The Journal of Systems and Software 84 (2011) 587–602

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On the estimation of adequate test set size using fault failure rates

Vidroha Debroy, W. Eric Wong*

Department of Computer Science, The University of Texas at Dallas, MS EC 31, 800 West Campbell Road, Richardson, TX 75080, USA

ARTICLE INFO

Article history:
Received 16 January 2010
Accepted 7 July 2010
Available online 11 August 2010

Keywords:
Fault failure rate
Fault detection
Software testing
Test set size
Test case

ABSTRACT

Test set size in terms of the number of test cases is an important consideration when testing software systems. Using too few test cases might result in poor fault detection and using too many might be very expensive and suffer from redundancy. We define the failure rate of a program as the fraction of test cases in an available test pool that result in execution failure on that program. This paper investigates the relationship between failure rates and the number of test cases required to detect the faults. Our experiments based on 11 sets of C programs suggest that an accurate estimation of failure rates of potential fault(s) in a program can provide a reliable estimate of adequate test set size with respect to fault detection and should therefore be one of the factors kept in mind during test set construction. Furthermore, the model proposed herein is fairly robust to incorrect estimations in failure rates and can still provide good predictive quality. Experiments are also performed to observe the relationship between multiple faults present in the same program using the concept of a failure rate. When predicting the effectiveness against a program with multiple faults, results indicate that not knowing the number of faults in the program is not a significant concern, as the predictive quality is typically not affected adversely.

1. Introduction

Fundamental to the software testing process is the utilization of good test cases to detect faults. In fact, the number of such test cases (i.e., the size of the test set) used to test software has a significant impact on the quality of testing. If a test set consists of too few test cases, it may not be able to detect faults effectively. On the other hand, if a test set has too many test cases, it would imply a greater cost associated with the testing, and may also result in some test cases being redundant. Test set size is therefore one of the primary determinants of the overall cost and effectiveness of the software testing process. Consequently, selecting an appropriate test set size becomes a very important concern for testers and managers everywhere, who want to improve the effectiveness of software testing while reducing its cost.

Test sets may be constructed to satisfy one or more criteria, such as various forms of code coverage, based on certain aspects of the software under test. A test set is adequate for a selected criterion if it covers the program according to that criterion (Harrold, 1991; Weyuker, 1986). For the purposes of this paper, test set adequacy is discussed in terms of the test set's fault detection effectiveness (which is to be formally defined later in this paper). Recognizing that large, comprehensive sets of test cases are rarely available and impractical to use due to their expense, several studies have explored the link between the size of a test set and its corresponding effectiveness in terms of its fault detection capabilities (Rothermel and Harrold, 1998; Wong et al., 1998, 1999). In contrast this study explores this link from the opposite direction, in that it proposes a model to estimate the corresponding number of test cases (test set size) that must be used in order to reach a certain level of fault detection effectiveness.

In order to build such a predictive model, assumptions similar to the ones made in random testing scenarios are also made in this paper. In random testing, test cases are selected randomly from the entire input domain (Chen and Yu, 1996; Leung et al., 2000), thus assuming that each test case may be equally as good (or bad) as the next in terms of its ability to detect a fault (result in failure). Random testing also assumes that the cost associated with the execution of one test case is the same as the cost associated with another. Since our empirical studies conducted are based on subject programs with pre-existing available test cases, we restrict these random testing assumptions to test cases from the available test pool rather than the entire input domain. Stated differently, for the purposes of this paper and the descriptions of experiments herein, when constructing a test set of a certain size – say *x* test cases, those *x* test cases are selected randomly (without replacement) from the entire available pool of test cases.

The model is developed by making use of failure rates – informally to be interpreted as the hardness of detection of fault(s) in a program (and formally to be defined subsequently in Section 2.3). Our approach utilizes these failure rates to make a probabilistic

* Corresponding author. Tel.: +1 972 883 6619; fax: +1 972 883 2399.
E-mail addresses: vxd024000@utdallas.edu (V. Debroy), ewong@utdallas.edu (W.E. Wong).

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.07.025

estimation of the expected number of faults (faulty programs[1]) detected by a test set. Supposing we have a total of n different faults that we are working with, we can reverse the above process to identify the test set size (number of test cases) such that all of the n faults are expected to be detected. Thus, we make it possible to predict an adequate test set size such that all the faults are expected to be detected, i.e., we expect to have 100% fault detection effectiveness. It is possible for us to have 100% fault detection effectiveness because we know exactly how many single-fault versions there are (n is a known variable), and in our case studies we restrict ourselves to faults that result in at least one test case failure. Then the quality of the prediction is evaluated by comparing the expected number of faults detected (as predicted by the model using a test set of a certain size) to the actual number of faults detected by real test sets of the same size.

Thus, the contributions made by this paper may be summarized as follows:

(1) We posit the use of fault failure rates as useful estimators of adequate test set size with respect to fault detection effectiveness (to be detailed in Section 2.1).
(2) To this effect, a predictive model is presented to illustrate how the above might be achieved.
(3) We present empirical data to illustrate the effectiveness and predictive quality of the proposed model via the seven programs of the Siemens suite, and the space, grep, gzip, and make programs (i.e., a total of 11 programs are experimented upon).
(4) The interference (or lack of independence) between multiple faults present in the same program is also investigated via the notion of failure rates.

The remainder of the paper is organized in the following manner. Section 2 first describes the fundamental concepts that help facilitate better understanding of this paper and its discussions. Section 3 then presents the probabilistic model that estimates adequate test set size using the failure rates, followed by Section 4 which evaluates the model against 11 C programs. Subsequently Section 5 identifies two relevant issues and provides more insights on the proposed model and derived data. Section 6 assesses the quality of the proposed model in the presence of incorrect failure rates and three different perturbation schemes are presented. Section 7 then goes on to evaluate the relationship between multiple faults in the same program from the point of view of the failure rate. Section 8 presents relevant discussions and details the threats to the validity of the approach and Section 9 is an overview of related work. Finally, we present our conclusions and ideas for future work in Section 10.

2. Preliminaries

In this section we present concepts that are fundamental to the understanding of the model and corresponding discussions presented in this paper.

2.1. Fault detection effectiveness

Given a set of faulty versions (faults) of the same program and a test set T, we define the fault detection effectiveness E as the percentage of faults detected by T. When we say 'detected by T' we mean that at least one of the test cases in T must result in execution failure (i.e., the test case fails) on the faulty version. For example, in a set of five faults, i.e., five programs with one fault each – if at least one test case in T fails on three of the programs (it does not have to be the same test case for all of the programs), but none fails on the other two, then the effectiveness is 60%. In this paper, we use the following terminology similar to that defined in (Avizienis et al., 2004). A failure is an event that occurs when a delivered service deviates from correct service. The deviation of system state from correct state is called an error. The adjudged or hypothesized cause of an error is called a fault. Thus, a test case fails on a program when the observed output/behavior differs from that of the expected output/behavior.

2.2. Adequate test set size

For the purposes of this paper, we define an adequate test set size as the number (of test cases) that results in an expected fault detection effectiveness of 100%, i.e., every faulty version is expected to be detected. Once again, this is possible because we know how many faulty versions there are (i.e., n) and we know that each one of them results in at least one failure among the test cases in the pool. Recall that we assume that every test case in the test set has the same 'potential' to result in failure, and every test case is associated with the same cost. Thus, given a test pool of u test cases, a test set with α test cases can be constructed by randomly selecting α test cases from the test pool ($\alpha \leq u$). We also assume that every test case in the test set is unique (no duplicate test cases) because repeating the same test case offers no gain in fault detection effectiveness, assuming the execution environment does not change. Thus, the α test cases that comprise the test set are selected without replacement.

2.3. Fault failure rate

Given a test pool (test cases that are available for use) that contains u test cases, let ω be the number of test cases that fail when executed against a faulty program (fault) f. Then the failure rate θ of the fault can be defined as the proportion of test cases from the pool that fail on the fault. Thus, $\theta = \omega/u$. Assuming that a test set T is constructed uniformly, at random from the test pool, then the definition of the failure rate can also be extended to that test set. Let us say, T consists of α test cases, out of which ε (where ε is between 0 and α) test cases fail on the faulty program. We therefore have, $\omega/u = \varepsilon/\alpha = \theta$.

The failure rate θ can be said to represent the hardness (or ease) of detection of a fault. To better understand what we mean by this, consider that a high failure rate implies that a relatively larger number of test cases from the test pool result in failure on the fault in question. This means that we will not have to sample too many test cases from the pool until one of the test cases fails, and we can say that we have detected the fault, which in turn means that the fault is relatively easy to detect. Instead, if a fault has a relatively lower failure rate, it implies that we may have to sample more test cases until a failing test case is encountered. Thus, such a fault will be relatively harder to detect. Roughly speaking we can therefore say that the lower the failure rate of a fault, the harder it is to detect. Note that a fault that is capable of being detected must have a non-zero failure rate (because in order to detect it, at least one test case fail on it), and also, the failure rate is restricted to a value between 0 and 1.

Now given the failure rate θ, for any α-sized test set T (i.e., with α test cases), we can compute the expected number of test cases that will fail (ε) as, $\varepsilon = \theta\alpha$. When this number reaches 1, we know that we expect to have at least one test case in the test set that will result in failure, thereby allowing us to detect that the program is faulty. Identifying such an α will allow us to identify the size of a

[1] Initially we assume that each faulty program has a single-fault in it. Thus, hereafter we use 'fault' to denote a faulty program with exactly one fault in it. Such an assumption is later relaxed in Section 7 where we discuss programs with multiple faults in them.

test set – which can be considered to be fault detection adequate with respect to this fault, as the fault is expected to be detected. It is this idea that forms the basis for the model proposed in the paper.

3. The proposed model

Let there be a set of n faults: $f_1, f_2, \ldots, f_n$. Let there also be a test set (without loss of generality, let us assume that this is our test pool) T that contains m test cases: $t_1, t_2, \ldots, t_m$. Let each fault f_i be associated with a failure rate θ_i such that

$$\theta_i = \beta_i / m \tag{1}$$

where β_i represents the number of test cases out of m that fail on fault f_i. Supposing we wish to use a subset of T, say T_α with α test cases such that $\alpha \le m$. We define $p_i^F(\alpha)$ as the probability of detecting fault f_i by a test case in T_α. The outcome of the fault detection process is binary because f_i is either detected by T_α or it is not. Therefore, we can say

$$p_i^F(\alpha) = 1 - p_i^S(\alpha) \tag{2}$$

where $p_i^S(\alpha)$ is the probability of not detecting f_i, i.e., none of the α test cases in T_α fails on the fault. Let Y_i denote the outcome of the fault detection process for fault f_i; then Y_i can take one of two values. Let us say

$$Y_i = \begin{cases} 1 & \text{if fault } f_i \text{ is detected} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Let the expected value of Y_i be defined as

$$EX(Y_i) = (1)(p_i^F(\alpha)) + (0)(1 - p_i^F(\alpha)) = p_i^F(\alpha) \tag{4}$$

Note that each faulty version has only one fault and therefore, the detection of a fault is independent of the detection of another. The expected number of faults detected by T_α (which is the sum of the individual expectations) is computed as

$$p_1^F(\alpha) + p_2^F(\alpha) + \cdots + p_n^F(\alpha) = \sum_{i=1}^{n} p_i^F(\alpha) \tag{5}$$

Accordingly, the effectiveness $\mathcal{E}$ is computed as

$$\mathcal{E} = \left(\frac{\sum_{i=1}^{n} p_i^F(\alpha)}{n} \right) \times 100 \tag{6}$$

Combining Eqs. (2) and (6), the effectiveness equation now reduces to

$$\mathcal{E} = \left(1 - \frac{\sum_{i=1}^{n} p_i^S(\alpha)}{n} \right) \times 100 \tag{7}$$

For any fault f_i, $p_i^S(\alpha)$ may be computed as follows

$$p_i^S(\alpha) = \frac{(m-\beta_i)!(m-\alpha)!}{(m-\beta_i-\alpha)!(m)!} \tag{8}$$

This computation is very similar to that of the Q-measure in random testing (Leung et al., 2000) that is used to compute the probability of detecting at least one failure when test case selection is done without replacement. Selection of test cases with replacement would lead us to simpler mathematical models. However, we wish to identify an adequate test set size (to detect all the n faults used in the study) while avoiding redundancy. Thus, we force ourselves to only consider test sets with distinct, non-repetitive test cases and duplicity within a test set is avoided. The effectiveness computation now resolves to

$$\mathcal{E} = \left(1 - \frac{\left(\frac{(m-\alpha)!}{m!}\right) \left\{ \frac{(m-\beta_1)!}{(m-\beta_1-\alpha)!} + \cdots + \frac{(m-\beta_n)!}{(m-\beta_n-\alpha)!} \right\}}{n} \right) \times 100 \tag{9}$$

Given n faults for testing, their respective failure rates $\theta_1, \theta_2, \ldots, \theta_n$ (where $\theta_i = \beta_i/m$), and the desired level of effectiveness $\mathcal{E}$, we can solve Eq. (9) for α in order to compute an expected test set size to reach that level of fault detection effectiveness.

Only integer solutions of α are meaningful and applicable because there is no such thing as a fractional test case. Unfortunately, using this equation to compute the value of α is analytically complex and computationally intractable (even with approximations such as Stirling's approximation (Knuth, 1997)) for increasingly large values of n. Therefore, we compute the value of α numerically, by successive substitution (namely, $\alpha = 1$, $\alpha = 2$, and so on), until the desired level of effectiveness is achieved.

4. Case studies

The model proposed in Section 3 was evaluated via case studies conducted on a total of 11 programs: the seven programs of the *Siemens suite*, the *space* program, the *grep* program, the *gzip* program, and the *make* program. A discussion regarding the choice of programs and why we feel they are ideal for our cases studies is presented in the Threats to Validity discussion in Section 8.5 of this paper. Additionally, detailed information on each program is provided as follows.

4.1. Subject programs

Siemens suite: The Siemens suite consists of seven distinct programs, namely – print_tokens, print_tokens2, schedule, schedule2, replace, tcas and tot_info. The correct versions, 132 faulty versions of the programs, and all the test cases were downloaded from Siemens (2007). Of these 132 faulty versions, three were excluded in our study: version 9 of "schedule2" because it did not result in any test case failure, and versions 4 and 6 of "print_tokens" because these faults were located in header files and not in the .c files

Space: The space program developed at the European Space Agency provides a language-oriented user interface that allows the user to describe the configuration of an array of antennas using a high level language (Cancellieri and Giorgi, 1994). The correct version, 38 faulty versions, and a suite of 13,585 test cases used in this study were downloaded from a software-artifact infrastructure repository (http://sir.unl.edu/portal/index.html). In another study making use of the space program (Jones and Harrold, 2005), eight of the 38 faulty versions were not used for various reasons. However, in our case, only three faulty versions were not used because there were no test cases that failed on these faulty versions

Grep: The source code of version 2.2 of the grep program was downloaded from http://sir.unl.edu/portal/index.html, along with 18 faulty versions and a suite of 470 test cases. Compared with the study in Liu et al. (2006), where none of these faults could be detected by any test case in the suite, we were able to detect four faults in our environment (discarding the other 14 downloaded faulty versions). Two additional faults injected by Liu et al. (2006) were also used. The authors of Liu et al. (2006) argue that although faults are manually injected, they do resemble realistic logic errors. An additional 13 faults were injected by us following the same approach, bringing the total number of faulty versions to 19. Further discussion on the injection of faults is presented in Section 8.5.

Gzip: Version 1.1.2 of the gzip program with 16 seeded faults and 217 test cases were also downloaded from http://sir.unl.edu/portal/index.html. Nine faults were excluded from the study since none of the test cases failed on them and six test cases were discarded because they could not be executed in our environment (see Section 4.2 for details). As before, we followed a similar approach as described in Liu et al. (2006) to

Table 1
Summary of the programs used in our studies.

Program	Number of faults used	Number of test cases used	LOC
print_tokens	5	4130	565
print_tokens2	10	4115	510
schedule	9	2650	412
schedule2	9	2710	307
replace	32	5542	563
tcas	41	1608	173
tot_info	23	1052	406
space	35	13,585	9126
grep	19	470	12,653
gzip	28	211	6573
make	31	793	20,014

inject 21 faults in addition to the seven usable original faults. Thus, in total, 28 faulty versions were used.

Make: Version 3.76.1 of make was downloaded from http://sir.unl.edu/portal/index.html along with 793 test cases and 19 faulty versions of the program. Of these, 15 faulty versions were excluded as they contained faults which could not be detected by any of the downloaded test cases in our environment. Using the above mentioned fault injection approach we generated an additional 27 faults for a total of 31 faulty versions to be used.

Table 1 presents a summary of the programs used in our studies in terms of the number of faults used, the number of test cases used, and the size of the programs in terms of the LOC (lines of code) count.

From the table we observe that each program has a varying number of test cases and faulty versions available. The study allows for a contrast between programs in terms of size (LOC) as the programs of the Siemens suite are of a relatively smaller size, whereas the space, grep, gzip and make programs are of a relatively larger size. Additionally, the study also allows for a contrast between programs in terms of the test cases (for example, gzip has 211 test cases as opposed to the 13,585 test cases for the space program), and faulty versions/faults (for example, print_tokens has five faulty versions whereas tcas has 41). These programs have been used extensively in testing and fault localization studies such as Cleve and Zeller (2005), Debroy and Wong (2010), Jones and Harrold (2005), Liu et al. (2006), Renieris and Reiss (2003) and Wong et al. (2008a,b, 2010) to name a few.

4.2. Data collection

All data was collected based on program executions on a PC with a 2.13 GHz Intel Core 2 Duo CPU and 8 GB physical memory. The operating system was SunOS 5.10 (Solaris 10) and the compiler used was GCC 3.4.3. The success or failure of an execution was determined by comparing the outputs of the faulty version and the correct version of a program. A failure was recorded whenever there was a deviation of observed output from the expected output. Also in our experiments, failure rates are empirically computed by executing all the available test cases against each faulty version of a program and observing the fraction of failed test cases on that faulty version. A discussion on the incorrect estimation of failure rates is presented subsequently in Section 6.

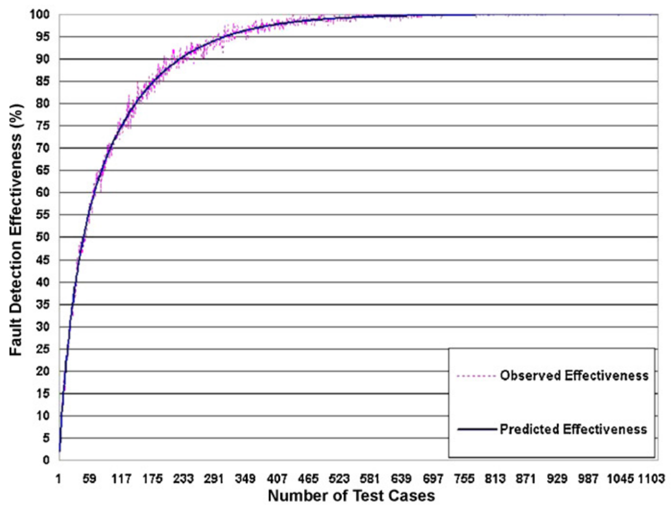
To evaluate how well the model predicted the fault detection effectiveness of a particular sized test set, an actual test set of the same size was constructed and the test cases in it were executed against the faults to observe the actual fault detection effectiveness (for that size). Thus, for each program, test sets were randomly generated for all possible test set sizes, i.e., between 1 and the number of test cases available for that program. Furthermore, multiple test sets were generated for each size, and the value of the observed effectiveness for that size was obtained by averaging the observed effectiveness values for each set of that size. This was done to minimize possible bias due to the choices of the test cases in each test set. For test sets between the sizes of 1 and 5000, 60 sets were generated for each size. For test sets between the sizes of 5001 and 10,000, 30 sets were generated for each size. Finally, for test sets between the sizes of 10,001 and 13,584, 10 sets were generated for each size. The reason that a different number of test sets were generated for different sizes is that as test set size increases, it becomes progressively harder and more time consuming to generate distinct test sets.

4.3. Results

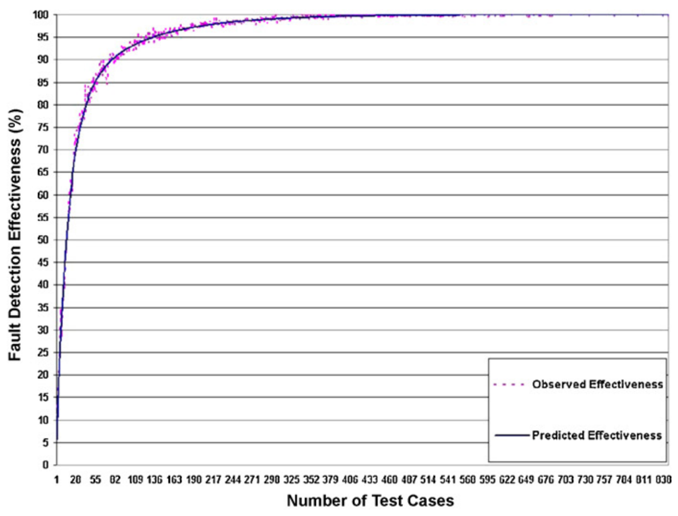
In this section we present comparisons between the effectiveness as predicted by Eq. (9) and the observed effectiveness for the programs under study, as detailed by the procedure in Section 4.2. The comparison is drawn until the predicted effectiveness reaches a level of 100% (we round off to 100% past three decimal places, i.e., 99.999% and above ≈100%). We remind the reader that effectiveness levels of 100% are possible in our controlled experiments because we exclude those faults that do not result in the failure of any of the test cases in the test pool. Were there a fault that is for all practical purposes undetectable, then the predicted effectiveness can never reach a level of 100%. Also, as long as every fault can be detected by at least one test case in a test set, a trivial way to guarantee 100% effectiveness is to simply use every test case in the set. However, this may not be cost-effective and thus is not a very practical solution. Parts (a)–(k) of Fig. 1 graphically illustrate the effectiveness comparison for the print_tokens, print_tokens2, schedule, schedule2, replace, tcas, tot_info, space, grep, gzip, and make programs, respectively. For each of the graphs, the x-axis represents the test set size and the y-axis represents the fault detection effectiveness.

We observe that in general the predicted effectiveness follows the observed effectiveness quite closely. Without loss of generality, we use the program print_tokens as an example. The maximum error (difference) between observed effectiveness and predicted effectiveness is 4.38%, and this is when just one test case is used (i.e., the test set consists of only one test case). When the test set consists of just one test case, then the choice of test case is extremely relevant as some test cases may be able to detect a lot more faults than others. Thus, there may be a drastic change in the observed effectiveness as we move from one test case to another. Intuitively, the smaller the size of the test set used, the greater the possibility of encountering such bias. The impact of test set size on prediction error (defined as observed effectiveness minus the predicted effectiveness) is further discussed in Section 5.2. Another thing to note is that while the predicted effectiveness monotonically increases with an increase in test set size until all of the faults have been detected, it may not be the case with the observed effectiveness. It is possible for a test set of size $(k+1)$ to have a lower effectiveness than that of a test set with k test cases. Admittedly however, the figures are not very clear, primarily due to the large number of data points and because of how close the predicted effectiveness is to the observed effectiveness. We therefore present the data in the form of a box plot in Fig. 2 to alleviate the problem.

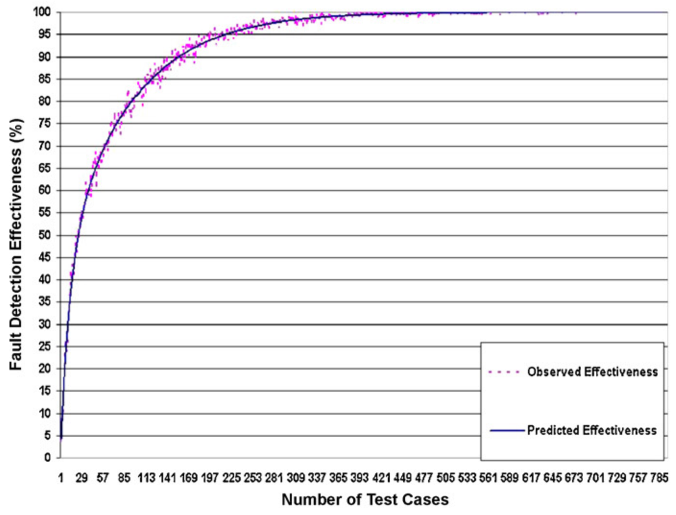
Based on the plots, we draw several conclusions. First, in none of the programs does the error seem to be highly skewed in either direction. Thus, the prediction error fluctuates rather evenly between positive and negative values, hinting at normality. Stated differently, we do not seem to have a tendency to specifically over-predict or under-predict in terms of effectiveness. However, these are simple deductions that are made at first glance based on the box plots. Further discussion on whether the predictions are conservative (generally under-predicting) or liberal (exaggerated or



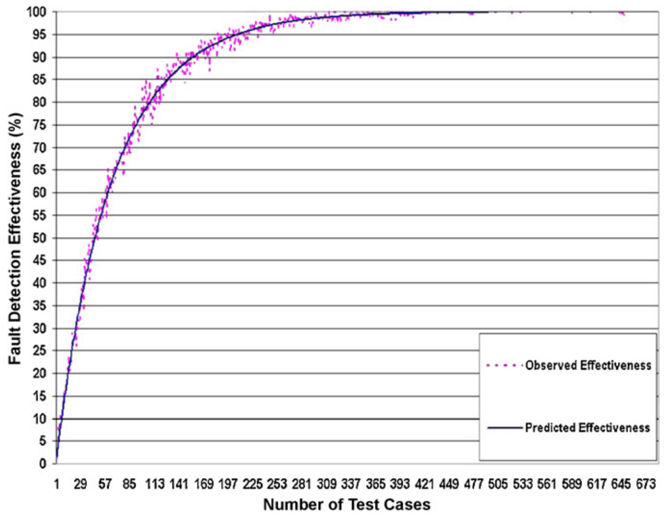
(a) Effectiveness comparison on print_tokens



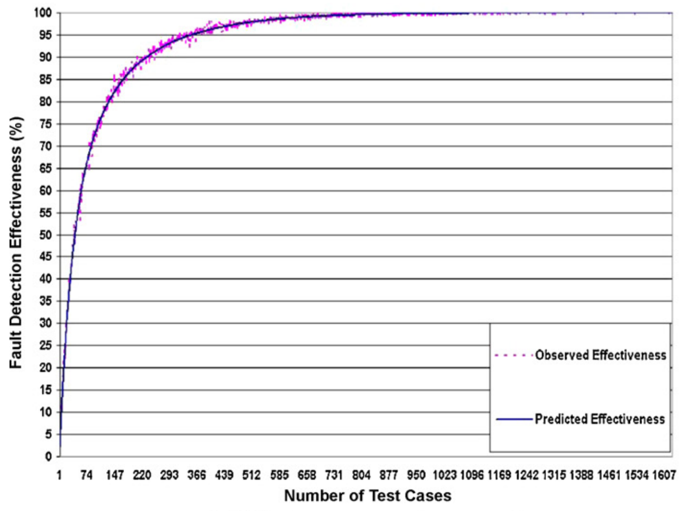
(b) Effectiveness comparison on print_tokens2



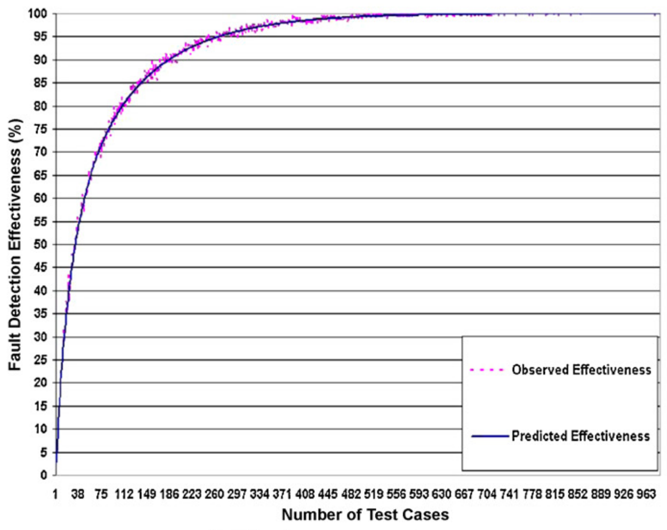
(c) Effectiveness comparison on schedule



(d) Effectiveness comparison on schedule2

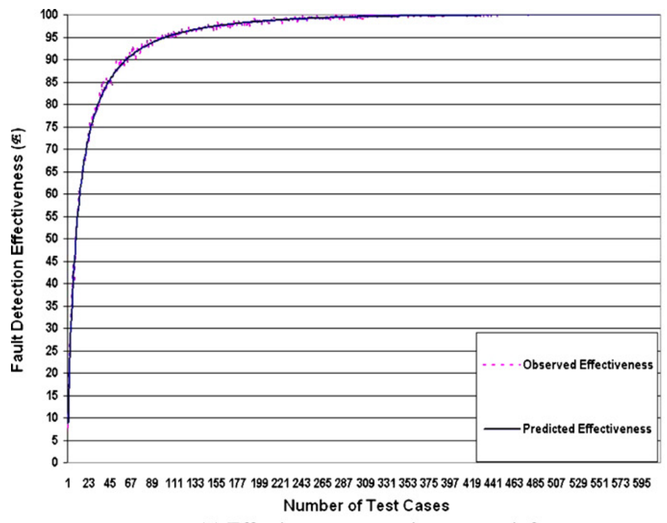


(e) Effectiveness comparison on replace

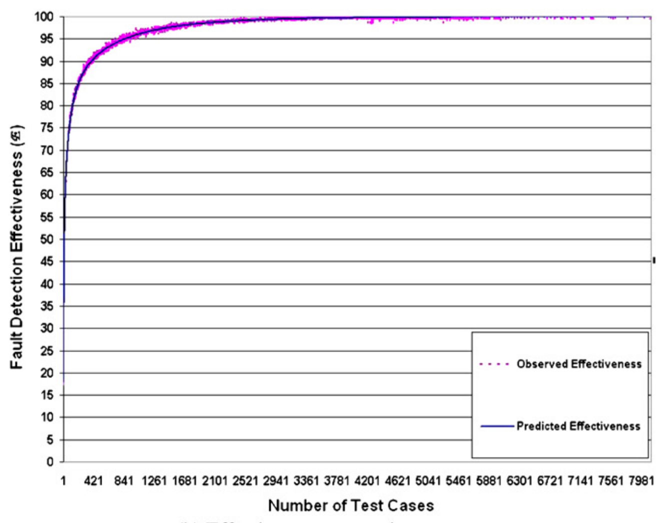


(f) Effectiveness comparison on tcas

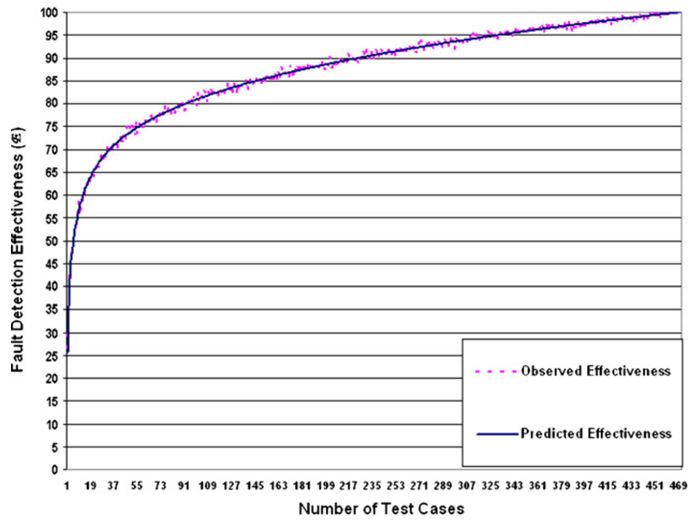
Fig. 1. Comparison between the predicted effectiveness and the observed effectiveness across varying test set size.



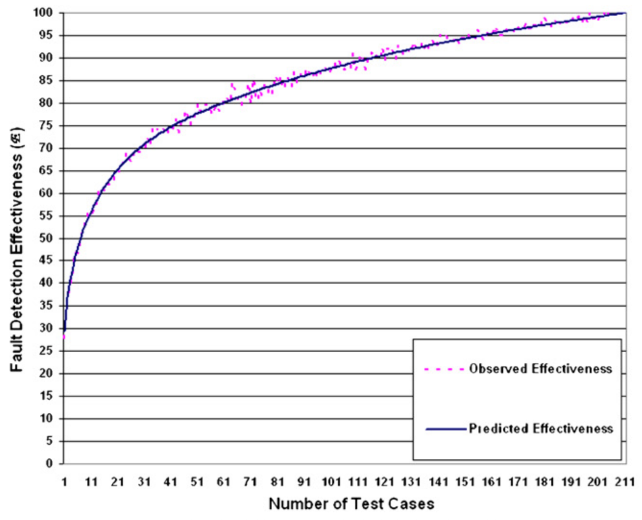
(g) Effectiveness comparison on tot_info



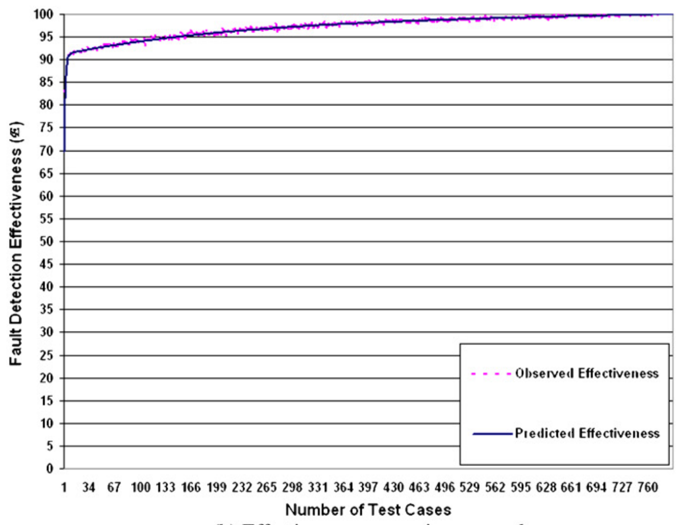
(h) Effectiveness comparison on space



(i) Effectiveness comparison on grep



(j) Effectiveness comparison on gzip



(k) Effectiveness comparison on make

Fig. 1. (*Continued*).

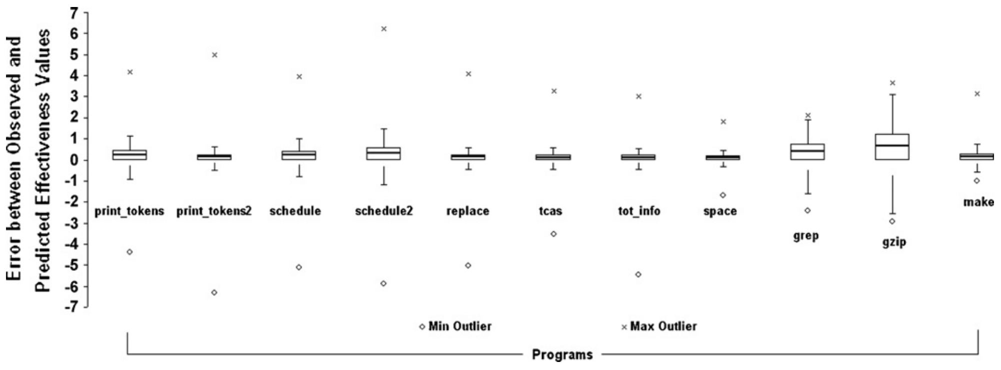


Fig. 2. Box plots representation of prediction error across all 11 subject programs.

over-predicting) is presented in Section 5.1. Second, we observe that for none of the programs is the error extremely large. In most of the cases, the error seems to be generally within the range of about ±3% (this range corresponds to the gzip program which in fact has the largest error range). In terms of outliers, the maximum outlier value is (an over-prediction by) 6.23% in the case of the program schedule2.

Considering that the objective of the experiment is to identify if failure rates can assist in identifying an adequate test set size, it is useful to note the circumstances under which either the predicted or observed effectiveness reaches exactly 100%. We recall that the observed effectiveness of any test set of a particular size is computed as the average effectiveness across multiple sets of that size in order to eliminate possible bias. Therefore, in order to achieve 100% effectiveness, each set must be able to detect all of the faults, i.e., each set must have an effectiveness of 100%. In contrast, for any set of faults (faults that do not result in test case failure are excluded), the predicted effectiveness will only be 100% when the size of the test set is greater than the minimum number of test cases required to guarantee detection of each and every fault. Such a guarantee can only be made when the size of the test set exceeds the number of successful test cases for the fault which caused the minimum number of failures. Recall from Section 3 that α is the size of the test set and β_i represents the number of failed test cases on fault f_i, where i ranges from 1 to n. Stating the above condition differently, whenever we have $\alpha > m - min(\beta_i)$ then the predicted effectiveness will be 100% as we must now include at least one failed test case for every fault. To re-iterate from Section 4.2, for the purposes of this paper a successful test case is one where the observed output is identical to the expected output, and in contrast for a failed test case, the observed output is different from the expected output.

5. Further exploring the results

While the results of the case studies are indeed encouraging (as the error between observed effectiveness and predicted effectiveness is generally low), they lead us to two issues of intrigue. First, we note that the predictions (when not perfectly accurate) are at times higher than the observed effectiveness and at times lower. Therefore, are we more likely to over-predict than under-predict or vice-versa? Second, while the data presented in Fig. 1 may not be as clear (and therefore not as easy to interpret) as the data in Fig. 2, it does offer us a glimpse of the trend between quality of predictions (i.e., the prediction error) and test set size. Thus, this prompts us to ask ourselves, what happens to the prediction error as the size of the test set undergoes change? In this section, we answer these questions with regard to the case studies and the related data presented in Section 4.3.

5.1. *Conservative predictions versus liberal predictions*

To resolve any possible ambiguity at this point, let us start by re-stating what is meant by a conservative prediction and a liberal prediction. By conservative prediction, we mean when the predicted effectiveness for a particular test set size is less than the average observed effectiveness using actual test sets of that same size (i.e., predicted effectiveness < observed effectiveness). Simply put, we under-estimate the effectiveness. Conversely, by liberal prediction we mean when the predicted effectiveness is greater than the average observed effectiveness using actual test sets of that same size (i.e., observed effectiveness < predicted effectiveness). Essentially, we over-estimate the effectiveness.

In order to answer if we are more likely to over-predict than under-predict or vice-versa, and to facilitate the discussion, we equate this problem to deciding whether a coin is unbiased/fair or not. For an unbiased coin, the probability of landing heads is equal to that of landing tails for any toss. Supposing we believe that the coin is weighted towards heads, i.e., the probability of landing heads >0.5. Then a standard hypothesis test could be constructed as:

$$H_0: \ p = 0.5$$
$$H_A: \ p > 0.5$$

where p is the probability that the coin toss results in a 'heads'.

Assuming tosses are mutually independent, the number of 'heads' in any number of coin tosses (an intuitive test statistic) is a binomial random variable (Ott, 1993). If X represents the number of heads in n tosses of the coin and the null hypothesis is true, then we have $X \sim Bin(n, 0.5)$.

In our experiments, the generation of a test set of one size is completely independent of another, and the number of times we conservatively predict or liberally predict can also be represented by a binomial random variable. The above example can be extended to our case – first, such that now p is the probability that we conservatively predict (H_C: $p > 0.5$) and then second, such that p is the probability that we liberally predict (H_L: $p > 0.5$) – resulting in two hypotheses which are tested in a manner similar to the coin toss example. The null hypothesis is untouched in either case (Table 2).

Table 1 lists the confidence levels at which the null hypotheses can be rejected for H_C (conservative estimate hypothesis) and H_L (liberal estimate hypothesis), respectively. From the table we can see that as far as liberal estimates go (i.e., H_L) we can never reject the null hypothesis with a significant degree of confidence for any of the programs studied. On the other hand, we find that for the majority (8 out of 11) of the programs, we are able to reject the null hypothesis at a very high level of confidence in the case of conservative estimates (i.e., H_C). This is suggestive of the fact that the effectiveness predicted by the proposed model is generally conservative in nature in that it is often less than the actual effectiveness observed.

Table 2
Confidence levels at which null hypotheses can be rejected.

Program	H_C	H_L
print_tokens	≈100%	≈0%
print_tokens2	≈100%	≈0% (8.48×10^{-12}%)
schedule	≈100%	≈0% (5.78×10^{-15}%)
schedule2	≈100%	≈0% (4.77×10^{-15}%)
replace	≈100%	≈0%
tcas	≈99.99%	≈1.34×10^{-4}%
tot_info	≈99.96%	≈0.0296%
space	≈100%	≈0%
grep	≈16.58%	≈81.01%
gzip	≈22.39%	≈73.27%
make	≈84.03%	≈14.31%

5.2. Prediction error versus test set size

While Section 5.1 deals with whether the proposed model generally results in conservative or liberal estimates, it does not explain what happens to the prediction error as the test set changes in size. Recall the discussion in Section 4.3, where we describe why there may be a slightly larger prediction error involved with small-sized test sets. Does this imply that as the size of the test set grows, the error in prediction will diminish? While addressing this, we are no longer interested in whether we under-predict or over-predict, but rather care only for the actual magnitude of the prediction error. Therefore, in this section all references to the prediction error refer to the absolute value of the prediction error.

Parts (a)–(k) of Fig. 3 allow us to view the trend in prediction error with respect to an increase in test set size for the programs under study. For each of the graphs, the x-axis represents the test set size and the y-axis represents the absolute prediction error (as defined in the beginning of this section). For the majority of the programs, we clearly observe a decrease in the absolute prediction error as the size of the test set increases. However, the graphs may not be so clear in the case of the other programs, especially space. To alleviate this issue, we also present alternative data in Table 3 in terms of the correlation (Pearson correlation (Ott, 1993)) between the absolute prediction error and test set size for each of the programs.

Based on the data in Table 3 we observe that the correlation is not especially significant for any of the programs. In terms of magnitude, it is lowest in the case of the program print_tokens and highest for schedule. However, another fact which is evident is that the coefficient of correlation is always negative regardless of the program. This means there is an inverse relation between test set size and absolute prediction error. It also suggests that as the test set increases in size (i.e., we have more test cases), the magnitude of the error will diminish (even if the relationship may not be very strong).

Table 3
Correlation between absolute prediction error and test set size for the programs under study.

Program	Pearson coefficient of correlation
print_tokens	−0.052923947
print_tokens2	−0.555281805
schedule	−0.664826517
schedule2	−0.637949305
replace	−0.610922771
tcas	−0.625718412
tot_info	−0.580992359
space	−0.528986606
grep	−0.337783251
gzip	−0.349561173
make	−0.367300078

6. On the incorrect estimation of failure rates

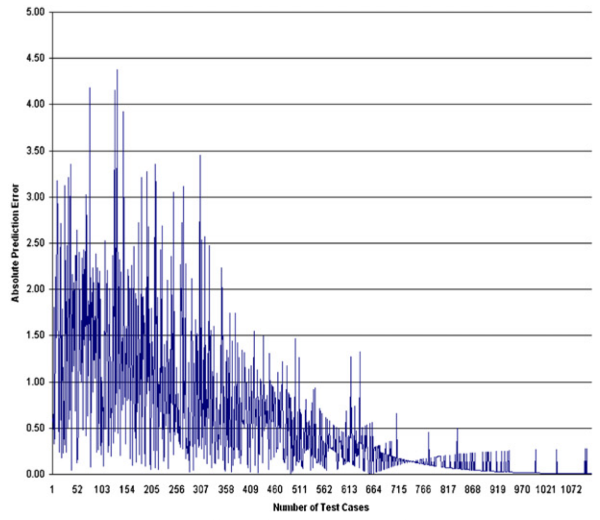
It is arguable that since the proposed model is based on fault failure rates, the accuracy of the model depends in turn on how accurately the failure rates have been estimated. For the purposes of this study, failure rates are directly obtained by running the entire set of available test cases against a faulty version of a program and observing the fraction of failures. We realize that in practice it is very difficult if not entirely impossible to obtain failure rates with such a high degree of accuracy. Furthermore, as a test set is minimized or augmented the corresponding failure rate linked to a fault is also subject to change. Therefore, we decided to extend our experiments to observe the effect on the accuracy of the proposed model in the presence of an imperfect estimation of failure rates.

In order to simulate an incorrect estimation of failure rates, we induce deviation from perfect estimation by modifying the existing failure rates. Recall from Eq. (1) that the failure rate of a fault is the fraction of test cases in the test pool that fail on the fault. Thus, a failure rate of 0.12 would mean that 12% of the test cases in the test pool fail on the fault. To modify the failure rates, we first section the possible failure rates into percentage intervals of 1%, 5%, 10%, 15%, 20%, . . ., 90%, 95%, and 99%. The edge conditions of 0% and 100% are not considered (but instead replaced by 1% and 99%), respectively, because a failure rate of 0% would imply that no test case can detect the fault and a failure rate of 100% would imply that every test case detects the fault. We wish to avoid such extreme scenarios. Each accurate failure rate is then rounded to an appropriate interval based on one of the following rounding schemes: *rounded-up*, *rounded-down*, or *mixed-rounding*. The experiments were repeated using these incorrect failure rates and the data re-collected. However, we point out early on that we refrain from presenting the data in the form of graphs as in Fig. 1. This is done to avoid presenting an excessive number of graphs (11 graphs × 3 rounding schemes = 33 different graphs), and because the usefulness of the graphs themselves is limited due to readability issues (they are not very clear). Instead we present the data as box plots as in Fig. 2. Interested readers may contact the authors for copies of the graphs. Also note that the observed effectiveness does not change in any way and the imperfect failure rates only affect the predicted effectiveness. We now describe each rounding scheme in detail and provide accompanying data on the results obtained.

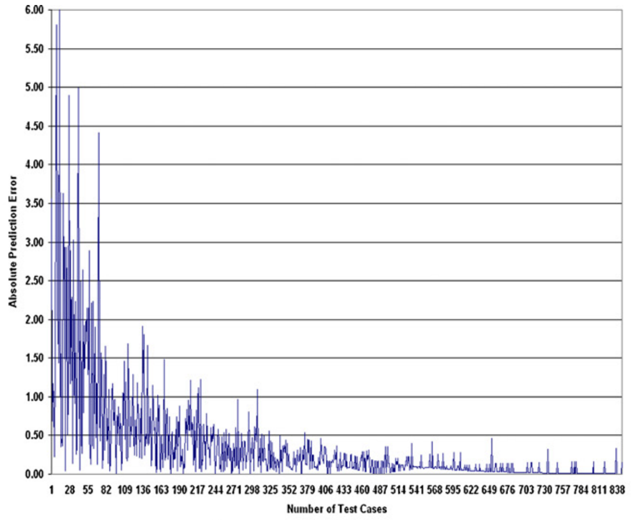
6.1. The rounded-up scheme

This scheme allows for a failure rate to be rounded up to the interval closest to it. Let us again consider the example of a failure rate of 0.12 or 12%. Since the failure rate occurs between the intervals of 10% and 15%, it is rounded up to 15% even though it is in fact closer to 10%. Similarly, any failure rate less than 5% would have been rounded to 5%. The highest possible interval as described above is 99% (instead of 100%). Thus, all failure rates above 95% (including those above 99%) are rounded to 99%. Fig. 4 presents the box plots for all the programs under study based on the incorrect failure rates obtained by the ‘rounded-up’ scheme.

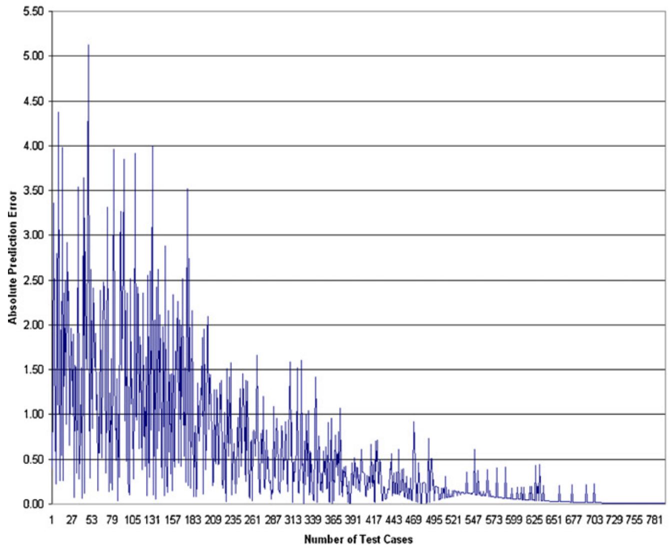
When comparing the box plots in Fig. 4 with those in Fig. 2 we observe that the sizes of the boxes in Fig. 4 are relatively larger for several of the programs. This means that one of the net effects of modifying the failure rates has been that data points that were relatively close to each other are now farther apart. We also observe that the minimum outliers are of a greater magnitude in Fig. 4 than those in Fig. 2. The maximum outliers however are closer to the whiskers than they were before, and some of the programs do not even have maximum outliers anymore. The range (gap between the maximum and the minimum) for several of the programs has also widened more in the case of Fig. 4. An important observation is that the distributions in Fig. 4 are skewed more



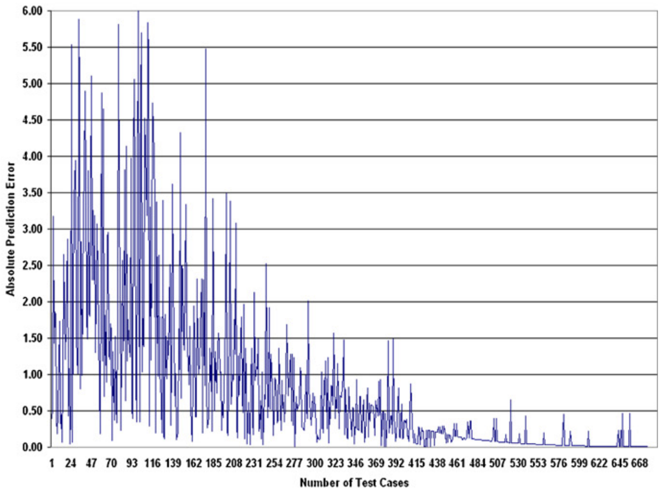
(a) analysis on print_tokens



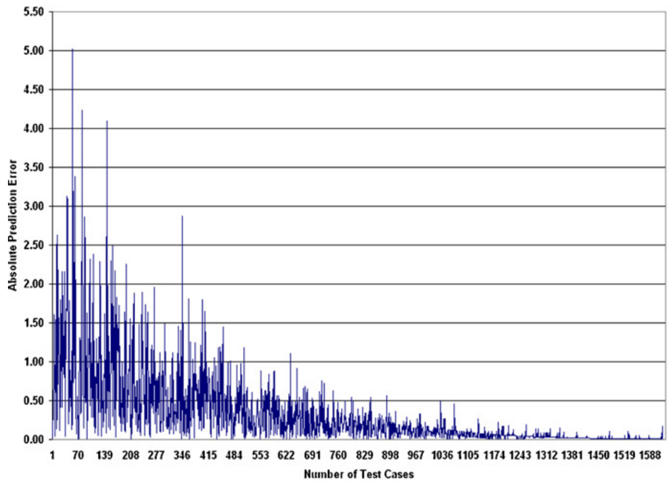
(b) analysis on print_tokens2



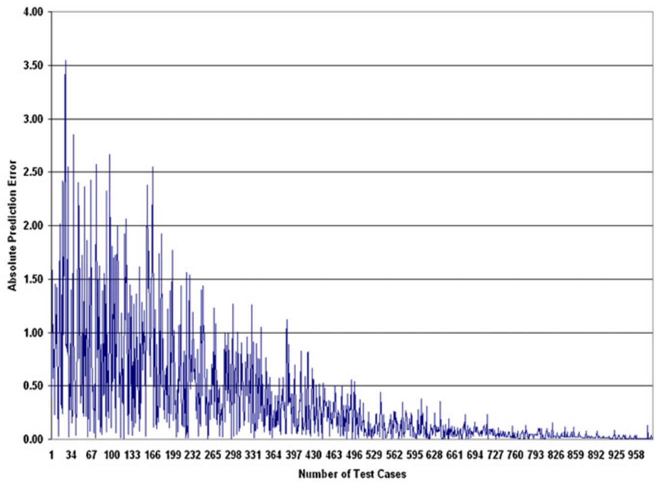
(c) analysis on schedule



(d) analysis on schedule2

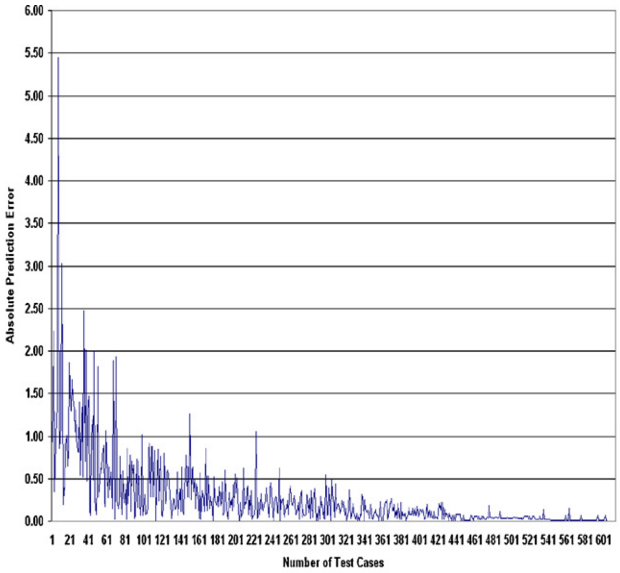


(e) analysis on replace

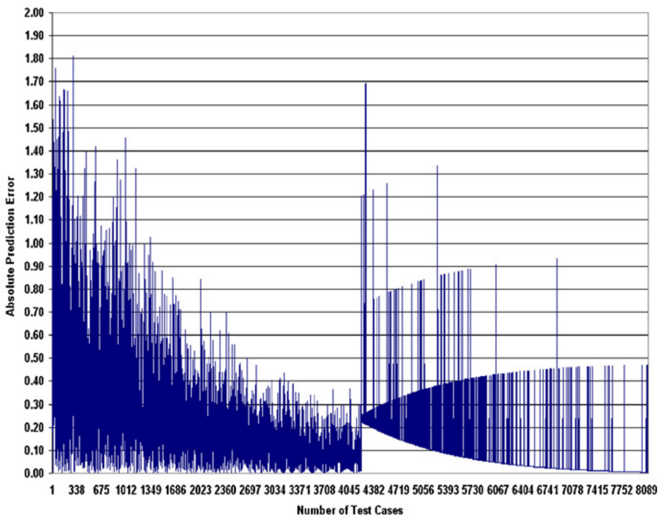


(f) analysis on tcas

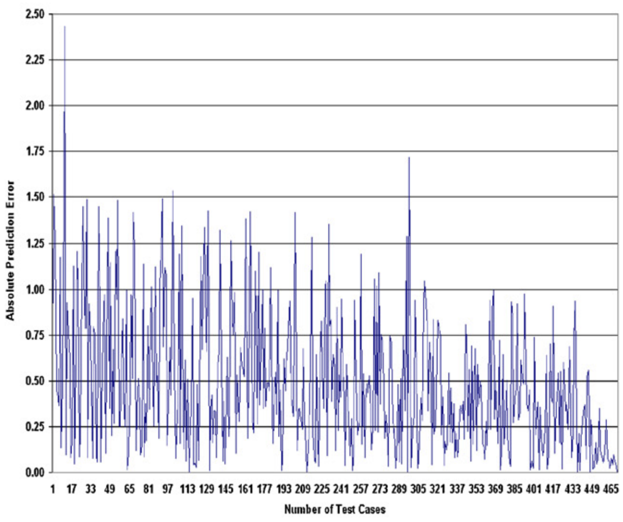
Fig. 3. Change in prediction error with respect to test set size.



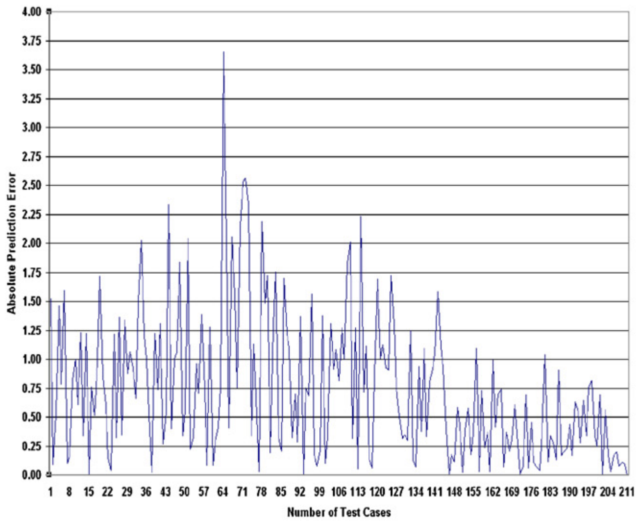
(g) analysis on tot_info



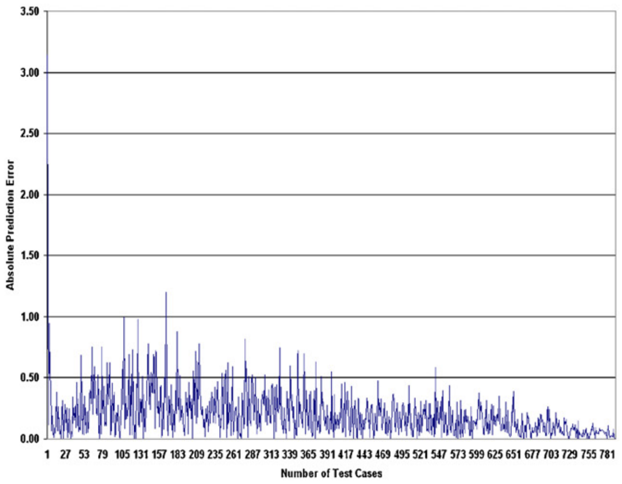
(h) analysis on space



(i) analysis on grep



(j) analysis on gzip



(k) analysis on make

Fig. 3. (*Continued*).

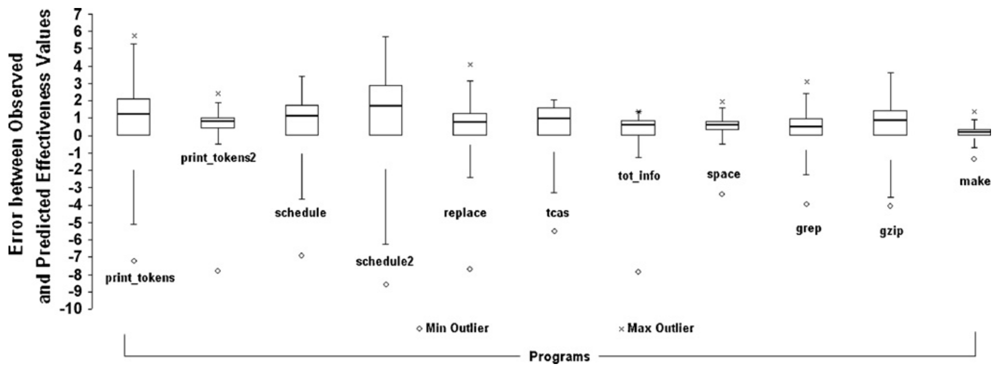


Fig. 4. Box plots representation of prediction error across all 11 subject programs using 'rounded-up' failure rates.

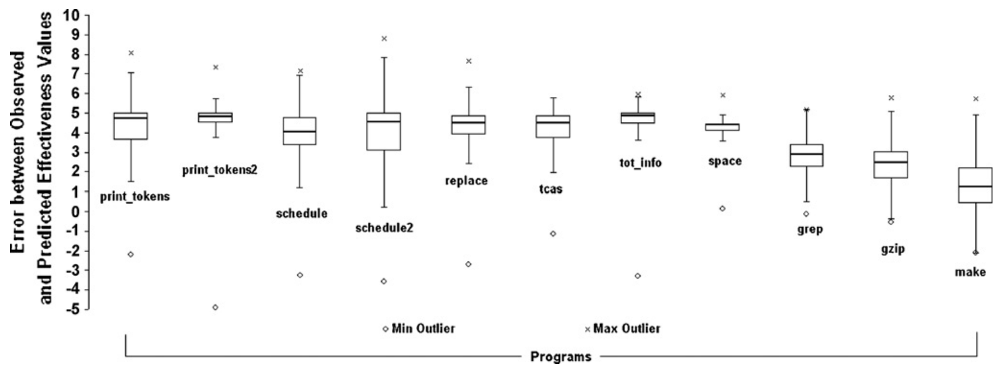


Fig. 5. Box plots representation of prediction error across all 11 subject programs using 'rounded-down' failure rates.

negatively (for several cases, the bottom whisker is considerably longer than the top whisker and extends well below 0) than in the case of Fig. 1. Such negative skewness is especially pronounced in the case of programs such as tcas. To understand why this has happened, we must think about what we have done by adopting the 'rounded-up' scheme. The general idea behind the 'rounded-up' scheme is to over-estimate the failure rate, i.e., assume that more test cases fail on a fault than actually do (i.e., we think faults are easier to detect than they really are). This means that given a test set of any size, we are predicting the detection of a greater number of faults than we really can with a test set of that size. The net result is that for a greater number of cases in the sample, the observed effectiveness is actually less than the predicted effectiveness, i.e., we have a negative prediction error. Thus, the distribution is more negatively skewed than with the exact failure rates. Note, however, that even with the incorrect failure rates the highest outlier value is still well under 10% (the exact value is −8.57% for schedule2). Furthermore, several programs such as make, tot_info, space, and print_tokens2 still have error ranges well below ±3% which means that even after perturbing the failure rates using the 'rounded-up' scheme, we still have good results.

6.2. *The rounded-down scheme*

This scheme allows for a failure rate to be rounded down to the interval closest to it. Taking, once again, the example of a failure rate of 0.12 or 12%; since the failure rate occurs between the intervals of 10% and 15%, it is rounded down to 10%. Similarly, even if a failure rate was 9.9%, it will be rounded down to 5%. The lowest possible interval is 1%. Thus, all failure rates below 5% (including those below 1%) are rounded to 1%. Fig. 5 presents the box plots for all the programs under study based on the incorrect failure rates obtained by the 'rounded-down' scheme.

We note deviations similar to Fig. 4 in terms of range and box-width when comparing Fig. 5 with Fig. 2. However, one of the main

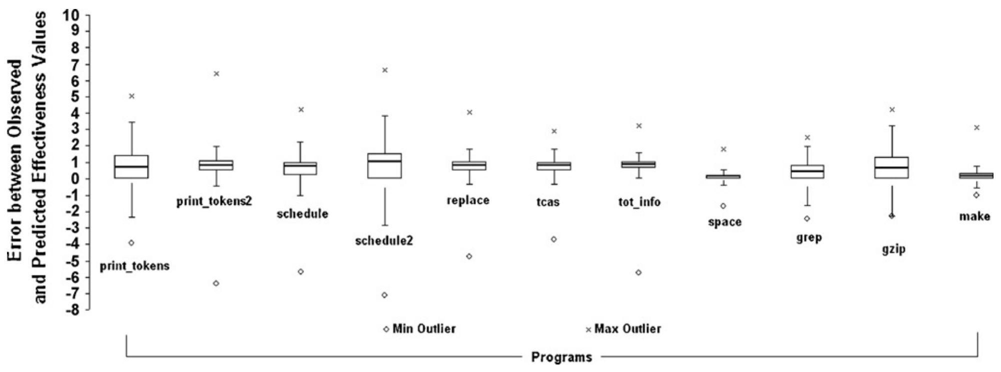


Fig. 6. Box plots representation of prediction error across all 11 subject programs using 'mixed-rounding' failure rates.

distinctions between the plots in Fig. 5 and those in Fig. 4 is that in Fig. 5 the distribution is strongly positive. To understand why this has happened, consider that the ‘rounded-down’ scheme lowers the failure rates, i.e., assuming that fewer test cases fail on a fault than the number that really do. Thus, given a test set of a particular size, we are predicting the detection of a fewer number of faults than what we can really detect with a real test set of that size. The net result is for the prediction error (observed effectiveness – predicted effectiveness) to be generally positive. However, we note that the minimum outliers for any of the programs are always above −5% and the maximum outliers are always below 9%, indicating that while the predictive accuracy has suffered, the quality of the predictions are still good to a considerable degree.

6.3. *The mixed-rounding scheme*

This scheme allows for a failure rate to be rounded either up or down to the interval closest to it. Thus, a failure rate may be rounded-up or -down depending on its position with respect to the median value of an interval. Again considering the example of a failure rate of 0.12 or 12%; since the failure rate occurs between the intervals of 10% and 15% and is less than 12.5% (the median), it is rounded down to 10%. Similarly, a failure rate of 12.5% and above (but below 17.5%) will be rounded to 15%. Failure rates towards the boundary of the possible intervals are handled such that the lowest possible interval is 1% and the highest possible interval is 99%. Thus, all failure rates below 2.5% (including those below 1%) are rounded to 1%, and all failure rates of and above 97.5% (including those above 99%) are rounded to 99%. Note that two significant differences between this scheme and the previous schemes (Sections 6.1 and 6.2) are that – first, in terms of magnitude the ‘mixed-rounding’ scheme only induces a deviation of up to 2.5% as opposed to ≈5% by the previous schemes, and second, while the previous schemes only induced deviation in one direction (rounded-up or rounded-down), this scheme may induce deviation in either direction. Fig. 6 presents the box plots for all the programs under study based on the incorrect failure rates obtained by the ‘mixed-rounding’ scheme.

We note that among the three rounding schemes, the mixed-rounding scheme produces plots (see Fig. 6) most similar to those produced using the accurate failure rates (see Fig. 2) in terms of the range, skewness, etc. Under the mixed-rounding scenario, the points do not seem to be skewed towards any extreme which is explained by the fact that some failure rates are rounded-up and some are rounded-down, giving us a mix of the phenomena discussed in Sections 6.1 and 6.2. Overall, the maximum outlier for any program is always below 7% and the minimum outlier for any program is always above −8%, suggesting that even with the perturbed failure rates, the performance of the proposed model is still good.

7. Programs with multiple faults

Thus far, when we have used the term ‘fault’, we have referred to the faulty version of a program which contains exactly one fault. It has therefore been quite convenient to refer to the failure rate used in this paper as the ‘fault failure rate’. However, this clearly changes if there are multiple faults present in the same program. To continue to refer to the proportion of test cases that fail as the ‘fault failure rate’ would lead to ambiguity as we do not know which fault in the program we are referring to. However, note that the failure rate is not defined in terms of any fault, but rather is simply the proportion of test cases in the pool that result in execution failure against a faulty program. This can now be referred to as the ‘program failure rate’ without any loss of generalization or change of definition. Furthermore, the definition stands irrespective of the number of faults in the program.

The notion of independence of failures is not a new one in the area of software reliability. Several Non-Homogenous Poisson Process Software Reliability Growth Models (NHPP SRGMs) assume software failures occur at random and are independent of each other (Blischke and Murthy, 2003). The area of software reliability often also makes use of Markov models and one of the main features of a Markov model is that a transition from one state to another depends only on the current state (Lyu, 1996). Along these lines, in Reussner et al. (2003) the authors assume that when a service is called, its reliability does not depend on methods previously called, and so failures of service are independent. However, when multiple faults are present in the same program, they may interact with each other in a number of different ways. A test case that would normally have failed due to a single fault may no longer fail when another fault is added to the same program. Also, a test case may fail due to the collective effort of two faults, which does not fail due to any one fault alone. The effects of multiple faults in the same program may also seemingly be independent from one another. In Debroy and Wong (2009) such interference and independence relationships between multiple faults present in the same program are studied, and conclusions are drawn that the ‘independence’ assumption between faults is not a safe one to make. In particular, two forms of interference are studied – constructive interference (where two or more faults collectively cause a test case failure that cannot be attributed to a single fault alone) and destructive interference (where two or more faults interfere with one another such that the failure-causing effect of one or more of them is nullified and a previously failing test case is now successful).

Given two faulty versions of the same program f_1 and f_2, each with just one fault; let the failure rate of the first one be θ_1 and that of the second be θ_2. Since these are both faulty versions of the same program, they can both be tested using the same test set T, which let us assume without loss of generality consists of m test cases. Suppose these two faulty versions are combined together such that the faulty program now contains both faults. Let the failure rate of this 2-fault version of the program (i.e., the program failure rate) be denoted as θ_{12}. Let us bear in mind that we care not as to how many faults are actually in the program or how they are to be debugged. Our scope is limited to proposing the use of failure rates to suggest an adequate test set size that shall result in failure and therefore our end goal is to ensure we have some test case failure that shall indicate that the program is faulty (irrespective of how many faults are actually in the program).

To continue, let us assume that we believe there is one fault in the program (whereas there are really two). To make circumstances harsher let us say we only know the fault with the larger failure rate, i.e., $max(\theta_1, \theta_2)$. Supposing there were only constructive interference, then it would result in the failure of some test cases that were not failing before. Thus, the program failure rate would also go up, i.e., $\theta_{12} \geq max(\theta_1, \theta_2)$. In such a scenario, the observed effectiveness would conceivably be greater than the predicted effectiveness as the likelihood of a test case failing would be higher. Thus, the predicted effectiveness would be incorrect (in the sense that it would under-predict the effectiveness). On the other hand, supposing there were only destructive interference, then it is possible that some of the test cases that we were expecting to fail would no longer fail. Then it is possible that the program failure rate would be less than the individual fault failure rates. In such a situation, the observed effectiveness could be less than the predicted effectiveness (as some of the test cases we were expecting to fail are now resulting in successful executions). This situation is worse than the former because we would rather under-predict than over-predict. Under-predictions are preferred to over-predictions because when we under-predict, at least we know that the effectiveness should not fall below what we predict. We may be using more test cases than we need to, but we are assured of detecting the faults we want to. In contrast, if we over-predict, then there is always the risk of not being able to detect some of the faults, which is against our objective. Note that in Debroy and Wong (2009) it is identified that constructive and destructive interference may also occur together. Based on our discussion, we would hope that the constructive form would dominate over the destructive form such that the net effect would be for the observed effectiveness to be more than the predicted effectiveness.

A typical concern related to programs with multiple faults is that in practice we do not know how many faults are actually present in the program. We may think there is one fault in a program, estimate the failure rate, and use the proposed model; but what happens if there are really more faults in the program? Based on the above discussion, if destructive interference is a rare event, then the net effect of really having more than one fault (but estimating according to only one fault) in a program is that the actual program failure rate is greater than or equal to the estimated failure rate. This in turn means that we would generally under-predict the effectiveness (which is preferred to over-predicting as discussed above) and thus in such a scenario, not knowing the exact number of faults in a program does not pose a significant problem.

The likelihood of such an occurrence can be observed by first combining single-fault versions (whose failure rates are known) and then comparing the program failure rate (of the multiple-fault version) to the individual fault failure rates. Due to resource limitations and since this is not within the scope of our study, we restrict ourselves to only testing on 2-fault versions of the Siemens suite programs. For each multiple-fault program, 2 scenarios can be identified:

Case I: The program failure rate is less than either of the fault failure rates.
Case II: The program failure rate is either greater than or equal to both of the fault failure rates.

Based on our discussion, Case I is what we are trying to avoid and Case II is what is preferred. Regarding data collection, individual fault failure rates are already known from the previous experiments and the program failure rates are computed by running all the test cases against the multiple-fault program and identifying the fraction of test cases that lead to execution failures. 2-Fault versions are generated by seeding together two faults into the same program as long as there is no location conflict (i.e., the faults do not share the same location). Some programs have a greater number of faults that can be seeded simultaneously than others, which is why the number of 2-fault versions per program may vary. Table 4 presents the relevant data regarding this portion of the experiment.

From the above table we observe that out of all the 2-fault versions studied, Case I occurs 13.25% (33/249) of the time. While this is not statistically insignificant, it is nevertheless a rare event and this supports our claim that the proposed model has great value even if the exact number of faults in a program is unknown. Since this experiment has been performed on a small scale, additional rigorous studies and statistical tests need to be conducted to address how likely is it that the proposed model will suffer if one does not identify the number of faults in a program correctly. However, as discussed this is not within the scope of this paper and therefore shall be deferred to subsequent studies.

Table 4
Evaluating fault interference from the point of view of failure rates.

Program	Number of 2-fault versions generated	Number of instances of Case I	Number of instances of Case II
print_tokens	10	0	10
print_tokens2	28	0	28
schedule	28	5	23
schedule2	28	6	22
replace	50	3	41
tcas	50	5	39
tot_info	100	14	86
Total	294	33	249

Table 5
Comparing the adequate test set size as predicted to the total number of available test cases.

Program	Adequate test set size as per proposed model	Total number of test cases available	Percentage of test cases that need to be used
print_tokens	1114	4130	26.97%
print_tokens2	844	4115	20.51%
schedule	794	2650	29.96%
schedule2	687	2710	25.35%
replace	1627	5542	29.36%
tcas	984	1608	61.19%
tot_info	613	1052	58.27%
space	8098	13,585	59.61%
grep	470	470	100.00%
gzip	211	211	100.00%
make	792	793	99.87%

8. Discussion

In this section we present discussions on some of the important aspects of the model proposed herein, address some important concerns, and discuss the threats to the validity of the approach and its derived results.

8.1. The usefulness of the model and its results

The quality of the predictions made by the proposed model has been demonstrated throughout this paper by means of case studies conducted. Based on the data, we observe that the model can make accurate estimations as to adequate test set size. However, the question remains, does the suggested test set size differ significantly from the total number of test cases available? To address this, we now present data that compares the predicted adequate test set size (using the correct failure rates) and the total number of test cases available. This is provided in Table 5.

From Table 5 we observe that for three of the programs – grep, gzip, and make, almost all of the cases are required in order to reach a predicted effectiveness of 100%. But this is because some of the faults (faulty versions) only result in the failure of 1 or 2 test cases in our environment making them extremely hard to detect. In fact if only one test case fails on a faulty version, then to guarantee detection of that fault we must trivially select every test case. In stark contrast however, we note that the proposed model only requires a fraction of the test pool in order to reach a predicted effectiveness of 100% for all the programs of the Siemens suite. In fact this number is as low as 20.51% in the case of the program print_tokens2. This means that 79.49% of the test cases would not need to be executed thereby leading to significant savings both in terms of time and resources.

8.2. The estimation of failure rates

For the purposes of this paper we have assumed that the failure rates are readily available to be plugged into our model. In practice it is difficult, if not entirely impossible, to estimate failure rates with such accuracy. However, experiments and discussions have been presented in Section 6 to evaluate the impact of incorrect failure rate estimations on the proposed model. In practice such failure rates might be obtained from historical data or by means of programmer intuition. Also, as shown in Section 7 via initial experiments, it may not be necessary to correctly identify the number of faults in a potentially multiple-fault program. Nor is it necessary to estimate the program failure rate with complete accuracy. Additionally, once failure rates have been identified during an initial phase of testing, they can be refined and validated through successive iterations and can be used in conjunction with a model as is proposed in this paper.

8.3. *The proposed model and mutation testing*

An area where the work proposed herein can be readily applied is mutation testing. Mutation is a technique that assesses the fault detection effectiveness of a test set for a particular program by introducing syntactic code changes into a program and then observing if the test set is capable of detecting these changes (Budd, 1980; DeMillo et al., 1978; Do and Rothermel, 2006). Each first-order mutant is a different faulty version (with exactly one fault) of a source program and the same test set is to be executed against all the mutants. A mutant is killed if at least one of the test cases in the test set results in failure on that mutant. This is very similar to the model proposed in this paper. Each fault or faulty version is essentially a mutant. The objective is to estimate the number of test cases that can detect each fault or in this context kill each mutant, i.e., the size a mutation-adequate test set. This is a very good point of application for the proposed model as mutation testing can be expensive and so any model that can be used early on to estimate a good test set size is of significant inherent value. Without actually running test cases against each mutant, were estimates of the failure rates available (as per Section 8.2) we could have a reasonable idea of whether the current test set should be augmented or not. However, research into how an actual test set, in the context of mutation testing, may be constructed is beyond our immediate scope (which focuses on test set size) and we defer this to future work.

8.4. *Upper bounds on test set size*

Arguably, the work presented in this paper indirectly proposes some ideas towards the estimation of an upper bound on an adequate test set size. Assuming the correctness of the model, it is able to estimate the number of randomly selected test cases required to detect all the faults in a set, provided the failure rate information is known. However, in coming up with such a model we have also made several assumptions such as the fact that test cases will be selected at random, the cost and execution requirements of all the test cases are equivalent, failure rate information is known, etc. In light of such concerns, we have purposefully refrained from using words such as ‘upper bound’ or ‘optimum test set size,’ etc. Additionally, we assume that faults must have a non-zero failure rate as defined in this paper. But once again, the primary objective of this paper was not to construct a model that was necessarily optimal or superior to another. Rather the objective was to put forth the idea of fault failure rates as useful indicators of adequate test set size and to provide initial data on their evaluation. Now that the importance of fault failure rates in estimating adequate test set size has been established, another point that needs to be mentioned is that in no way do we claim that failure rates should be the only consideration when trying to identify adequate test set size. Instead, our results suggest that an accurate estimation of failure rates can lead to a reliable estimate of adequate test set size and therefore should be one of the factors to be kept in mind during test set construction.

8.5. *Threats to validity*

There are several threats to the validity of the ideas put forward in this paper, which include but are not limited to the following.

First, this experiment makes use of real fault failure rates that have been computed by actual test case executions against faulty programs. In real life such data may not be available. However, to assess the impact of an incorrect estimation of failure rates on the proposed model, we have performed experiments using perturbed failure rates generated with respect to three different rounding (perturbation) schemes in Section 6. Also, we discuss possible ways to obtain failure rates in Section 8.2. Second, we assume a random testing scenario in that test sets can be constructed by randomly selecting test cases from an available test pool. We also initially assume that each faulty version of a program has only one fault in it. However, this assumption is subsequently relaxed and a discussion and data based on programs with multiple faults is presented in Section 7.

An important threat to the internal validity that needs to be noted is that mutation and similar fault injection techniques were used in order to create faulty versions that could be studied. The primary motivator for this however was that faulty versions were not always readily available and this allowed our study to be more comprehensive. Also, mutation has been shown to be an effective approach to simulating realistic faults that can be used in software testing research to yield trustworthy results (Andrews et al., 2005). The mutation-based fault injection strategy has been actively used in other studies as well that are not necessarily related to the work presented herein. Do and Rothermel (2006) make use of this strategy in the context of assessing test case prioritization techniques and claim that mutation faults may provide a low-cost avenue to obtaining data sets on which statistically significant conclusions can be obtained. Therefore, while the mutation-based fault injection technique does represent a potential threat to validity, we feel it is a necessary and also a sound step to enlarge our data set.

An important threat to the external validity of our experiments is the choice of subject programs. We may not be able to generalize our results to every program and test set. However, the choice of subject programs is validated by the fact that the Siemens and space programs have been used in several studies (Jones and Harrold, 2005; Wong et al., 2010; Wong and Qi, 2009) and are quite well known and established as suitable benchmarks. Also, the programs in the Siemens suite allow us to experiment on programs of a relatively smaller size (in terms of the LOC) and the space, grep, gzip and make programs allow us to contrast that with programs of a relatively larger size. Our study, thus, includes a total of 11 programs each with varying faulty versions and test sets. This diversity allows us to have greater confidence in our results.

9. Related work

We now overview work that is related to this paper and the ideas expressed herein. Failure rates have been used extensively in the area of adaptive random testing (Cangussu et al., 2009; Chen et al., 2004; Kuo et al., 2007) which seeks to distribute test cases more evenly within the input space, as an enhanced form of random testing. As per the description in Chen et al. (2004), given a faulty program for an input domain D, let d, m and n denote the size, number of failure-causing inputs and number of test cases, respectively. The failure rate θ is then defined as m/d. The difference between the failure rate defined in Chen et al. (2004) and ours is that the authors of Chen et al. (2004) base their failure rate on the entire input domain whereas we based ours on the test pool, i.e., all the test cases that are available. Under the assumption that the test pool is constructed uniformly from the input domain (i.e., it is representative of the input domain), then the two failure rates should be equivalent as the ratio of failure-causing inputs to the entire input domain should be the same as the ratio of the failing tests in a test set to the total number of test cases in that set.

The concept of a software failure rate is also used extensively in the area of software reliability, but with a definition different from that used in this paper and in adaptive random testing. In the context of software reliability, the failure rate is defined as the rate at which failures occur in a certain time interval $(t, t+\Delta t)$ (Trivedi, 2002).

As per the discussion in Section 8.4, this paper can be viewed as proposing an upper bound on the number of test cases that need to be employed in a test set to make it fault detection adequate, using failure rate information. In Chen and Robert Merkel (2008) the authors examine the upper bound of debug testing effectiveness by making assumptions about the size, shape and orientation of failure patterns. Failure patterns describe typical ways in which inputs revealing program failure are distributed in the input domain – in many cases clustered in contiguous regions (Chen and Robert Merkel, 2008). Their results show that for failure patterns that are small in all dimensions of the input domain, even if the size, orientation, and shape of failure patterns are known but information about the location is not, no testing strategy can be designed to have an *F*-measure less than half the *F*-measure of random testing with replacement. We direct the interested reader to Chen and Robert Merkel (2008) for a formal definition of the term *F*-measure. In contrast, the work presented in this paper makes no assumptions about the failure patterns or how they are distributed, but instead makes a different set of assumptions in that the failure rate of a faulty version or program is known before-hand in order to make a predictive model. Also instead of assessing the effectiveness of a test set, we try to reverse the process by first identifying a desired level of effectiveness and then finding an appropriate test set size that corresponds to the desired effectiveness.

Various methods and techniques exist to reduce, if not minimize, the size of a test set. In Harrold et al. (1993) and Wong et al. (1998, 1999) the authors describe a methodology to minimize the size of a test set by eliminating redundant test cases while maintaining the same level of coverage. In Cangussu et al. (2009) the authors propose a process to reduce the number of test cases used in the performance evaluation of components. The process is based on sequential curve fittings from an incremental number of test cases until a minimal pre-specified residual error is achieved. Even though in general the test set minimization problem is NP-complete, greedy heuristics such as the one presented in Tallam and Gupta (2005) and other such methods (e.g., the one used by the χSuds tool suite (χSuds, 1998) developed by Telcordia Technologies, formerly Bellcore) are useful strategies that work towards test set optimization. The objective of such studies and the one discussed in this paper is similar in the sense that both try to identify a suitable test set size in order to achieve certain thresholds across criteria such as coverage or fault detection effectiveness (the latter is the criterion in this paper).

In our studies, a fault detection effectiveness of 100% is possible because all faults that do not result in any test case failure in our environment have been discarded. Such a step has also been taken in several other studies like Jones and Harrold (2005), Liu et al. (2006), Wong et al. (2010) and Wong and Qi (2009). Recent studies have also tried to identify strategies to debug programs with multiple faults (Jones et al., 2007; Liu and Han, 2006), while others have explored the validity of the independence assumption among multiple faults in the same program (Debroy and Wong, 2009; Knight and Leveson, 1986). This study also explores how multiple faults interact in the same program, but does so from the perspective of the failure rate that has been defined in this paper.

10. Conclusions and future work

This paper explores the link between adequate test set size (in terms of fault detection effectiveness) and fault failure rates. A probabilistic model is developed that makes use of failure rate information in order to predict an adequate test set size with respect to the fault detection. Via case studies performed on 11 sets of programs, the usefulness and validity of failure rates is established. The results are indicative of the fact that a precise estimation of fault failure rates can indeed help accurately predict an adequate test set size with respect to the fault detection criterion.

Analyses are conducted to examine if the predictive model generally over-predicts or under-predicts when compared to the effectiveness of actual test sets of the same size. By means of hypothesis tests we observe that the predictions are generally conservative in nature. We also perform an analysis on the relationship between prediction error and test set size. Observations support that as the size of the test set increases, the bias (because some test cases may detect more faults than others) involved is reduced and the absolute value of the error (observed effectiveness – predicted effectiveness) goes down. Additional studies are carried out to explore the effect of an incorrect estimation of failure rates on prediction quality, and to this effect three modes of perturbation of failure rates are used. Results indicate that the predictions even with the incorrect failure rates are still of encouraging quality. The relationship between multiple faults present in the same program is also discussed via the notion of fault and program failure rates.

Additional case studies using programs with multiple faults are currently ongoing. The work presented here can be applicable to regression testing, mutation testing, etc. and future work also includes performing case studies related to the application of our model to such areas.

References

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, May 2005, pp. 402–411.

Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1 (1 (January)), 11–33.

Blischke, W.R., Murthy, D.N.P. (Eds.), 2003. Case Studies in Reliability and Maintenance. John Wiley & Sons.

Budd, T.A., 1980. Mutation analysis of program test data. Ph.D. Dissertation, Yale University.

Cancellieri, A., Giorgi, A., 1994. Array preprocessor user manual. Technical Report IDS-RT94/052.

Cangussu, J., Cooper, K., Wong, W.E., 2009. A segment-based approach for the reduction of the number of test cases for performance evaluation of components. International Journal of Software Engineering and Knowledge Engineering 19 (4 (June)), 481–505.

Chen, T.Y., Leung, H., Mak, I.K., 2004. Adaptive random testing. In: Proceedings of the 9th Asian Computing Science Conference, Chiang Mai, Thailand, December 2004, pp. 320–329.

Chen, T.Y., Robert Merkel, 2008. An upper bound on software testing effectiveness. ACM Transactions on Software Engineering and Methodology 17 (3 (June)), 16.

Chen, T.Y., Yu, Y.T., 1996. On the expected number of failures detected by subdomain testing and random testing. IEEE Transactions on Software Engineering 22 (2 (February)), 109–119.

Cleve, H., Zeller, A., 2005. Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, May 2005, pp. 342–351.

Debroy, V., Wong, W.E., 2009. Insights on Fault Interference for Programs with Multiple Bugs. In: Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering (ISSRE), Mysuru, Karnataka, India, November 2009, pp. 165–174.

Debroy, V., Wong, W.E., 2010. Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST), Paris, France, April 2010, pp. 65–74.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. IEEE Computer 11 (4), 34–41.

Do, H., Rothermel, G., 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Transactions on Software Engineering 32 (9 (September)), 733–752.

Harrold, M.J., Gupta, R., Soffa, M.L., 1993. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2 (3 (July)), 270–285.

Harrold, M.J., 1991. The effects of optimizing transformations on data-flow adequate test sets. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Victoria, BC, Canada, October 1991, pp. 130–138.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, CA, USA, November 2005, pp. 273–283.

Jones, J.A., Bowring, J., Harrold, M.J., 2007. Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, UK, July 2007, pp. 16–26.

Knight, J.C., Leveson, N.G., 1986. An experimental evaluation of the assumption of independence in multi-version programming. IEEE Transactions on Software Engineering 12 (1 (January)), 96–109.

Knuth, D.E., 1997. The Art of Computer Programming. Fundamental Algorithms, vol. 1. Addison-Wesley.

Kuo, F.C., Chen, T.Y., Liu, H., Chan, W.K., 2007. Enhancing adaptive random testing in high dimensional input domains. In: Proceedings of the 2007 ACM Symposium on Applied Computing, Seoul, Korea, March 2007, pp. 1467–1472.

Leung, H., Tse, T.H., Chan, F.T., Chen, T.Y., 2000. Test case selection with and without replacement. Journal of Information Sciences 129 (1–4 (November)), 81–103.

Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P., 2006. Statistical debugging: a hypothesis testing-based approach. IEEE Transactions on Software Engineering 32 (10 (October)), 831–848.

Liu, C., Han, J., 2006. Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Oregon, November 2006, pp. 286–295.

Lyu, M.R. (Ed.), 1996. Handbook of Software Reliability Engineering. IEEE Computer Society Press.

Ott, R.L., 1993. An Introduction to Statistical Methods and Data Analysis, fourth ed. Duxbury Press, Wadsworth Publishing Company.

Renieris, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada, October 2003, pp. 30–39.

Reussner, R.H., Scmidt, H.W., Poernomo, I.H., 2003. Reliability prediction for component-based software architectures. Journal of Systems and Software 66, 241–252, 2002.

Rothermel, G., Harrold, M.J., 1998. Empirical studies of a safe regression test selection technique. IEEE Transactions on Software Engineering 24 (6 (June)), 401–419.

Siemens Suite, http://www-static.cc.gatech.edu/aristotle/Tools/subjects/, January 2007.

Tallam, S., Gupta, N., 2005. A concept analysis inspired greedy algorithm for test suite minimization. In: Proceedings of the 6th ACM Workshop on Program Analysis for Software Tools and Engineering, Lisbon, Portugal, September 2005, pp. 35–42.

Trivedi, K.S., 2002. Probability and Statistics with Reliability Queuing and Computer Science Applications. John Wiley & Sons.

Weyuker, E.J., 1986. Axiomatizing software test data adequacy. IEEE Transactions on Software Engineering 12 (12 (December)), 1128–1138.

Wong, W.E., Debroy, V., Choi, B., 2010. A family of code coverage-based heuristics for effective fault localization. Journal of Systems and Software 83 (2 (February)), 188–208.

Wong, W.E., Horgan, J.R., London, S., Mathur, A.P., 1998. Effect of test set minimization on fault detection effectiveness. Software-Practice and Experience 28 (4 (April)), 347–369.

Wong, W.E., Horgan, J.R., Mathur, A.P., Pasquini, A., 1999. Test set size minimization and fault detection effectiveness: a case study in a space application. Journal of Systems and Software 48 (2 (October)), 79–89.

Wong, W.E., Qi, Y., 2009. BP neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering 19 (4 (June)), 573–597.

Wong, W.E., Shi, Y., Qi, Y., Golden, R., 2008a. Using an RBF neural network to locate program bugs. In: Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), Seattle, USA, November 2008, pp. 27–38.

Wong, W.E., Wei, T., Qi, Y., Zhao, L., 2008b. A Crosstab-based statistical method for effective fault localization. In: Proceedings of the First International Conference on Software Testing, Verification and Validation, Lillehammer, Norway, April 2008, pp. 42–51.

χSuds User's Manual, 1998. Telcordia Technologies.

Vidroha Debroy received his BS in software engineering and his MS in computer science from the University of Texas at Dallas. He is currently a PhD student in computer science at UT-Dallas. His interests include software testing and fault localization, program debugging and automated and semi-automated ways to repair software faults. He is a student member of the IEEE and the ACM.

W. Eric Wong received his MS and PhD in computer science from Purdue University, West Lafayette, Indiana. He is currently an associate professor in the Department of Computer Science at the University of Texas at Dallas. Dr. Wong is a recipient of the Quality Assurance Special Achievement Award from Johnson Space Center, NASA (1997). Prior to joining UT-Dallas, he was with Telcordia Technologies (formerly Bell Communications Research a.k.a. Bellcore) as a Senior Research Scientist and as the project manager in charge of the initiative for Dependable Telecom Software Development. Dr. Wong's research focus is on the technology to help practitioners produce high quality software at low cost. In particular, he is doing research in the areas of software testing, debugging, safety, reliability, and metrics. He has very strong experience in applying his research results to real-life industry projects. Dr. Wong has received funding from such organizations as NSF, NASA, Avaya Research, Texas Instruments, and EDS/HP among others. He has published over 120 refereed papers in journals and conference/workshop proceedings. Dr. Wong has served as special issue guest editor for six journals and as general or program chair for many international conferences. He also serves as the Secretary of ACM SIGAPP and is on the Administrative Committee of the IEEE Reliability Society.