulation.

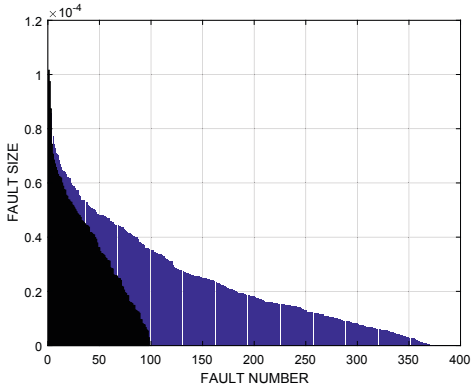

Figure 3. Size sorted normally distributed faults. The y-axis shows the fault size. The x-axis shows the number of each fault. The smaller area to the left is the detected faults and the blue area the undetected.

We believe it provides an estimate accurate enough to form an opinion of the quality of the software.

One way to analyze and display how well the estimates match the results is to draw a histogram of the differences between the two. In Fig. 4 we see that we have a slight underestimation.

Another way of analyzing the results is by constructing an ROC curve (receiver operating characteristic). First we create a confusion matrix by examining if the estimate is within or outside a given tolerance, see Table II. In this case, there are no True Negatives nor False Positives, which gives a recall of 1. The precision is calculated as True Positives divided by the sum of True Positives and False Negatives. Furthermore, the threshold is here chosen to be either relative (as a percentage of the result) or absolute (the overall standard deviation), see Figs. 5–6. The AUC values (area under the curve) are 0.70 and 0.91 respectively.

A closer look, at the data from all performed simulations, reveals that the correlation and the relative accuracy of the

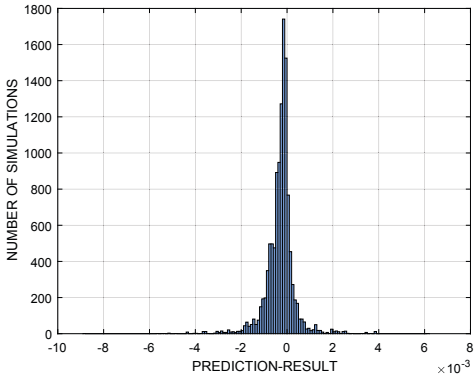

Figure 4. Histogram of the difference in estimation. The offset from zero indicates a slight underestimation of the result.

Table II
CONFUSION MATRIX

		estimate outcome p	estimate outcome n	total
actual value	p′	Estimates inside	Estimates outside	P′
	n′	Always 0	Always 0	N′
total		P	N	

estimate is lower as the values approach zero. However, as can be expected, the absolute difference is lower as well.

A. An Example

Assume that an industry project, that develops a safety-critical software component, decides to perform 5,000 random tests to reinforce the test process and to be able to predict the failure density of the software. The result is that 46 failures are encountered. The subsequent analysis reveals that the failures are caused by 38 different faults in the software. The faults are carefully removed in accordance with the stringent procedures in place (ensuring that no new faults are introduced).

Using the following data: number of tests = 5000, number of failures encountered = 46, number of unique faults detected = 38, and remaining (undetected) failure density = ρ, we first calculate λ:

$$\lambda = 38^2/(46 \times 5000) \tag{2}$$

and then estimate ρ:

$$\rho = \lambda \times e^{-\lambda} \approx 0.0063 \tag{3}$$

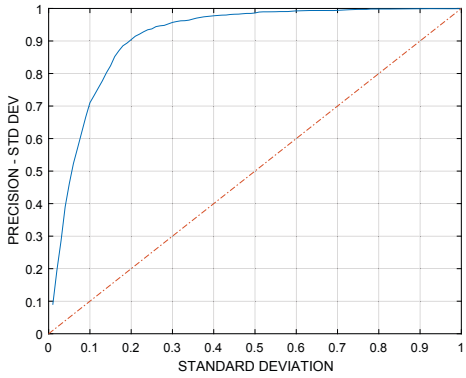

Figure 5. ROC curve of estimate vs. result. Here the threshold for an acceptable deviation is set as part of the overall standard deviation.

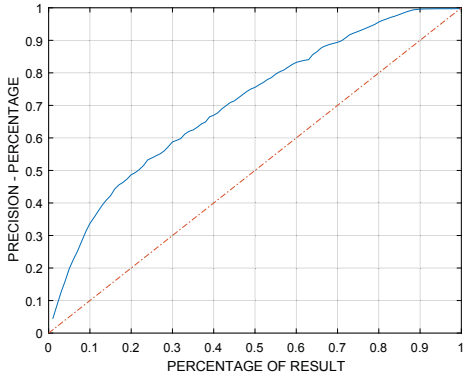

Figure 6. ROC curve of estimate vs. result. Here the threshold for an acceptable deviation is set as a percentage of the result.

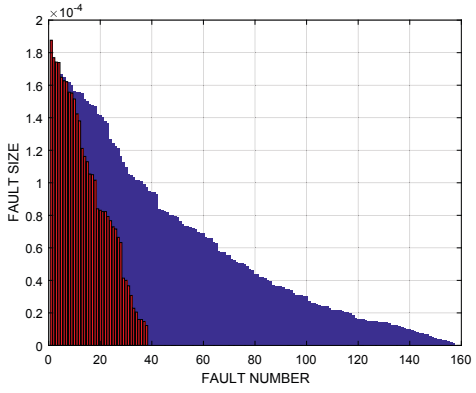

Figure 7. Our example case. 5,000 test cases have detected 38 faults (red area). 120 faults remain undetected (blue area). The blue area is predicted to be 0.00628 but is in reality 0.00638.

To summarize, after the removal of the 38 identified faults, it is predicted that the remaining failure density is approximately 0.0063, i.e., execution of 10,000 random samples of the input vector would result in an estimated 63 failures.

VII. DISCUSSION

Exhaustive testing is not a viable option for larger software systems. However, it is possible to make predictions, about the full state space, from the results of many test cases. The Central Limit Theorem guarantees that, as the number of test cases is increased the quota ϵ/τ will converge towards the failure density of the system. If information about υ is considered it is also possible to calculate the amount of undetected faults. This is possible as both the ratios υ/ϵ and υ/τ are reliable from the large number of test cases. These undetected, lingering faults constitute the most interesting part, as they are potential causes of failures when the software is deployed. The encountered faults pose less of a threat as they obviously can be removed.

If a large enough number of test cases are executed (in the way described), the size of the undetected faults can be estimated. The resulting τ, ϵ and υ corresponds to one specific undetected failure density ρ. Though theoretically possible, in a large state space it is highly unlikely that the executed test cases have revealed all faults. Even thousands of tests only represent a small sample of the full state space. Or vice versa if the detected faults would be the only faults present in the system it is very improbable that you would get those ϵ and υ for that τ.

The more test cases that are executed the fewer faults can remain undetected and the better the estimate. However, from a sufficiently long test series you can early on calculate the current observed and undetected failure density of the system and determine if it has reached an acceptable level.

The actual operational profile of a system is not always known, it is not always static, and it might change over time, or due to changes in the surrounding environment. Different operators might use different operational profiles, e.g., high-gain vs. low-gain pilots who fly the same aircraft differently. This means that it is difficult to draw conclusions of in-service-failure-rate from testing. However, a random (uniform) sampling avoids this asperity by looking at the entire input space. The results are not related to an assumed operational profile. Instead the overall failure density is an indication of the software quality, this aspect is important, especially when considering safety-critical software. A major advantage with this approach is its predictability.

The failure density of the entire system correlates with the failure densities of any subspaces that are used more frequently during operation. Regardless of which operational profile is used, sampling from a system with a low failure density will, on average, result in fewer failures during operation, compared to a system with a high failure density.

The estimate is calculated from known parameters τ, ϵ, and υ. At runtime only τ and ϵ are known but determining υ typically requires some degree of analysis, in order to

understand which part of the code that cause the failure. The number of test cases until a new fault is found (υ is incremented) increases as more faults are encountered and fewer remain to be detected. If these numbers were known the variance of the estimate could be reduced. However, here the aim is to find a pragmatic estimate that can be directly applied in industry.

The presented results are based on that the partition sizes of the faults are normally distributed. However, the prediction formula still gives acceptable results as long as the υ/ϵ ratio is above 0.15. Several thousands of simulations have been performed using other distributions, e.g., $\mathcal{B}, \gamma, \mathcal{U}$, etc. The best results are obtained by assuming uniform fault partition sizes.

The simulation model generates faults that can overlap. This contributes to the underestimation of the prediction model.

VIII. THREATS TO VALIDITY

The model used in this study describes an ideal system. It is assumed that the entire input space and state space of the system is completely reachable, any failures that occur are detected, there is no dead code, etc. The reality is somewhat different.

For example, control loops or code that is highly nested and sequential, and thus is dependent on previous input, can be very hard to reach. Great care has to be put into the generation of input to make sure all combinations are as feasible. Achieving a random (uniform) sampling of the input space may require some craftsmanship. The main idea is to randomly exercise all code. If the input parameters are sampled on $[-\infty, \infty]$ only the code that handles out-of-range values will be tested. Instead, the sampling should be made in such a way that the all the code is evenly exercised.

No empirical data describing the fault size has been found and used in this paper. Instead, ontological (the approximate Gaussian distribution is common) and epistemological (if all we want to say about a distribution is their mean and variance) arguments were used. If a fault is encountered during development it is corrected, without further analysis of its size in terms of failure density. Only if a critical fault is found, which has already been deployed and put into service, such an analysis mandated. Due to its sensitive nature such data is hard to come by.

IX. CONCLUSIONS

The contribution of this paper is a defined prediction formula for calculation of the undetected failure density of the software. This prediction quantifies the percentage of the input space that can be expected to cause failures of the software for a random input. In order to perform this calculation a sufficiently large number of test cases need to be executed, typically this is done with random testing. The large number ensures that the estimates are sufficiently precise. Furthermore, a root cause analysis need to be performed to identify the number of detected unique faults.

ACKNOWLEDGMENT

This work was partly supported by The Swedish Knowledge Foundation (KKS) through the research profile Dependable Platforms for Autonomous Platforms and Control (DPAC), and the Industrial Graduate School for Reliable Embedded Sensor Systems (ITS ESS-H).

REFERENCES

[1] J. A. McDermid and T. P. Kelly, "Software in safety critical systems: Achievement and prediction," *Nuclear Future*, vol. 2, no. 3, pp. 140–146, 2006.

[2] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan, "Evaluation of safety-critical software," *Communications of the ACM*, vol. 33, no. 6, pp. 636–648, Jun. 1990.

[3] RTCA, "DO-178C, software considerations in airborne systems and equipment certification," 2011.

[4] IEC, "Functional safety of electrical/electronic/ programmable electronic safety-related systems—Part 7: Overview of techniques and measures," International Electrotechnical Commission, Geneva, Switzerland, Standard IEC 61508-7:2010, apr 2010.

[5] ISO Central Secretary, "Road vehicles—Functional safety," International Organization for Standardization, Geneva, Switzerland, Standard ISO 26262-1:2011, nov 2011.

[6] IEC, "Nuclear power plants—Instrumentation and control important to safety—General requirements for systems," International Electrotechnical Commission, Geneva, Switzerland, Standard IEC 61513:2011, aug 2011.

[7] D. Hamlet and J. M. Voas, "Faults on its sleeve: Amplifying software reliability testing," in *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '93. New York, NY, USA: ACM, 1993, pp. 89–98.

[8] B. Littlewood and L. Strigini, "Validation of ultrahigh dependability for software-based systems," *Communications of the ACM*, vol. 36, no. 11, pp. 69–80, Nov. 1993.

[9] D. R. Miller, "Making statistical inferences about software reliability," National Aeronautics and Space Administration, Scientific and Technical Information Division, Tech. Rep. NASA CR-4197, 1988.

[10] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Software Eng.*, vol. 25, no. 5, pp. 675–689, 1999.

[11] J. M. Voas, C. C. Michael, and K. W. Miller, "Confidently assessing a zero probability of software failure," in *SAFECOMP '93*, J. Górski, Ed. London: Springer, 1993, pp. 197–206.

[12] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, March 2012.

[13] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," *ACM*

SIGSOFT Software Engineering Notes, vol. 30, no. 4, pp. 1–7, 2005.

[14] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 435–445.

[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[16] A. German and G. Mooney, "Air vehicle software static code analysis lessons learnt," in *Aspects of Safety Management*, F. Redmill and T. Anderson, Eds. London: Springer London, 2001, pp. 175–193.

[17] S. Mankefors, R. Torkar, and A. Boklund, "New quality estimations in random testing," in *14th International Symposium on Software Reliability Engineering (ISSRE 2003)*. New York, NY, USA: IEEE Computer Society, Nov. 2003, pp. 468–478.