2010 IEEE 21st International Symposium on Software Reliability Engineering

Prioritizing Mutation Operators based on Importance Sampling

Mohan Sridharan
Stochastic Estimation and Autonomous Robotics Lab
Computer Science Department
Texas Tech University
Lubbock, TX, USA
Email: mohan.sridharan@ttu.edu

Akbar Siami Namin
Advanced Empirical Software Testing and Analysis Group
Computer Science Department
Texas Tech University
Lubbock, TX, USA
Email: akbar.namin@ttu.edu

***Abstract*—Mutation testing is a fault-based testing technique for measuring the adequacy of a test suite. Test suites are assigned scores based on their ability to expose synthetic faults (i.e., mutants) generated by a range of well-defined mathematical operators. The test suites can then be augmented to expose the mutants that remain undetected and are not semantically equivalent to the original code. However, the mutation score can be increased superfluously by mutants that are easy to expose. In addition, it is infeasible to examine all the mutants generated by a large set of mutation operators. Existing approaches have therefore focused on determining the sufficient set of mutation operators and the set of equivalent mutants. Instead, this paper proposes a novel Bayesian approach that prioritizes operators whose mutants are likely to remain unexposed by the existing test suites. Probabilistic sampling methods are adapted to iteratively examine a subset of the available mutants and direct focus towards the more informative operators. Experimental results show that the proposed approach identifies more than 90% of the important operators by examining $\leq 20\%$ of the available mutants, and causes a 6% increase in the importance measure of the selected mutants.**

***Keywords*-Mutation Testing; Testing Effectiveness; Importance Sampling; Bayesian Reasoning.**

I. Introduction

The adequacy of test suites used for program analysis is typically measured using specific criteria. For instance, code coverage measures the ability of a test suite to cover certain elements of the program. Black box testing, on the other hand, defines a specification-based coverage adequacy measure for a test suite. In these cases, the test suite is adequate for covering specific elements of the code or for the given specification. However, test engineers are typically interested in measuring the adequacy of a test suite in terms of its ability to find possible faults in the program.

Mutation testing is a fault-based testing technique that automatically generates synthetic faulty variants (i.e., mutants) of the original code using a set of mathematically well-defined mutation operators. The existing test suite is run on all mutants to compute the number of mutants that are exposed (i.e., killed) or left undetected (i.e., alive). The test suite is then incrementally augmented with test cases to detect the unexposed mutants, until the remaining alive mutants are judged to be semantically equivalent to the original code. The test suite's *effectiveness* is defined as the ratio of the number of mutants detected to the total number of non-equivalent mutants, and this is denoted by *mutation score* (MS) or *mutation adequacy ratio* (AM). This paper defines a test suite's *mutation importance measure* (IM) as the ratio of the number of undetected mutants to the number of non-equivalent mutants (i.e., $IM = 1 - AM$). A key challenge to mutation testing is the infeasibility of examining all mutants generated by applying a large number of mutation operators on the target program. Researchers have therefore focused on computing the sufficient set of mutation operators or the set of equivalent mutants.

Each mutation operator represents a type of fault. Depending on the program and test suite, some faults may be exposed while others may require augmentation of the test suite. This paper hypothesizes that a test suite that exposes certain faults is likely to detect similar faults. Similarly, a mutation operator, most of whose mutants have been caught by a test suite, is likely to produce mutants that can be exposed using the test suite. Based on this hypothesis, this paper describes an approach that directs attention towards the mutation operators whose application on a given program is likely to produce mutants that would not be exposed by a given test suite. These operators are considered "*high priority*" or "*important*" because they need to be identified quickly to augment the test suite appropriately. The major contributions of this work are hence as follows:

- An importance sampling-based probabilistic approach to iteratively direct focus towards operators whose mutants tend not to be killed by the existing test suites.
- Information-theoretic and adaptive sampling-based constraints to achieve automatic and efficient mutation testing on different programs and test suites.

This paper begins with a review of the technical background in Section II. Section III presents the proposed approach. Sections IV-V use case studies on subject programs to show that this approach identifies $\geq 90\%$ of the important operators by examining $\leq 20\%$ of the mutants, and increases the IM by 6% over the approach of examining all non-equivalent mutants. Section VI discusses the threats to validity and Section VII presents the conclusions.

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.16

II. Related Work

This section reviews some representative approaches for mutation testing, operator sufficiency and equivalent mutants, followed by a brief overview of stochastic sampling.

A. Mutation Testing

Mutation testing is the process of assessing test suites by measuring their ability to detect synthetic faults that are automatically generated by applying well-defined mathematical transformations (i.e., mutation operators) on the target program [1], [2]. A test suite is augmented to expose the undetected mutants, and it is said to be adequate if it kills all non-equivalent mutants i.e., its $AM = 1$. Proteum [3] and MuJava [4] are tools for mutating C and Java programs. Mutation testing has been reported to be more powerful than statement, branch and all-use coverage criteria [5]. Research has shown that mutants can act like real faults [6], and mutants have been used for assessing the effectiveness of different software testing schemes. However, performance depends on the type of operators designed for specific classes of faults. A large number of mutation operators have been proposed to represent different classes of faults [7]. Despite years of research, computation and analysis of all the mutants remains a major challenge to the feasibility of mutation testing. Approaches such as meta-mutants [8], selective mutation [9] and sufficient mutation operator identification [10], [11] aim to reduce the cost of mutation. For instance, Siami Namin et al. [10] proposed a cost-based regression procedure that modeled the number of mutants generated by each operator as the cost for that operator, thereby generating 92% fewer mutants. Identifying equivalent mutants is also a challenge. It is undecidable to predict whether an alive mutant can be killed by adding a test case. Proposed solutions include heuristic methods [12] and methods based on constraint satisfaction [13]. Research has also been done on localizing the effect of equivalent mutants [14], and measuring the impact of mutants [15]—mutants with higher impact are unlikely to be equivalent.

B. Stochastic Sampling

In many practical domains, computing the true distribution of the variables characterizing the domain is intractable. Several prediction tasks therefore use approximate inference methods that numerically *sample* from the true (unknown) underlying distribution to estimate it [16]. Such *Monte Carlo* methods have been used extensively in fields such as computer vision and robotics [17], [18]. Sampling is applicable to domains where it is necessary to track the likelihood of multiple hypotheses about the state of the system being analyzed. Each "sample" is an instance of a hypothesis (e.g., the occurrence of a type of fault) and it is associated with a probability that represents the likelihood that the hypothesis is true. Each iteration of sampling consists of three steps: (1) each hypothesis is modified to account for dynamic changes in the system; (2) the probability of each hypothesis is updated by examining some samples of that hypothesis; and (3) a larger number of samples are drawn from hypotheses with a larger (relative) probability, to be analyzed in the next iteration. Over a few iterations, sampling converges to the true probability distribution over the set of hypotheses, thereby identifying hypotheses that are more likely to be true. Sampling methods [16] are hence well-suited to model the inherent stochasticity of software testing—each program displays certain unique characteristics during testing.

III. Probabilistic Formulation

This section describes the proposed stochastic approach for mutation testing that automatically adapts to the characteristics of any given program and test suite. It iteratively examines a small set of mutants of the program to prioritize operators whose mutants are more likely to remain unexposed by the test suite. As a result, the test suite is augmented efficiently and the program is analyzed reliably.

A. 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. $\mathcal{S}$ kills a mutant if at least one test case in $\mathcal{S}$ exposes the faulty behavior of the mutant. The following terms are used in this paper:

- $Nm_i(\mathcal{P})$: number of mutants generated by μ_i on $\mathcal{P}$.
- $NM(\mathcal{P})$ is the total number of mutants of all $\mu_i \in \mathcal{OP}$.
- $Im_i(\mathcal{P}, \mathcal{S})$ is the mutation importance measure of $\mathcal{S}$ 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 $\mathcal{S}$. $Im_i^t(\mathcal{P}, \mathcal{S})$ is the mutation importance measure for $\mathcal{S}$ in iteration t.
- $IM(\mathcal{P}, \mathcal{S})$ is the mutation importance measure for $\mathcal{S}$ on $\mathcal{P}$. It is the fraction of all mutants of $\mathcal{P}$ (generated by $\mathcal{OP}$) that are left alive by $\mathcal{S}$. $IM^t(\mathcal{P}, \mathcal{S})$ is the corresponding mutation importance measure for iteration t.

In order to prioritize the operators whose mutants (of $\mathcal{P}$) are hard to expose with $\mathcal{S}$, a probability measure is associated with each operator:

$$< \mu_i, p_i > :\ p_i \propto \text{importance of operator } \mu_i \qquad (1)$$

where probability p_i of μ_i is low if $\mathcal{S}$ kills a large proportion of the mutants generated by μ_i on $\mathcal{P}$. Since mutants are generated by applying the operators on a specific $\mathcal{P}$ and evaluated using specific $\mathcal{S}$, references to $\mathcal{P}$, $\mathcal{S}$ are frequently dropped—for e.g., Nm_i is used instead of $Nm_i(\mathcal{P})$.

B. Initial Distributions

The iterative procedure selects an initial (small) set of mutants from the set of all mutants. This set can be chosen by sampling *uniformly*, i.e., the same number of mutants can be chosen for each operator with a non-zero set of available mutants. Another option is to make the number of chosen

mutants of each operator proportional to the relative number of available mutants of that operator:

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

where $numMutantSamps_i^0$ is the number of mutants of operator μ_i to be examined in the first iteration and c is an arbitrary integer. The corresponding initial operator probabilities are also set to be uniform (i.e., $\frac{1}{N}$ if $\forall i\, Nm_i(\mathcal{P}) > 0$) or proportional to the relative number of mutants generated by each operator (i.e., $\propto \frac{Nm_i}{NM}$). With either scheme, the number of mutants selected in the initial set is *much* smaller than the total number of available mutants. This initial set is used to initiate the iterative process described below.

C. Probability Update

Since the target program does not change while it is being analyzed, step-1 of the iterative procedure described in Section II-B is not required. Step-2 of the iterative procedure updates the likelihood that each operator's mutants will remain unexposed. This probability update is based on the proportion of mutants selected that are left alive by the test suite in each iteration:

$$p_i^t = p_i^{t-1} + \delta p_i^t \frac{1}{totalMutantSamps^t} \quad (3)$$

$$\delta p_i^t = -1.0 + 2.0 \frac{numAlive_i^t}{numMutantSamps_i^t} \; : \in [-1.0, 1.0]$$

$$totalMutantSamps^t = \sum_{i=0}^{N-1} numMutantSamps_i^t$$

where $numMutantSamps_i^t$ is the number of mutants of μ_i examined in iteration t, of which $numAlive_i^t$ mutants are left alive, while $totalMutantSamps^t$ is the total number of mutants analyzed in this iteration. The probability of operator μ_i in iteration t (i.e., p_i^t) is the sum of the probability in the previous iteration (p_i^{t-1}) and an incremental factor. This factor is based on the fraction of chosen mutants of μ_i that are left alive in the current iteration ($\frac{numAlive_i^t}{numMutantSamps_i^t}$), and it is inversely proportional to the total number of mutants considered in iteration t. The term $\delta p_i^t \in [-1, 1]$ constrains the change in probability (in each iteration) and provides robustness to noisy observations. After the update, the operator probabilities are normalized to sum to one, a necessary condition for probability measures:

$$p_i^t = \frac{p_i^t}{\sum_j p_j^t} \quad (4)$$

The updated probabilities represent the current estimate of the relative priority of the operators.

D. Importance Sampling

Step-3 of the iterative process uses the updated operator probabilities to compute the number of mutants of each operator to be examined in the next iteration. The goal is to select a proportionately larger number of mutants of operators with larger probabilities, i.e., to focus more on operators whose mutants are more likely to remain unexposed. This goal is achieved using Algorithm 1, a modified version of the *importance sampling* scheme [16], [19].

Algorithm 1 Importance Sampling for Mutation Testing.

Require: Set of <operator, probability> vectors (in iteration t) of the form: $\{< \mu_{ds,k}, p_{ds,k}^t >: k \in [0, N-1]\}$ sorted based on probability such that $p_{ds,k}^t \geq p_{ds,k+1}^t$.

{Initialize mutant counts and cumulative probability.}

1: $\forall k \in [0, N-1]$, $Count_k^{t+1} = 0$; $b_0 = p_{ds,0}^t$

{Compute the cumulative probability distribution.}

2: **for** $k = 1$ to $N-1$ **do**

3: $\quad b_k = b_{k-1} + p_{ds,k}^t$.

4: **end for**

5: Compute $\mathcal{Y}$, the number of mutants to be examined in iteration $(t+1)$ – see Equation 6.

6: $r_0 \sim U\,]0, \frac{1}{\mathcal{Y}}\,]$, iterator $k = 0$

{Compute the mutant counts for each operator.}

7: **for** $j = 0$ to $\mathcal{Y} - 1$ **do**

8: $\quad$ **while** $r_j > b_k$ **do**

9: $\quad\quad k = k + 1$

10: $\quad$ **end while**

11: $\quad Count_k^{t+1} = Count_k^{t+1} + 1$

12: $\quad r_{j+1} = r_j + \frac{1}{\mathcal{Y}}$

13: **end for**

14: Return re-ordered $\{Count_k^{t+1}\} : k \in [0, N-1]$.

The algorithm takes as input the operators sorted in decreasing order of their updated probabilities in iteration t: $\{< \mu_{ds,k}, p_{ds,k}^t >: k \in [0, N-1]\}$—for e.g., $\mu_{s,0}$ is the operator with the highest probability. The first step initializes the number of mutants of each operator that will be analyzed in the next iteration (line 1 of Algorithm 1). Next, the cumulative probability distribution is computed (lines 2–4), which sums up the probability contribution of all operators. Similar to the default importance sampling algorithm, a random number is chosen from a uniform distribution ($r_0 \sim U\,]0, 1/\mathcal{Y}\,]$) based on the total number of mutant samples that will be analyzed in the next iteration ($\mathcal{Y}$). The value of $\mathcal{Y}$ is computed dynamically as described in Equation 6. Based on this selection of r_0 and the arrangement of operators in decreasing order of probability, the count of samples for the operator with the highest probability is incremented (line 11) until this operator's relative contribution to the cumulative probability distribution is taken into account (i.e., until $r_j > b_k$ is true). Then the focus shifts to the operator that made the second largest contribution to the cumulative probability (line 9). The algorithm terminates when the desired count of samples is reached, and the indices of the computed counts are adjusted to account for the sorting of the operators at the beginning of Algorithm 1.

E. Effect of Sampling Parameters

The proposed approach prioritizes mutation operators such that more attention is devoted to operators whose mutants are more likely to remain unexposed with the existing test suite. The test suite can then be appropriately augmented and the program can be tested thoroughly. However, the following questions need to be answered:

(a) Should mutants be sampled with or without replacement? Mutation testing is a *stationary* domain. Since no additional information is gained by examining a mutant of a specific program more than once with a specific test suite, mutants are examined without replacement.

(b) How many iterations must be performed? The objective of sampling is to gather information that prioritizes the mutation operators. *Entropy*[1] can be used as a measure of the information encoded by the current probabilities of the mutation operators [20]:

$$E^t = -\sum_{j=0}^{N-1} p_j^t \cdot ln(p_j^t) \tag{5}$$

where E^t and p_j^t are the entropy and j^{th} operator's probability respectively (in iteration t). The entropy is maximum when the probability distribution is uniform, i.e., nothing is known about the relative importance of the operators. The entropy is small when the probabilities of a small number of mutation operators are significantly larger than the others. The entropy decreases as more information becomes available, and sampling is hence terminated when the entropy does not decrease substantially between two iterations.

(c) How many mutants should be examined in each iteration? The probability distribution over the set of mutation operators is used to estimate the true (unknown) distribution of relative operator importance. Existing research in sampling-based inference can be used to compute a bound on the number of samples that need to be examined in order to ensure that the estimated distribution of operator probabilities matches the true distribution with a high degree of certainty [17]. As more information becomes available (over successive iterations), the sampling-based distribution becomes more accurate and causes a progressive decrease in the number of samples that need to be examined. The value of parameter $\mathcal{Y}$ in Algorithm 1 can hence be computed as:

$$\begin{aligned}\mathcal{Y}^{t+1} &= \frac{1}{2\epsilon}\chi^2_{q^t-1,1-\delta} \\ &\simeq \frac{q^t-1}{2\epsilon}\left\{1-\frac{2}{9(q^t-1)}+\sqrt{\frac{2}{9(q^t-1)}}z_{1-\delta}\right\}^3\end{aligned} \tag{6}$$

where $\mathcal{Y}^{t+1}$ is computed based on approximated quantiles of the chi-square distribution ($\chi^2_{q^t-1,1-\delta}$). Equation 6 guarantees that with probability $1-\delta$, the sampling-based estimate approximates the true distribution with an error $\leq \epsilon$. Here, q represents the number of operators that had a non-zero number of mutants examined in iteration t. For a given δ, ϵ (e.g., $\delta = 0.1, \epsilon = 0.2$), the value of $z_{1-\delta}$ can be obtained from standard normal tables to estimate the number of samples that need to be examined in iteration $t+1$ in order to accurately estimate the true distribution of relative operator importance. Equations 5,6 can be used to automatically trade-off computation against accuracy.

[1]Entropy is a measure of the uncertainty associated with a random variable (RV). A large value of entropy indicates a higher uncertainty, while a small value implies a greater confidence in the estimated value of the RV.

Program	NLOC	NTC	NMG	NMS	NM
printtokens	343	4130	11741	2000	1551
printtokens2	355	4115	10266	2000	1942
replace	513	5542	23847	2000	1969
schedule	296	2650	4130	2000	1760
schedule2	263	2710	6552	2000	1497
tcas	137	1608	4935	4935	4935
totinfo	281	1052	8767	2000	1740

Table I
DESCRIPTION OF SUBJECT PROGRAMS. NLOC: NET LINES OF CODE. NTC: NUMBER OF TEST CASES. NMG: NUMBER OF MUTANTS GENERATED BY ALL MUTATION OPERATORS. NMS: NUMBER OF RANDOMLY SELECTED MUTANTS. NM: NUMBER OF SELECTED MUTANTS THAT WERE NON-EQUIVALENT.

IV. CASE STUDIES

This section describes the case studies performed on a range of subject programs.

A. Subject Program

The first set of experiments were performed on the seven Siemens programs [21] and the associated test cases[2]. As summarized in Table I, this is a collection of C programs ranging in size from 137 to 513 lines of code (excluding comments and blank lines).

B. Data Collection

For each program, mutants from an earlier study [10] based on the Proteum mutant generator [3] were used. Proteum implements 108 mutation operators based on the specification designed by Agrawal et al. [7]. In order to make it feasible to run the experiments[3], 2000 mutants were selected randomly for each program, with the number of selected mutants of each operator being proportional to the relative number of mutants generated by the operator. The only exception was the program tcas, where an earlier study provided results corresponding to all generated mutants [22]. Mutants that were not killed by any test case in the test pool were considered equivalent. In the experiments described below, the terms "all" or "existing" refer to the set of non-equivalent mutants shown in Table I.

For each subject program, test cases were chosen randomly from the test pool to create one test suite each of size $1-50$ test cases. Two different schemes were used to generate the initial set of mutants for each combination of

[2]The subject programs used in this paper were obtained from the Subject Infrastructure Repository (SIR).

[3]Earlier exploratory work with tcas indicated that using all generated mutants was infeasible [22].

Prior Distribution	δIM(%)	Improvement (%)
Uniform	5.80 ± 2.29	91.4
Proportional	4.95 ± 2.32	98.0

Table II
IMPROVEMENT IN IM WITH DIFFERENT INITIAL OPERATOR PROBABILITY DISTRIBUTIONS.

of program and test suite (Equation 2). In the first scheme, the initial set was composed of a fixed number of mutants of each operator ($c = 2$) that resulted in a non-zero number of mutants. In the second scheme, the number of chosen mutants of an operator was proportional to the relative number of mutants generated by the operator. The subsequent steps were identical for both schemes. For the $\mathcal{P}$ and $\mathcal{S}$ under consideration, the set of mutants chosen in iteration t were analyzed to compute the fraction of mutants left alive by $\mathcal{S}$ (i.e., Im_i^t, IM^t), thereby updating the operator probabilities and computing the number of mutants of each operator to be examined in the next iteration. In order to evaluate the performance of the proposed approach, Im_i, IM were also computed for each $\mathcal{P}$ and $\mathcal{S}$ using the default approach of analyzing all (non-equivalent) mutants of each operator.

C. Evaluation

The subject programs were used to evaluate the ability of the proposed approach to prioritize the mutation operators.

1) Sampling Effectiveness: Figures 1(a)–1(e) show the evolution of operator probabilities over a few iterations, for a specific program and test suite. All operators with a non-zero number of mutants are sampled uniformly in the first step. The sampling process terminates after a small set of iterations, i.e., $t = 5$, at which point the focus of attention converges to a small set of operators. In addition, Figure 1(f) shows that the number of mutants examined decreases as the iterations proceed. The total number of mutants analyzed is $\leq 20\%$ of the existing mutants. When each operator's initial probability is proportional to the relative number of its mutants, the convergence is faster but the number of mutants examined in each iteration decreases slowly, as shown in Figures 2(a)–2(f).

Next, Table II compares the performance of the two initial distribution schemes against the default approach, over all programs and test suite sizes described in Section IV-B. The column "δIM(%)" reports the increase in IM achieved by the sampling-based approaches, in comparison to the default approach. With adaptive sampling, a larger fraction of the examined mutants are alive (i.e., more attention is devoted to the alive mutants) even though only a small subset of the mutants are examined. The "Improvement" is seen in a larger number of trials when the initial operator probabilities are proportional to the size of the corresponding mutant sets.

2) Operators Selection: The ability to prioritize the operators whose mutants are hard to expose, was evaluated next. The Im_i of the operators, computed by analyzing each operator's mutants for each $\mathcal{P}$ and $\mathcal{S}$, were sorted in descending order to obtain a *ground-truth* list of the operators in decreasing order of priority. The operator probabilities computed during the sampling iterations were also sorted to create the *observed* list of important operators. The *operator overlap* measure was then defined as:

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

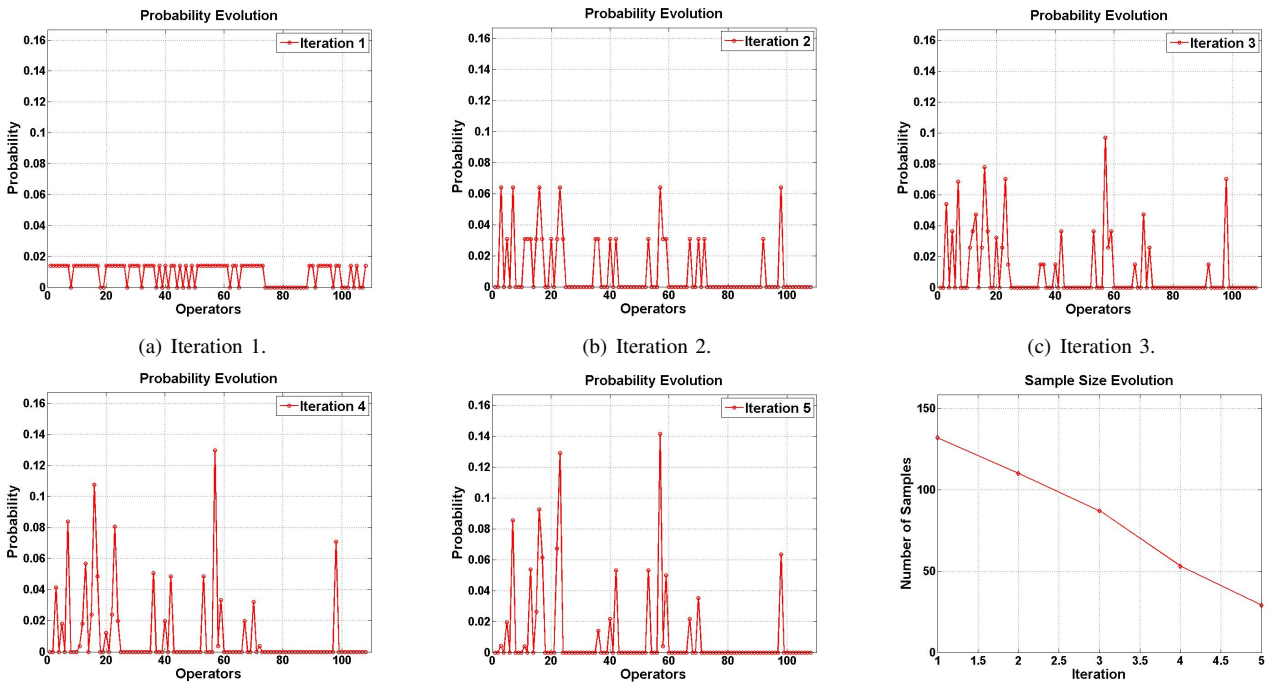
which computes the fraction of top $T\%$ operators in the ground-truth list (Gt) that also exist among the top $T\%$ of the observed list (Obs), for a specific $\mathcal{P}$ and $\mathcal{S}$. Table III summarizes the results for the two initial operator probability assignment schemes (Equation 2).

Programs	Operator Overlap			
	Max	Min	Average	Dynamic
Uniform				
printtokens	0.93	0.56	0.76 ± 0.07	0.9
printtokens2	0.92	0.48	0.69 ± 0.11	0.94
replace	1.0	0.80	0.87 ± 0.06	0.9
schedule	0.90	0.48	0.68 ± 0.1	0.9
schedule2	0.91	0.68	0.77 ± 0.04	0.89
tcas	1.0	0.74	0.85 ± 0.06	0.89
totinfo	1.0	0.51	0.69 ± 0.10	0.95
Proportional				
printtokens	0.94	0.60	0.77 ± 0.09	0.92
printtokens2	0.97	0.47	0.66 ± 0.13	0.9
replace	1.0	0.65	0.86 ± 0.07	0.94
schedule	0.90	0.48	0.67 ± 0.11	0.98
schedule2	0.92	0.66	0.78 ± 0.05	0.91
tcas	0.98	0.85	0.91 ± 0.04	0.92
totinfo	0.96	0.51	0.74 ± 0.09	0.96

Table III
OPERATOR OVERLAP WITH THE TWO SCHEMES FOR INITIAL OPERATOR PROBABILITY ASSIGNMENT. DYNAMIC SELECTION OF "T" PROVIDES BETTER RESULTS.

The "Max", "Min" and "Average" columns of Table III report the maximum, minimum and average OpOverlap (operator overlap) obtained with the proposed approach over all 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 many cases, the top operators in the ground-truth list have low IM values, i.e., they do not represent the important operators whose mutants are hard to expose. Therefore, experiments were also run with the value of T being set dynamically based on the Im_i values in the ground-truth list—only operators with Im_i above a reasonable threshold (0.4) were considered important. The corresponding (average) results are summarized in the columns labeled "Dynamic": $\approx 90\%$ (average $\approx 95\%$) of the truly important operators are detected using the sampling-based approach. Even the top operators in the ground-truth list that are not found by the sampling approach, exist just beyond the top $T\%$ in the observed list. The following observations can be made:

- The proposed method increases IM by $\approx 6\%$ in comparison to the default approach. Though the differences are more pronounced for smaller test suites, further analysis is required before stronger claims can be made.



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3.

(d) Iteration 4. (e) Iteration 5. (f) Sample sizes.

Figure 1. (a)-(e) Evolution of operator probabilities for program `printtokens` with test suite size $= 20$, starting with a *uniform* initial distribution; (f) Number of samples used in each iteration.

- The proposed approach is able to identify $\geq 90\%$ of the important operators identified by examining all the mutants, while examining $\leq 20\%$ of the mutants over $3-4$ sampling iterations.

D. Analysis of High-Priority Operators

Table IV summarizes the top ten mutation operators, whose mutants were identified as being the most difficult to expose with the existing test suites, for the two different initial operator probability assignment schemes. These results document the contributions of all operators over all subject programs. However, for any given program and test suite, either scheme of initial probability assignment results in a similar list of important operators. The results also make sense from a software testing perspective. For instance, inserting negations on arguments and deleting `return` statements produce hard-to-detect mutants.

V. Studies on Other Programs

In order to evaluate the performance of the proposed approach on larger programs, it was applied on the `gzip` and `space` programs from SIR. For `space`, mutants generated by all mutation operators were considered. However, for `gzip`, the relationship between adaptive sampling and a sufficient set of mutation operators [10] was investigated.

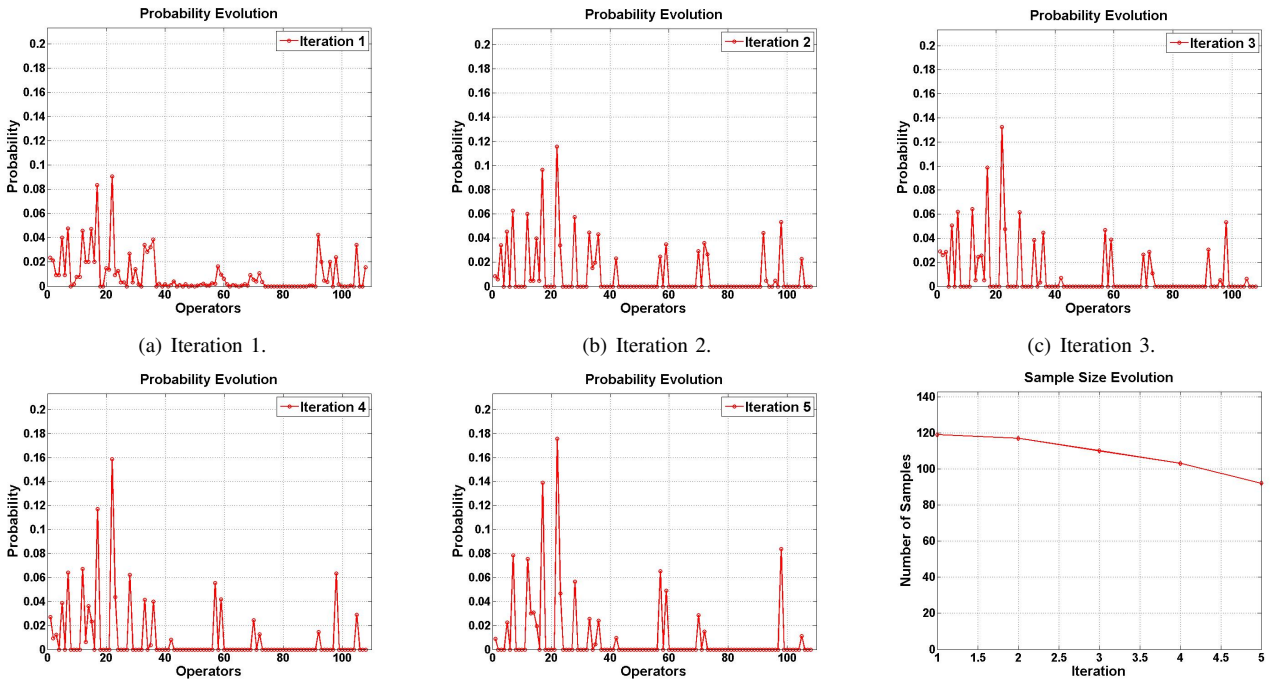
A. Subject Programs

Version 5 of program `gzip` was used in the experiments which contains 5680 net lines of code and 214 test cases. The experiments used 212 test cases because two of them did not work properly on the experimental platform. The second program `space` contains 5905 net lines of code and 13585 test cases.

B. Data Collection

Mutants were generated for both `gzip` and `space` using Proteum [3]. For `space`, Proteum generated 301356 mutants using 108 mutation operators. In order to make the study feasible, 30136 mutants (i.e., 10%) were selected, with the number of mutants of each operator being proportional to the total number of mutants generated by that operator. When the mutants were evaluated with all test cases, 6428 were not killed by any test case—the remaining 23708 non-equivalent mutants were used in the experiments.

Siami Namin et al. [10] had previously identified a sufficient set of 28 Proteum mutation operators, i.e., the behavior of subject programs on these operators is equivalent to the behavior when the mutants of all operators are considered. In addition, 15 operators identified using coverage-based test suites were also included to result in a set of 43 mutation operators. For these 43 operators, Proteum generated 38621 mutants of `gzip`, which were considered for this case study. Since each run of `gzip` took a long time, $\approx 1\%$ of the mutants from each operator were randomly selected and the non-equivalent mutants (317) were used in the experiments. As before, test suites of sizes $1-50$ test cases were generated by randomly choosing test cases from the test pool.



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3.

(d) Iteration 4. (e) Iteration 5. (f) Sample sizes.

Figure 2. (a)-(e) Evolution of operator probabilities for program `printtokens` with test suite size = 20, starting with an initial distribution *proportional* to the number of existing mutants; (f) Number of samples used in each iteration.

C. Data Analysis

Table V summarizes the operator overlap results for `gzip` and `space`. In comparison to the default approach of evaluating all non-equivalent mutants, adaptive sampling identifies at least 97% and 81% of the important operators for `gzip` and `space` respectively. Even when some of the high-priority operators (on the ground truth list) are not within the top "T" on the observed list, they are just a few positions further down on the observed list.

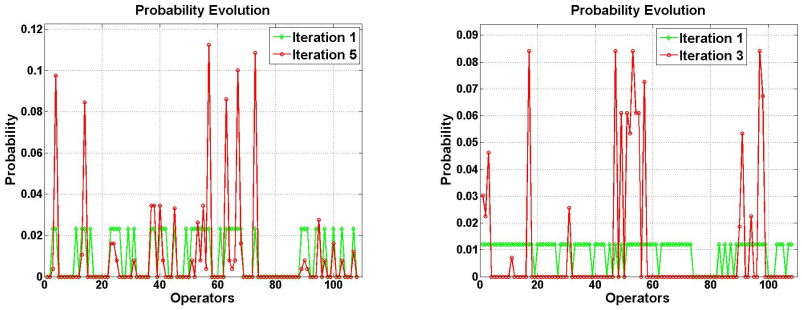
Table VI compares the proposed approach against the default approach in terms of the increase in the mutation importance (IM). There is a substantial increase in IM for both initial operator probability assignment schemes: uniform and proportional. However, the gain is observed in a larger number of cases (column labeled "Improvement") when the initial operator probabilities are proportional to the corresponding number of existing mutants.

Figure 3(a) shows the operator probabilities at the beginning and end of the sampling process for `gzip` for a test suite of size 25. Figure 3(b) shows the corresponding plot of operator probabilities of `space` for a test suite of size 25. Both figures correspond to the case where the initial operator probabilities are distributed uniformly over operators with a non-zero number of mutants. Next, Figure 3(c) shows the number of mutants examined in each iteration for the program `gzip`, for both schemes for initial operator probability assignment and a test suite of size 25. Figure 3(d) shows the corresponding plot for `space`. The following observations can be made:

- The proposed method adapts to different programs: $\geq 80\%$ of the important operators (average $\approx 90\%$) are identified by examining a subset of the mutants.
- The approach performs better for `gzip`—it is able to find important operators even among the sufficient set of mutation operators, by examining a subset of the mutants.
- The operator probabilities do not increase substantially for `space` and prioritizing the operators is more difficult. This accounts for the lower $OpOverlap$ values in comparison to those for `gzip`, especially with a uniform prior. One reason for this performance could be the large pool of test cases developed for `space`.
- Since the number of mutants of `space` is large, the *uniform* scheme for initial probability assignment results in a smaller number of iterations (and examined mutants) than the *proportional* scheme. However, the performance is better with the *proportional* scheme.

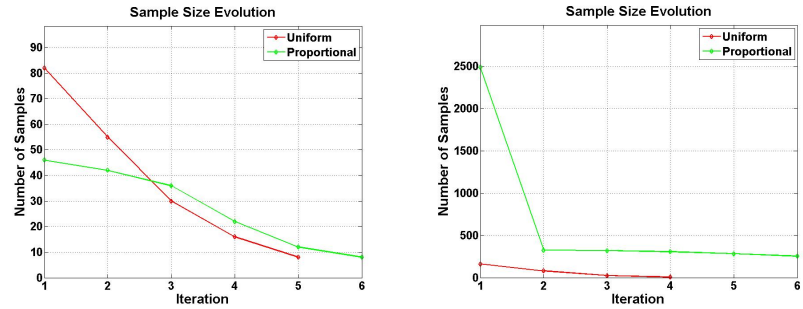
D. Discussion

Table VII summarizes some high-priority operators for `gzip`, for the two different schemes for initial operator probability assignment. Table VIII summarizes the corresponding results for `space`. There is a significant overlap between the important operators identified for `gzip` with the *uniform* and *proportional* schemes. However, for `space`, the high-priority operators identified with the two initial operator



(a) Probability evolution for `gzip`.

(b) Probability evolution for `space`.



(c) Sample size evolution for `gzip`.

(d) Sample size evolution for `space`.

Figure 3. (a)-(b) First and last iteration of probability evolution for `gzip` and `space` for uniform initial probabilities and $|\mathcal{S}| = 25$; (c)-(d) The number of samples used in each iteration for `gzip` and `space` for both schemes for initial probability assignment, for $|\mathcal{S}| = 25$.

probability assignment schemes are different because: (a) the *proportional* scheme is significantly better than the *uniform* scheme; and (b) the increase in operator probabilities is not very large during the sampling process.

VI. Threats to Validity

Threats to external validity include the use of relatively small to medium-sized C programs. The results reported in this paper, based on these programs, support the initial hypothesis, i.e., that a test suite that detects a certain type of fault is likely to detect other similar faults. Larger programs may have more complex structure, which may lead to different results. Object-oriented programs also need to be investigated, since they contain additional features that may lead to different results. In addition, a mutant generator for object-oriented programs (such as MuJava [4]) that implements class mutation operators, may behave differently. These are directions for future research.

Threats to construct validity include the randomly selection of mutants from each program other than `tcas`. This was necessary to make the study feasible. Performing all the computation for mutants, such as generation, compilation and execution is computationally expensive. 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. Each author monitored the results of the intermediate processes separately to assure their correctness.

VII. Conclusion and Future Work

This paper describes a novel probabilistic approach for effective mutation testing. The approach incorporates innovations of well-established information-theoretic and sampling methods to iteratively direct the attention towards operators whose mutants are hard to expose using the existing test suites. As a result, there is an average improvement of $\approx 6.0\%$ in the mutation importance measure, while detecting $\approx 90\%$ of the high-priority operators (whose mutants are hard to catch) by examining as few as 20% of the available mutants. Furthermore, the approach automatically and elegantly trades-off computational effort against accuracy.

This work has only opened up a new direction of further research. As mentioned in Section VI, there are many interesting questions that require further investigation. This work is part of a long-term goal of designing Bayesian formulations for software testing challenges.

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977.

Operator
Uniform
-I-RetStaDel
Delete Return Statement
-u-VDTR
Domain Traps
-u-OEBA
Plain Assignment by Bitwise Assignment
-u-OLBN
Logical Operator by Bitwise Operator
-I-IndVarAriNeg
Insert Arithmetic Negation at Non Interface Variables
-u-OCOR
Cast Operator by cast Operator
-I-DirVarAriNeg
Insert Arithmetic Negation at Interface Variables
-I-IndVarBitNeg
Insert Bit Negation at Non Interface Variables
-I-CovAllEdg
Coverage of Edges
-I-IndVarIncDec
Increment and Decrement Non Interface Variables
Proportional
-I-IndVarRepReq
Replace Non Interface Variables by Required Constants
-u-VDTR
Domain Traps
-u-CRCR
Required Constant Replacement
-I-DirVarRepReq
Replace Interface Variables by Required Constants
-u-VLSR
Mutate Scalar References Using Local Scalar References
-I-DirVarIncDec
Increment and Decrement Interface Variables
-u-Cccr
Constant for Constant Replacement
-I-IndVarIncDec
Increments and Decrements Non Interface Variables
-I-DirVarRepCon
Replace Interface Variables by Used Constants
-I-IndVarRepCon
Replace Non Interface Variables by Used Constants

Table IV
TOP TEN MUTATION OPERATORS IDENTIFIED BY UNIFORM AND PROPORTIONAL INITIAL OPERATOR PROBABILITY DISTRIBUTIONS, AVERAGED OVER THE SUBJECT PROGRAMS SUMMARIZED IN TABLE I.

Programs	Operator Overlap			
	Max	Min	Average	Dynamic
Uniform				
`gzip`	0.98	0.95	0.96 ± 0.01	0.97
`space`	0.91	0.35	0.64 ± 0.17	0.81
Proportional				
`gzip`	0.98	0.95	0.96 ± 0.01	0.97
`space`	0.96	0.34	0.67 ± 0.19	0.87

Table V
OPERATOR OVERLAP FOR `gzip` AND `space` WITH UNIFORM AND PROPORTIONAL INITIAL OPERATOR PROBABILITY DISTRIBUTIONS.

Prior Distribution	δIM(%)	Improvement (%)
Uniform	4.85 ± 2.35	84.23
Proportional	5.09 ± 2.69	93.33

Table VI
IMPROVEMENT IN IM FOR `gzip` AND `space` WITH UNIFORM AND PROPORTIONAL INITIAL OPERATOR PROBABILITY DISTRIBUTIONS.

Important Operators: `gzip`, uniform
-I-IndVarAriNeg
Insert Arithmetic Negation at Non Interface Variables
-I-IndVarLogNeg
Insert Logical Negation at Non Interface Variables
-u-OCOR
Cast Operator by cast Operator
-u-OASN
Replace Arithmetic Operator by Shift Operator
-I-IndVarBitNeg
Insert Bit Negation at Non Interface Variables
-u-OLRN
Logical Operator by Relational Operator
-I-DirVarRepPar
Replace Interface Variables by Formal Parameters
-I-DirVarBitNeg
Insert Bit Negation at Interface Variables
-u-OAAN
Arithmetic Operator Mutation
-I-DirVarAriNeg
Insert Arithmetic Negation at Interface Variables

Important Operators: `gzip`, proportional
-u-OCOR
Cast Operator by cast Operator
-I-IndVarLogNeg
Insert Logical Negation at Non Interface Variables
-u-VGPR
Mutate pointer References using Global pointer References
-I-IndVarAriNeg
Insert Arithmetic Negation at Non Interface Variables
-I-DirVarBitNeg
Insert Bit Negation at Interface Variables
-I-IndVarBitNeg
Insert Bit Negation at Non Interface Variables
-I-DirVarRepPar
Replace Interface Variables by Formal Parameters
-u-OASN
Replace Arithmetic Operator by Shift Operator
-u-OLRN
Logical Operator by Relational Operator
-u-STRI
Trap on if Condition

Table VII
TOP TEN MUTATION OPERATORS FOR `gzip` WITH UNIFORM AND PROPORTIONAL INITIAL OPERATOR PROBABILITY DISTRIBUTIONS.

[3] M. E. Delamaro and J. C. Maldonado, "Proteum – a tool for the assessment of test adequacy for C programs," in *Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, NJ, July 1996, pp. 79–95.

[4] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava : An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[5] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses versus mutation testing," *Systems and Software*, 1996.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation

Important Operators: `space`**, uniform**
-I-IndVarRepCon
Replace Non Interface Variables by Used Constants
-u-VDTR
Domain Traps
-u-Ccsr
Constant for Scalar Replacement
-u-OCOR
Cast Operator by cast Operator
-I-CovAllEdg
Coverage of Edges
-u-SMVB
Move brace up and down
-u-OBRN
Bitwise Operator by Relational Operator
-u-OBAN
Bitwise Operator by Arithmetic Assignment
-u-SWDD
while Replacement by do-while
-u-OALN
Replace Arithmetic Operator by Logical Operator

Important Operators: `space`**, proportional**
-I-IndVarRepCon
Replace Non Interface Variables by Used Constants
-u-Cccr
Constant for Constant Replacement
-I-DirVarRepCon
Replace Interface Variables by Used Constants
-I-IndVarRepReq
Replace Non Interface Variables by Required Constants
-u-VLSR
Mutate Scalar References Using Local Scalar References
-u-CRCR
Required Constant Replacement
-u-VDTR
Domain Traps
-u-DirVarIncDec
Increment and Decrement Interface Variables
-u-Ccsr
Constant for Scalar Replacement
-I-IndVarIncDec
Increments and Decrements Non Interface Variables

Table VIII
TOP TEN MUTATION OPERATORS FOR `space` WITH UNIFORM AND PROPORTIONAL INITIAL OPERATOR PROBABILITY DISTRIBUTIONS.

an appropriate tool for testing experiments?" in *International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005.

[7] 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," Department of Computer Science, Purdue University, Tech. Rep. SERC-TR-41-P, April 2006.

[8] 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*, San Jose, USA, October 2000, pp. 45–55.

[9] 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*, vol. 5, no. 2, pp. 99–118, April 1996.

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

[11] R. H. Untch, "On reduced neighborhood mutation analysis using a single mutagenic operator," in *Annual Southeast Regional Conference*, Clemson, SC, USA, 2009.

[12] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations," Yale University, Department of Computer Science, Tech. Rep., 1979.

[13] A. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *The Annual Conference on Computer Assurance*, Gaithersburg, USA, 1996, pp. 224–236.

[14] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233 – 262, 1999.

[15] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *International Symposium on Software Testing and Analysis*, Chicago, USA, 2009, pp. 69 – 80.

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

[17] D. Fox, "Adapting the Sample Size in Particle Filters through KLD-Sampling," *International Journal of Robotics Research*, vol. 22, no. 12, pp. 985–1004, December 2003.

[18] 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, pp. 3366–3371.

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

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

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

[22] A. Siami Namin and J. Andrews, "Finding sufficient mutation operators via variable reduction," in *Mutation 2006*, Raleigh, NC, USA, November 2006.