In *Department of Computer Science Technical Report, Texas Tech University*, Lubbock, Texas, USA, May 19 2010.

A Probabilistic Model for Effective Mutation Testing

Mohan Sridharan
Department of Computer Science
Texas Tech University
mohan.sridharan@ttu.edu

Akbar Siami Namin
Department of Computer Science
Texas Tech University
akbar.namin@ttu.edu

May 19, 2010

Abstract

Mutation is an expensive fault-based testing technique. Most of the cost of mutation is due to the compilation and execution costs associated with evaluating an enormous number of mutants with the entire pool of test cases. The test pool is typically augmented to account for mutants that remain unexposed after such an analysis. It can be argued that a test pool capable of detecting certain types of mutants is also capable of exposing other similar mutants. However, determining whether the test pool would expose the faulty behavior of a mutant, without actually evaluating the mutant with the test pool, is an undecidable problem.

There has been considerable research in the probabilistic reasoning community on identifying the important hypotheses among a set of candidate hypotheses. This paper applies such a Bayesian approach to mutation testing, in order to identify the *important* mutation operators whose mutants tend to remain unexposed after analysis by the existing test pool. We propose a novel approach that adapts stochastic sampling techniques to repeatedly select, examine and record the true behavior of a small proportion of generated mutants. Based on the performance of these selected mutants, the probabilities of the corresponding mutation operators are updated, and a proportionately suitable number of mutants of the operators are examined in the subsequent iterations. Over a few iterations, the process identifies the important mutation operators. Our exploratory case-study shows that the proposed approach has significant promise: it identifies over 90% of the important mutation operators by examining just 20% of the available mutants, while increasing the effectiveness of mutation testing by $\approx 5.0\%$.

1 Introduction

Mutation testing, first proposed by DeMillo [7] and Hamlet [10], is a fault-based testing technique to assess the adequacy of test suites based on their ability to detect synthetically injected faults (i.e. mutants) generated by well-defined mathematical transformations (i.e. mutation operators). It has been reported that mutants can act like real faults [2], making it feasible to measure the adequacy of test suites when there is a shortage in the number of available real faults. Despite its effectiveness, mutation testing is a computationally expensive technique, primarily due to the enormous number of mutants generated by applying a large number of mutation operators. Each generated mutant requires compilation, execution, and could possibly result in test case augmentation. However, not all mutants make significant contributions to the measurement of the adequacy of existing test suites. In particular, easy-to-detect mutants are likely to be exposed by the existing test suites, which may lead to the wrong conclusion that the existing test suites do not require any further augmentation.

Several mutation operators have been proposed, each representing a category of possible faults. Though each mutation operator is designed to generate a certain type of fault, there is no clear understanding of the relationship between mutants generated by a particular operator. To the best of our knowledge, the only related work is that by Offutt [17], where the author claimed that the test datasets that detect most of the simple faults are capable of detecting more complex faults. Further, Offutt claimed that this coupling effect was true of mutants i.e. the complex mutants are coupled to simple mutants in such a way that a test dataset that detects all simple mutants of a program will detect a large percentage of the more complex mutants.

Though the coupling effect may or may not hold true for faults and mutants, it exploits the idea of reducing the mutation cost by focusing only on single-entry mutated programs. In this paper, we make a more reasonable hypothesis that a test suite that is capable of detecting some of the mutants generated by a mutation operator is capable of detecting other similar mutants generated by the same operator. In other words, we can examine a small set of mutants and limit our attention to those mutation operators whose mutants remain unexposed by the existing test suites. We describe a novel Bayesian (i.e. probabilistic) formulation for mutation testing based on stochastic sampling techniques, in particular importance sampling. The proposed technique iteratively selects a small subset of mutants generated by each operator. In each iteration, the approach then directs its focus towards those mutation operators whose mutants remain unexposed (i.e. alive) in the previous iteration. The chief contributions of this paper are hence as follows:

1. It incorporates an *importance sampling* scheme to iteratively direct attention towards the *important* mutation operators whose application to a target program would require augmentation of the existing test suites.
2. It analyzes the sensitivity to prior distributions to show that the proposed approach is reasonably robust to different types of prior distributions.

The proposed approach was evaluated through a case-study on a specific subject program, and it shows promising results. The stochastic sampling approach increases the effectiveness of mutation testing by $\approx 5\%$ over the default approach of analyzing a random subset of mutants. Furthermore, the approach is able to identify $\geq 90\%$ of the interesting mutation operators by examining just 20% of the available mutants.

The remainder of the paper is organized as follows. Section 2 briefly reviews mutation testing, while Section 3 presents the sampling-based formulation for mutation testing. Section 4 describes data collection and the subject program used to evaluate the proposed approach. The experimental results and its ramifications are discussed in Section 4.3. Section 5 analyzes the effect of certain parameters involved in the proposed approach and presents the operators identified as being important by the proposed method. Section 6 includes a brief discussion of the threats to validity, followed by the conclusions in Section 7.

2 Related Work

Testing is a key component of software development, which accounts for 80% of development budget. Robust and reliable testing methods are hence a major requirement. One significant component of software testing methods is the choice of the adequacy criterion. Examples include code-based and specification-based coverage, where the adequacy of a given test suite is assessed in terms of its ability to cover certain elements of the code and specifications respectively. A test suite is adequate if it satisfies the given adequacy criterion. However, test practitioners are often more interested in assessing the adequacy of a given test suite in terms of its ability to detect faults in the given program.

Mutation testing is a fault-based testing technique that is used to assess the adequacy of a given test suite for detecting artificially injected faults [7, 10] that are automatically generated based on a set of mathematical transformations (i.e. mutation operators). A test suite *kills* a mutant if there is at least one test case in the test suite whose execution on the mutant and the original program (i.e. without the injected fault) results in different outputs. If no test case in the existing test suite is able to kill the mutant, the corresponding mutant is said to be *alive* i.e. running an existing test case on the mutant and the original program results in identical outputs. The existing test suite is then augmented by adding test cases in order to kill the unexposed mutants. This augmentation continues until the test practitioners judge that there does not exist any test case to detect the remaining alive mutants, which are then labeled *equivalent* mutants. The fraction of mutants killed by the existing test suite among the total number of non-equivalent mutants generated for the program represents the adequacy of the test suite, and it is denoted by MS (i.e. mutation score) or AM (i.e. adequacy measure). Analogous to the mutation score AM, we define mutation effectiveness score EM as the fraction of mutants of a program that remain alive after the application of a test suite, i.e. $EM = 1 - AM$.

It has been shown that mutants can act like real faults [2]. In addition, research has shown the effectiveness of mutation testing in revealing faults [9]. It has been also reported that mutation testing is more powerful than statement, branch and all-use coverage criteria [9]. However, the effectiveness of mutation testing depends greatly on the number of mutation operators designed for representing specific classes of faults. Several mutation operators have been proposed, each representing a class of faults [1]. The major challenge to the feasibility of mutation testing is the enormous number of mutants generated by the application of these mutation operators on a reasonable-sized program. Approaches have therefore been developed to reduce the computational cost by identifying meta-mutants [16], a selective set of mutants [15], or a sufficient set of mutation operators via multiple linear regression [20, 21, 22]. In addition to the number of mutants, identifying equivalent mutants is also a challenge. Since the problem is essentially undecidable, existing methods identify equivalent mutants through heuristic methods [3, 13], constraint satisfaction methods [14], and methods that measure the impact of mutants on invariants [19].

An intuitively appealing approach would be to focus more on operators whose mutants are more likely to remain unexposed with the existing test suites—we consider such mutation operators to be more *important*. This definition can be extended to equivalent mutants i.e. an operator is important if its mutants tend to be equivalent to the original program. As mentioned above, the assumption is that a test suite that exposes certain faults is likely to detect similar faults. In terms of mutation testing, an operator that represents a specific class of faults is likely to generate similar faults, most of which will either be exposed or remain alive when analyzed with the existing test suites. This paper presents a novel Bayesian approach that dynamically adapts the focus towards the important operators, thereby eliminating the computational cost associated with mutants that are likely to be exposed using existing test suites. We begin with a brief description of stochastic sampling.

2.1 Sampling Methods

In many practical domains, the exact inference of the true distribution of the variables characterizing the domain is often intractable. Several prediction tasks hence find it more efficient to use approximate inference methods that numerically *sample* from the underlying distribution [4]. Such sampling methods, also known as *Monte Carlo* methods, have been used extensively in research areas such as computer vision and robotics [8, 23].

Sampling is typically used in domains where there are multiple hypotheses representing the state of the system that is being analyzed. One popular example in robotics is to estimate the most likely locations of one or more mobile robots or others objects in a specific environment. Each "sample" represents a hypothesis, for instance the occurrence of a specific event, and it is associated with a probability measure that represents the likelihood that the corresponding hypothesis is true. Over a series of iterations, the probabilities are dynamically updated to converge to the most likely hypotheses. Each such iteration in the sampling process consists of three steps. First, each hypothesis is modified to account for any dynamic changes in the system being analyzed. Second, the probability of each hypothesis is updated based on data that is obtained by observing the system or running specific tests on the system. Third, the samples are replicated such that there are a relatively larger number of copies of samples with larger probabilities (after the current probability update). Given a fixed number of samples, over a series of iterations, sampling converges to analyzing only those hypotheses that show more evidence of being correct. There are several different forms of sampling (e.g., Gibbs sampling, rejection sampling, importance sampling and slice sampling) that can be used to analyze systems with different levels of complexity. See [4] for a description of the mathematical concepts underlying stochastic sampling techniques.

A sampling-based approach is well-suited for the mutation testing domain, where the goal is to direct attention towards the operators whose mutants are difficult to expose with the existing test suites. The next section describes our approach that achieves this goal reliably and efficiently.

3 Stochastic Analysis

We adapt the general sampling scheme to iteratively sample a subset of the existing mutants and focus increasingly on operators whose mutants are less likely to be exposed by the existing test suites. The proposed approach begins by selecting a small subset of mutants generated by each operator (Section 3.2) that are evaluated using the existing test suites. As with the general sampling scheme described in Section 2.1, the probabilities of the operators are updated based on the proportion of mutants left alive (Section 3.3). The updated probabilities are used to determine the number of mutants (of each operator) to be analyzed in the next iteration (Section 3.4). Over a few iterations, the process converges to looking at the important operators. Throughout this paper, the terms *mutants*, *samples*, and *mutant samples* are used interchangeably. We begin with a probabilistic formulation for mutation testing.

3.1 Probabilistic Formulation

Consider a set of mutation operators $\mathcal{OP} = \{\mu_0, \ldots, \mu_{N-1}\}$, a program $\mathcal{P}$ and a test suite $\mathcal{S}$ containing one or more test cases. Test suite $\mathcal{S}$ kills a mutant if at least one test case in $\mathcal{S}$ exposes the faulty behavior of the mutant. We define:

- $Nm_i(\mathcal{P})$: the number of mutants generated by μ_i on $\mathcal{P}$. $NM(\mathcal{P})$ is the total number of mutants of all operators.
- $Em_i(\mathcal{P}, \mathcal{S})$ is the mutation effectiveness score of test suite $\mathcal{S}$ for program $\mathcal{P}$ and operator μ_i. It is the fraction of mutants of $\mathcal{P}$ generated by μ_i that are left alive by $\mathcal{S}$. By extension, $Em_i^t(\mathcal{P}, \mathcal{S})$ is the mutation effectiveness score for test suite $\mathcal{S}$ in iteration t.
- $EM(\mathcal{P}, \mathcal{S})$ is the mutation effectiveness score for $\mathcal{P}$ and $\mathcal{S}$ over all operators. It is the fraction of the total number of mutants that are left alive by $\mathcal{S}$. By extension, $EM^t(\mathcal{P}, \mathcal{S})$ is the mutation effectiveness score of $\mathcal{S}$ (on program $\mathcal{P}$) in iteration t.

The goal is to identify the operators whose mutants are hard to expose using the existing test suites, so that the test suites can be suitably augmented. In addition, the EM is to be maximized and the number of mutants examined is to be minimized. In order to do so, a probability measure is associated with each operator:

$$< \mu_i, p_i > : \; p_i \propto \text{importance of operator } \mu_i \tag{1}$$

where the probability p_i of operator μ_i would be low if the test suite $\mathcal{S}$ kills a large proportion of the mutants generated by μ_i on $\mathcal{P}$. Since the mutants are generated by applying mutation operators on a specific program, and are evaluated using specific test suites, references to $\mathcal{P}$, $\mathcal{S}$ are dropped—for instance Nm_i is used instead of $Nm_i(\mathcal{P})$.

3.2 Prior Distributions

The first iteration can start with a uniform distribution i.e. it can choose the same number of mutants for each operator that generates a non-zero number of mutants on a specific program. Another option is to make each operator's probability proportional to the relative number of its existing mutants.

$$numMutantSamps_i^0 \simeq \begin{cases} c & \text{uniform} \\ \propto \frac{Nm_i}{NM} & \text{proportional} \end{cases} \tag{2}$$

where $numMutantSamps_i^0$ is the number of mutants of operator μ_i selected and examined in the first iteration and c is an arbitrary integer. The total number of mutants analyzed in the first iteration is hence a fixed number or a fraction of the total number of mutants generated by all operators. The performance of these two schemes is compared empirically in Section 4.

3.3 Probability Update

In each iteration, a small subset of the mutants of each operator are evaluated using the existing test suite(s). The probability of each operator is updated based on the proportion of mutants left alive by the existing test suite(s):

$$\begin{aligned} p_i^t &= p_i^{t-1} + \delta p_i^t \frac{k}{totalMutantSamps^t} \\ \delta p_i^t &= -1.0 + 2.0 \frac{numAlive_i^t}{numMutantSamps_i^t} \quad : \in [-1.0, 1.0] \\ totalMutantSamps^t &= \sum_{i=0}^{N-1} numMutantSamps_i^t \end{aligned} \tag{3}$$

where $numMutantSamps_i^t$ is the number of mutants of operator μ_i examined in iteration t, while $totalMutantSamps^t$ is the total number of mutants analyzed in this iteration. The probability of μ_i in iteration t is the sum of the probability in iteration $t-1$ (i.e. p_i^{t-1}) and an incremental factor (δp_i^t). This factor is based on the fraction of chosen mutants of operator μ_i that are not exposed in the current iteration ($\frac{numAlive_i^t}{numMutantSamps_i^t}$). It is also inversely proportional to the number of mutants examined in iteration t. The parameter k (typically set to 1) provides an additional means of controlling the value being added or subtracted. After the update, the operator probabilities are normalized so that they sum to one[1]:

$$p_i^t = \frac{p_i^t}{\sum_j p_j^t} \tag{4}$$

The updated probabilities represent the current estimate of the importance of the operators, which can be used to identify the important operators while examining a small subset of the mutants, as described below.

[1]This is a necessary condition for a probability measure.

3.4 Adaptive Sampling

Sampling is typically used to track a set of hypotheses and iteratively identify the most likely hypotheses. As described in Section 2.1, a larger number of copies are to be created of samples with high relative probability. In other words, the task is to examine a proportionately larger number of mutants of the operators whose mutants are more likely to remain unexposed. The updated and normalized probabilities (Equation 4) are used to determine the number of mutants of each operator that will be analyzed in the next iteration. In order to do so, we modify the Stochastic Universal Sampling method [24], as shown in Algorithm 1.

Algorithm 1 Modified sampling algorithm.

```
Require: Current (iteration $t$) set of ¡operator, probability¿ vectors of the form: $\{<\mu_i, p_i^t>: i \in [0, N-1]\}$ sorted in decreasing order of $p_i^t$. MaxNum: Maximum number of mutants to be analyzed.
1:  $Count_i^{t+1} = 0 : i \in [0, N-1]$, $\pi_0 = p_0^t$.
    {Set up the cumulative probability distribution $\pi_i$.}
2:  for $i = 1$ to $N-1$ do
3:      $\pi_i = \pi_{i-1} + p_i^t$.
4:  end for
5:  $r_0 \sim U\,]0, 1/MaxNum\,]$, iterator $i = 0$
    {Compute the sample counts for each operator.}
6:  for $j = 0$ to $MaxNum - 1$ do
7:      while $r_j > \pi_i$ do
8:          $i = i + 1$
9:      end while
10:     $Count_i^{t+1} = Count_i^{t+1} + 1$
11:     $r_{j+1} = r_j + 1/MaxNum$
12: end for
13: Return $\{Count_i^{t+1}\} : i \in [0, N-1]$
```

The algorithm requires as input the vectors $\{<\mu_i, p_i^t>: i \in [0, N-1]\}$ that represent the operators and their probabilities after the iteration t (Equation 4). The vectors are sorted in decreasing order of probability.

The algorithm begins by initializing the number of mutants of each operator μ_i that will be analyzed in the next iteration ($Count_i^{t+1} = 0$, line 1 of Algorithm 1). The algorithm then sets up the cumulative probability distribution (lines 2–4) that represents the contribution of all the operators. In addition, a random real number is chosen from a uniform distribution ($r_0 \sim U\,]0, 1/MaxNum\,]$) based on the total number of mutant samples that will be analyzed in the next iteration ($MaxNum$ in line 5). The next step (lines 6–12) forms the core of the sampling process. Intuitively, the idea is to use the relative probability of each operator to determine the number of its mutants to be examined in the next iteration. With the operator vectors arranged in decreasing order of probability and the selection of r_0 as shown in line 5, the count of samples corresponding to the most probable mutation operator is incremented first. This sample count for the most important operator continues to be incremented (line 10) until this operator's relative contribution to the cumulative probability distribution is taken into account. Then the selected random number is suitably modified (line 11) and the focus shifts to the operator that made the second largest contribution to the cumulative probability distribution (lines 7–9). The process is terminated when the desired count of samples ($MaxNum$) is reached. Sampling therefore provides an elegant means of selecting more mutants corresponding to operators whose probabilities are higher. In other words, the number of mutants of an operator to be examined in the subsequent iteration depends on the current estimate of how well the existing test suites can expose the operator's mutants.

4 Case Study

This paper presents the results of an initial exploratory case study, wherein the proposed stochastic method was applied to a subject program developed in the C language. The results show the validity of the initial hypothesis that mutants from the same family display similar characteristics.

Our chosen subject program is `schedule2`, one of the seven program from the Siemens set [11], which was obtained from the Subject Infrastructure Repository (SIR) at University of Nebraska-Lincoln. The program is a priority scheduler application that has 263 net lines of code (excluding comments and blank lines) along with 2710 test cases. Most of the test cases were developed by Siemens researchers [11] and augmented by Rothermel et al. [18].

4.1 Data Collection

We used Proteum [6], a mutant generator for C language, which implements 108 mutation operators based on the specification designed by Agrawal et al. [1]. The comprehensive set of mutation operators implemented by Proteum was the major criterion for its selection instead of other mutation tools such as MuJava [12] for Java programs. Proteum generated 6552 mutants from which 2000 mutants were selected randomly for the case study. To avoid any bias towards particular operators, the sampling procedure randomly selected the same proportion of mutants for each operator. The selection of a subset of mutants was necessary in order to make the case study feasible. Performing the computation over all mutants, including operations such as generation, compilation and evaluation of the mutants, would have taken a very long time.

We generated one test suite of each size from 1 to 50 test cases, choosing the test cases randomly from the test pool. Generating more than one test suite for each size would have been computationally expensive because each test suite would be involved in a separate iterative analysis to determine the most important operators.

The entire set of test cases was executed for all 2000 mutants as well as for the original version of the program to determine the list of mutants killed by each test case. Mutants that were not killed by any test case in the test pool were considered equivalent. The procedure resulted in 1497 non-equivalent mutants, which were used for subsequent computation. The equivalent mutants were computed using only the existing test pool. The relatively large test pool for `schedule2` led us to believe that the test pool was thorough enough to kill all non-equivalent mutants. Below, the terms "all mutants" or "existing mutants" refer to this set of non-equivalent mutants.

As described in Equation 2 we pursued two different schemes (i.e. uniform and proportional) for generating the prior distribution of operator probabilities for each test suite. In the first scheme, the first iteration involved the random selection of the same small number of mutants generated by each operator (i.e. a uniform probability distribution)—we set $c = 2$ in Equation 2. Only those operators whose application resulted in a non-zero number of mutants were considered. In the second scheme, the initial probability of each operator was proportional to the number of mutants generated by the operator i.e. $p_i^0 = Nm_i/NM$ for operator μ_i. The number of mutants to be sampled in the first iteration is therefore $p_i^0 * Nm_i$ for operator μ_i.

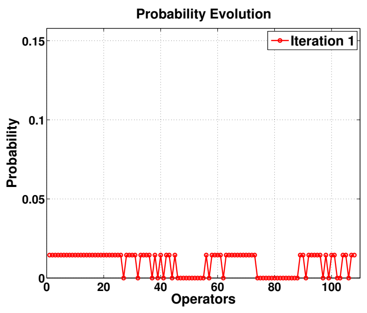
For either scheme of generating the prior distribution, the subsequent steps in each iteration were identical. For each test suite $\mathcal{S}$ under consideration, we computed the number of selected mutants killed by the test suite and the fraction (%) of selected mutants that were not killed by the test suite (i.e. $Em_i^t : i \in [0, N-1]$). This data was used to update the probability of the operators and compute the number of samples to be examined in the next iteration. We performed $5-6$ such iterations, though (as we show below) the desired results were typically obtained in the first $3-4$ iterations. In order to evaluate the performance of our method, we also computed EM, Em_i for each test suite $\mathcal{S}$ by examining all non-equivalent mutants of each mutation operator.

4.2 Data Analysis

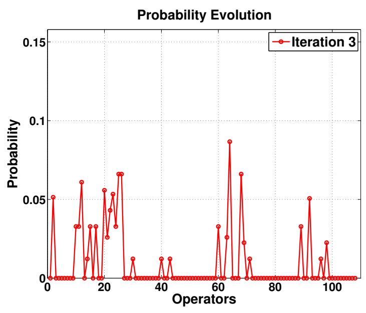
In order to evaluate the ability of the proposed approach to identify the operators whose mutants are left unexposed by the existing test suites, we applied different test suites on mutants of the program `schedule2`.

Figures 1(a)–1(f) show the evolution of operator probabilities over a few iterations. These results correspond to the situation when a test suite of size 35, i.e. a test suite with 35 test cases, is applied on the target program. Figures 1(a)–1(c) show the evolution of operator probabilities in the first, third and sixth iterations when the initial distribution of operator probabilities is uniform over the set of operators with a non-zero number of mutants. Figures 1(d)–1(f) show similar graphs for the case where the initial operator probabilities are proportional to the relative number of existing mutants of the mutation operators. The figures show that the proposed approach takes a small number of iterations to quickly converge on a set of important operators whose mutants are difficult to expose with the corresponding test suite. With either scheme for the assignment of initial operator probabilities, the most important operators are approximately the same. In addition, the set of important operators identified with either scheme is a small subset of the available mutation operators. Similar results were obtained for other test-suite sizes (from 1 to 50) but we do not show them here because of space limitations.

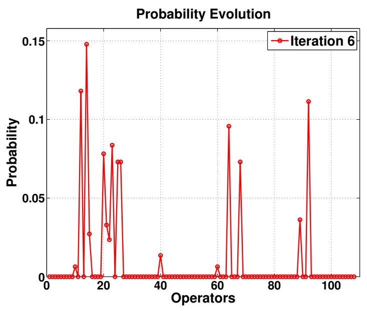
One significant consequence of the proposed approach is that the important operators are identified in a small number of iterations—we terminate the iterations when the operator probabilities do not change substantially between two iterations. For the test suite sizes used in our experiments, the maximum number of iterations is never more than six, and a large fraction of the trials terminate within $3-4$ iterations.



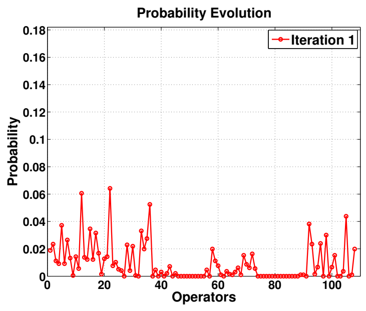
(a) Uniform: First Iteration.



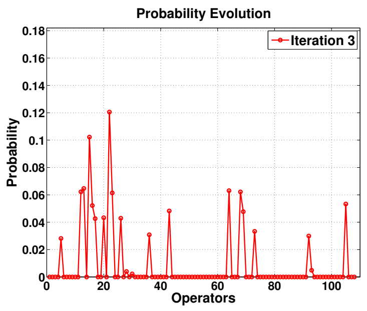
(b) Uniform: Intermediate Iteration.



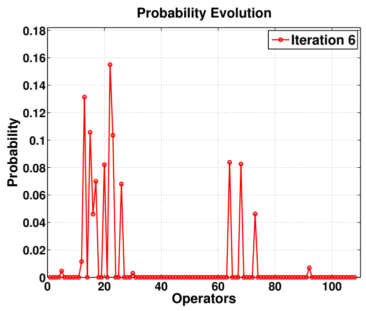
(c) Uniform: Final Iteration.



(d) Proportional: First Iteration.



(e) Proportional: Intermediate Iteration.

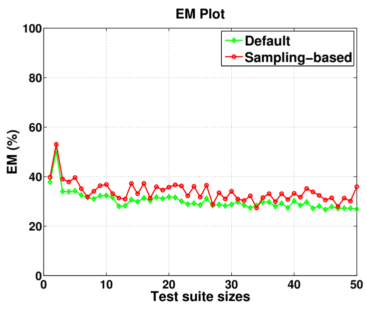


(f) Proportional: Final Iteration.

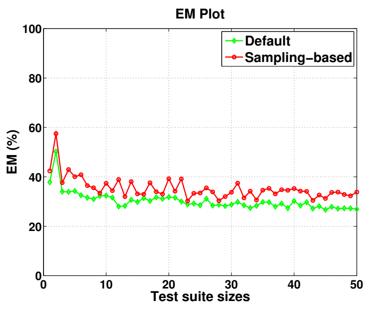
Figure 1: Evolution of operator probabilities for `schedule2` and test suite size 35, starting with an initial distribution that is: (a)-(c) uniform for all operators with a non-zero number of mutants; and (d)-(f) proportional to the relative number of existing mutants of each operator.

4.3 Overall Evaluation

Though the proposed approach converges in a small number of iterations, it is essential to determine if the mutation operators considered to be important are actually important. In addition, we need to quantify the overall performance of the approach over



(a) Uniform (initial) distribution



(b) Proportional (initial) distribution

Figure 2: Mutation Effectiveness (EM) as a function of test-suite sizes for `schedule2` for: (a) uniform initial probability distribution; and (b) initial operator probabilities proportional to the number of available mutants.

different test suite sizes. In order to do so, we compared the proposed method against the default method of examining all non-equivalent mutants.

Figures 2(a)–2(b) show the performance of the proposed approach as a function of the test suite sizes, for the two different schemes for the assignment of initial operator probabilities. We use mutation effectiveness score (EM) as the performance measure, i.e. the fraction of existing mutants that are left alive by the corresponding test suite. Given that the goal is to identify the operators whose mutants are difficult to kill, a larger EM represents a better performance. The figures show that the sampling-based approach performs better than the default approach. The performance is all the more significant because the proposed approach only examines a small fraction of the existing mutants ($\approx 20\%$).

Prior	EM (%)	δEM(%)
Uniform	34.05	3.79
Proportional	35.28	5.02

Table 1: Improvement in EM with different initial operator probability distributions.

Table 1 quantifies the EM performance. It shows the improvement in EM obtained with the different initial distributions of operator probabilities, as compared with the default approach. The result of a paired t-test showed that the numbers represented in the table are statistically significant.

Finally, we evaluated the ability of the proposed approach to detect important operators. In order to do so, the Em_i of each operator μ_i, i.e. the effectiveness measure computed by analyzing all its mutants with a particular test suite $\mathcal{S}$, were sorted in descending order to obtain the *ground-truth* list of the operators in decreasing order of importance. The operator probabilities computed during the sampling iterations were also sorted to create the *observed* list of important operators. We then defined the *operator overlap* measure as given below:

$$OpOverlap(\mathcal{P}, \mathcal{S}, T) = Overlap(Gt(\mathcal{P}, \mathcal{S}), Obs(\mathcal{P}, \mathcal{S})) \tag{5}$$

which computes the fraction of top $T\%$ operators in the ground-truth list (Gt) that also exist among the top $T\%$ of the observed list (Obs), for a particular $\mathcal{P}$ and $\mathcal{S}$. Table 2 summarizes the results for `schedule2` over the range of test suite sizes $(1 - 50)$. The two rows depict results corresponding to the two different schemes for setting the initial operator probability distributions (Equation 2).

Prior	Operator Overlap%			
	Min	Max	Average	Dynamic
Uniform	0.68	0.91	0.77 ± 0.04	0.89
Proportional	0.66	0.92	0.78 ± 0.05	0.91

Table 2: Operator overlap for different initial operator probability assignment schemes.

The "Max", "Min" and "Average" columns of Table 2 report the maximum, minimum and average OpOverlap (operator overlap) obtained with the proposed approach over the different test suite sizes. For these results, the top 25% of the operators in the ground-truth and observed lists were compared (i.e. $T = 25$). However, in some instances, the top 25% operators in the ground-truth list have low mutation effectiveness values and those in the observed list do not have significantly higher probabilities than the less important operators. Experiments were therefore conducted such that only the operators in the ground-truth list with Em_i above a threshold were compared with the operators in the observed list with probabilities significantly higher than the mean—the value of T was hence set dynamically for each trial. The corresponding results are summarized in the columns labeled "Dynamic" in Table 2. We observe that $\approx 90\%$ of the truly important operators are detected using the sampling-based approach. Furthermore, such high OpOverlap values are obtained in ≈ 4 iterations on average. Based on all the empirical evaluations described above, we conclude that the proposed technique identifies the important mutation operators for a range of test suite sizes by analyzing a small subset of the available mutants over a small number of iterations.

5 Discussion

In this section, we discuss two important issues related to the performance of the stochastic sampling technique. First, we discuss the effect of the parameters of Algorithm 1 on the performance of the method. Next, we analyze the important operators identified by the proposed technique.

5.1 Effects of Sampling Parameters

The proposed stochastic technique for identifying important mutation operators has some parameters, which can be used to improve the performance of the proposed technique. Among these parameters are: the selection of mutants from the existing set with/without replacement, the number of iterations (i.e. $NumIter$), and the number of mutants selected in each iteration (i.e. $MaxNum$ in Algorithm 1).

Unlike a server where the status of nodes may change, or a mobile robot whose position and orientation may change over time, mutation testing is essentially a *stationary* domain. The mutants will either be killed or remain alive when tested with the given test suites. No additional information is gained by examining a mutant more than once with the same test suites. In our work, mutants are therefore sampled without replacement i.e. mutants that have been examined once are not included in subsequent iterations. As a result, some operators may not have any un-tested mutants after a few iterations. Any such operators are not included in the sorted list or the resampling procedure during subsequent iterations of Algorithm 1. Their probabilities will then change only as a result of the normalization of operator probabilities in subsequent iterations (Equation 4). An interesting situation arises when the desired sample count ($MaxNum$) exceeds the number of existing un-tested mutants. One possible solution that requires further analysis is to distribute the excess "share" of such mutation operators among other operators that still have unexamined mutants. Such situations did not arise in the experimental results reported in this paper.

In most cases, the proposed stochastic sampling approach converges to the important operators within a small number of iterations ($NumIter \approx 4$) and subsequent iterations do not cause a significant change in the relative probabilities of the operators. However, larger programs with a substantially larger number of mutants may require a different number of iterations. Instead of the current heuristic approach, the termination of sampling can be automated by continuing the process until there is no substantial *information gain* between two successive iterations. This requirement can be readily formulated as a information-theoretic problem using the *entropy* of the probability distribution over the mutation operators as a measure of information gain [5]—the entropy would ideally decrease substantially between two successive iterations.

Ideally, the number of mutants examined in each iteration (i.e. $MaxNum$) should be as small as possible to avoid unnecessary computation. Though we currently examine a fixed number of mutants in each iteration, the goal is to determine a bound on the minimum number of samples required to approximate the underlying probability distribution of operator importance. Existing research in robotics uses approximated quantiles of a chi-square distribution to compute such a bound that guarantees that the distance between the sampling-based estimate and the underlying distribution would be within specified error bounds [8]. Further research is required to use such methods to automate the selection of the number of mutants examined in each iteration of the stochastic sampling process.

5.2 Analysis of Important Operators

We also analyzed the implications of the operators identified as being important, and evaluated the sensitivity of the proposed stochastic model to different prior distributions. In order to do so, we looked at the most important operators identified when the two different prior distributions (i.e. uniform and proportional—Equation 2) are used. There were 50 mutation operators that did not generate any mutants, and these operators were soon excluded from the probability updates and sampling procedure. The remaining 58 operators participated in all the iterations of the sampling procedure. Table 3 represents the top ten mutation operators whose mutants were identified as being the most difficult to expose with the existing test suites at the end of five iterations. Table 3 also shows the number of test suites (among the 50 available suites) for which these operators were considered important. These results show that there is significant overlap between the results obtained with the different initial probability assignment schemes. In addition, it shows that the proposed stochastic procedure is reasonably robust to the choice of the prior distributions.

Operator	Uniform	Proportional
-u-OCOR	50	50
Cast Operator by cast Operator		
-u-OLBN	50	44
Logical Operator by Bitwise Operator		
-I-RetStaRep	50	39
Replace Return Statement		
-u-VDTR	39	50
Domain Traps		
-u-OASN	40	39
Replace Arithmetic Operator by Shift Operator		
-I-IndVarRepReq	24	50
Replace Non Interface Variables by Required Constants		
-u-OLSN	47	27
Logical Operator by Shift Operator		
-I-IndVarRepGlo	41	28
Replace Non Interface Variables by Global Variables		
-u-OLLN	42	26
Logical Operator Mutation		
-u-OABN	40	23
Arithmetic by Bitwise Operator		

Table 3: Top ten mutation operators considered important for the two different schemes for initial operator probability assignment. Numbers in the second and third columns represent the count of test suites for which these operators are considered important.

In terms of software testing, the results reported in Table 3 state that replacing *logical operators*, *cast operators* or *return* statements produce hard-to-detect mutants. These conclusions make sense from a software testing perspective. More importantly, the results show that the proposed stochastic sampling-based approach significantly improves the effectiveness, reliability and efficiency of mutation testing.

6 Threats to Validity

Threats to external validity include the use of only one C program (`schedule2`). The results reported in this paper for this program supports our initial assumption that a test suite that detects a certain type of fault is likely to detect other faults of the same type. Larger programs may have more complex structure and possibly more mutants, which may lead to different results. Object-oriented programs need to be investigated, since they contain features that may lead to results different from those reported here. In particular, using a mutant generator for object-oriented programs (such as MuJava [12]) that implements class mutation operators may behave differently.

Threats to construct validity include the random selection of 2000 mutants from the program. This was necessary to make the study feasible. Performing the computation over all possible mutants would have taken a very long time. However, bias for a specific operator was avoided by randomly selecting the same proportion of mutants for each operator. Finally, threats to internal validity include the correctness of the mutation tool, scripts, and data collection processes. We developed our own C script for implementing importance sampling. Each author (separately) monitored the results at intermediate steps of the processes to ensure correctness.

7 Conclusion and Future Work

In this paper, we reported the results of an exploratory case study on a Bayesian formulation for mutation testing. The goal is to determine the important mutation operators i.e. operators whose mutants cannot be detected with existing test suites and need further augmentation of the existing test suites. We viewed the problem as an instance of stochastic sampling, where mutants of each operator that shows a likelihood of being important are examined further over a set of iterations. The focus is incrementally directed towards the important mutation operators. The preliminary empirical evaluation conducted on a representative subject program (`schedule2`) shows that the proposed approach can detect over 90% of the important operators, while examining only 20% of the mutants examined by the default approach that analyzes all non-equivalent mutants. In addition, there is an improvement of $\approx 5\%$ in the mutation effectiveness scores. Furthermore, the results support our initial assumption that a test suite capable of detecting a certain category of faults is likely to detect other similar faults.

The work reported in this paper opens up a new and exciting direction of research. Though we have only applied the technique so far to a single program (`schedule2`), the results are significantly promising. In the future, we will conduct a more extensive case study with several other (and larger) programs, beginning with an empirical evaluation based on all seven Siemens programs. In addition, we will investigate the use of other mutation tools for object-oriented programming languages (e.g. MuJava [12] for Java). We aim to incorporate the proposed stochastic technique in existing mutation tools in order to make mutation testing more effective and yet computationally less expensive. The long-term goal is to construct Bayesian formulations for a wide variety of challenges that are of interest to the software testing community.

References

[1] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Department of Computer Science, Purdue University, April 2006.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *The International Conference on Software Engineering*, St. Louis, USA, May 2005.

[3] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical report, Yale University, Department of Computer Science, 1979.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, New York, 2008.

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Publishing House, 1991.

[6] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *The Conference on Performability in Computing Systems*, pages 79–95, New Brunswick, USA, July 1996.

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

[8] D. Fox. Adapting the Sample Size in Particle Filters through KLD-Sampling. *International Journal of Robotics Research*, 2003.

[9] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing. *Systems and Software*, 1997.

[10] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *The International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[12] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava : An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[13] A. Offutt and W. Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4(3):131–154, September 1994.

[14] A. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *The Annual Conference on Computer Assurance*, pages 224 – 236, Gaithersburg, USA, June 1996.

[15] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[16] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, USA, October 2000.

[17] J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(3):3 – 18, 1992.

[18] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *The International Conference on Software Maintenance*, pages 34–43, Washington DC, USA, November 1998.

[19] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *The International Symposium on Software Testing and Analysis*, pages 69 – 80, Chicago, USA, July 2009.

[20] A. Siami Namin and J. Andrews. Finding sufficient mutation operators via variable reduction. In *Mutation 2006*, Raleigh, USA, November 2006.

[21] A. Siami Namin and J. Andrews. On sufficiency of mutants. In *The International Conference on Software Engineering*, pages 73 – 74, Minneapolis, USA, 2007.

[22] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *The International Conference on Software Engineering*, pages 351–360, Leipzig, Germany, May 2008.

[23] M. Sridharan, G. Kuhlmann, and P. Stone. Practical Vision-Based Monte Carlo Localization on a Legged Robot. In *The International Conference on Robotics and Automation*, April 2005.

[24] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, USA, 2005.