Tailored Mutants Fit Bugs Better

Miltiadis Allamanis
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
m.allamanis@ed.ac.uk

Earl T. Barr
Dept. of Computer Science
University College London
London, UK
e.barr@ucl.ac.uk

René Just
Dept. of Computer Science
University of Massachusetts
Amherst, MA, USA
rjust@cs.umass.edu

Charles Sutton
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
csutton@ed.ac.uk

***Abstract*—Mutation analysis measures test suite adequacy, the degree to which a test suite detects seeded faults (mutants): one test suite is better than another if it detects more mutants. The effectiveness of mutation analysis rests on the assumption that mutants are coupled with real faults—that is, mutant detection is strongly correlated with real fault detection. The work that validated this assumption also showed that a large portion of defects remain out of reach.**

We introduce tailored mutation operators to reach and capture these defects. Tailored mutation operators are built from and apply to an existing code base and its version history. They can, for instance, identify and replay errors specific to the project for which they are tailored. As our point of departure, we define tailored mutation operators for identifiers, which mutation analysis has largely ignored, because there are too many ways to mutate identifiers. Evaluated on the Defects4J data set, our new mutation operators allow mutation analysis to create mutants coupled to 14% more faults, compared to traditional mutation operators.

These new mutation operators, however, quadruple the number of mutants, exacerbating the problem of mutant selection. To combat this problem, this paper proposes a new approach to mutant selection that focuses on the location at which to apply mutation operators and the unnaturalness of the mutated code. The results demonstrate that the location selection heuristics produce mutants more closely coupled to real faults for a given budget of mutation operator applications.

In summary, this paper defines and explores tailored mutation operators, advancing the state of the art in mutation testing in two ways: 1) it suggests mutation operators that mutate identifiers and literals, extending mutation analysis to a new class of faults and 2) it demonstrates that selecting the location where a mutation operator is applied decreases the number of generated mutants without affecting the coupling of mutants and real faults.

***Keywords*-mutation testing; submodular optimization; code naturalness**

I. Introduction

Mutation analysis is a well-established approach to measure the effectiveness of a test suite. The *mutation score*, which measures a test suite's ability to distinguish a program under test from many small syntactic variations (*mutants*), is a proxy metric for that test suite's ability to detect real faults. These mutants are created by applying well-defined *mutation operators*, which are program transformations that systematically inject small artificial faults into the program under test. Examples of such mutation operators include replacing an arithmetic operator, manipulating a branch condition, or deleting a statement.

The usefulness of this proxy metric rests on the assumption that a test suite's ability to detect mutants is strongly correlated with its ability to detect real faults, and prior work provided empirical evidence for the existence of such a correlation [1], [2]. Indeed, one fundamental assumption of mutation analysis is the existence of the *coupling effect* [3]. A real fault, which is usually complex, is coupled to a set of mutants, which are usually simple, if a test that detects all the mutants also detects the real fault. Just *et al.* empirically showed that such a coupling effect indeed exists between *traditional mutants*, *i.e.*, mutants generated by commonly used mutation operators, and most real faults [2]. However, they also identified inherent limitations of mutation analysis: 27% of real faults were not coupled to any mutant due to the lack of suitable mutation operators. This suggests a need to introduce new mutation operators that are designed to be coupled to these uncovered faults.

However, introducing new mutation operators can be problematic, because mutation analysis suffers from scalability problems due to the large number of mutants that can be generated—even for mid-sized programs, and selective mutation is the most common approach to control the computational costs. A number of empirical studies conclude that randomly selecting mutants from a pool of generated mutants is at least as effective as selecting a particular set of mutation operators [4], [5]. Moreover, Kurtz *et al.* showed that no single selective mutation approach works reasonably well for all programs, and they argue that a good mutant selection strategy depends on the program under test [6].

This paper aims to improve the effectiveness of mutation analysis by introducing new mutation operators that are tailored to the specific program under analysis. By *tailored* we mean that the set of code transformations produced by the mutation operator is specific to the project in question, in the sense that the mutants are built from and apply to an existing code base and its version history. For example, a tailored mutation operator might use literals that occur elsewhere in the project, insert calls to project-specific APIs, insert code fragments from elsewhere in the project, or even identify and replay errors from the project's version history. To the best of our knowledge, we are the first to introduce such tailored operators. More specifically, in this work, we introduce tailored mutation operators for identifiers and literals: Our operator for identifiers replaces one identifier with another that is of the correct type

and is in scope, and our operator for literals replaces a literal with one that is used elsewhere in the project.

These tailored operators are extraordinarily prolific. A single operator can in some cases generate thousands of mutants at a single lines of code, and tailored operators generate on average *4 times* more mutants than a state of the art set of traditional mutation operators. Because each individual mutation operator is so prolific, selecting mutations at the operator level is not effective. Instead we introduce two new methods for selecting individual mutants. First, we introduce a novel approach to selecting locations at which to mutate based on *submodular optimization* that intuitively attempts to select locations that are as far apart as possible in the program's control flow graph. Submodularity is a well-known property of set functions, analogous to convexity in continuous functions, that has seen wide applications in economics, operations research, and artificial intelligence, but seems to have received less attention in software engineering. Second, we introduce a method for ranking individual mutants based on naturalness. Following the insight of Ray *et al.* [7] that bugs are often unnatural, we rank highest unnatural mutants, that is, those that have low probability according to an n-gram language model.

Our empirical evaluation studies the improvements in effectiveness and efficiency these operators bring to mutation testing, using 253 real faults and 659,326 mutants. The results show that the new mutation operators are coupled with 67% of the defects, increasing the overall ratio of coupled defects to 80% when combined with traditional mutants. In addition, our novel mutant selection approach, for a given target effectiveness, can greatly reduce the execution budget required for mutation testing. For example, to find mutants that are coupled to 70% of the defects, naive random selection of mutants requires on average twice as many mutants as our selection method to reach this level of effectiveness.

The practical implications for mutation testing include

- When focusing on a specific scope of a method or a line to detect bugs, using a combination of traditional and tailored mutation operators will improve the effectiveness of the mutation analysis;
- When using either traditional mutants and tailored mutants, methods that select mutants based on their location can provide better efficiency for a given budget.

II. Approach

Traditional mutation operators have proven their utility and enjoyed industrial adoption. One reason for their success is their universality: to increase their applicability to programs across many application domains and even across programming languages, these operators have tended to focus on local, fine-grained changes to symbols with near universal meaning, like relational operators. Despite their success, previous work [2], [8] has shown that traditional mutants are dissimilar to real faults and, more importantly, are not coupled to important classes of defects.

Motivated by Kurtz *et al.* [6] who found that an effective set of mutation operators depends on the program, and Just *et al.* [2] who found that new stronger mutation operators are needed, such as faults that have a "similar (library) method called", "specific literal replacement" *etc.*, we extend traditional mutation operators with tailored operators. These operators are tailored to a particular program, yet retain universality in the sense that they are automatically generated for an input program. One class of these operators can be extracted from a project's version history. For instance, one might identify bug-introducing commits and extract a mutation operator from them. A canonical class of tailored operators is literal mutators. Here a naïve choice of replacements is unbounded.

To square the circle of realizing finite customization, we turn to the notion that software is natural, that it is locally regular and predictable [9]. Using this observation, Ray *et al.* [7] found that less natural code tends to be more "buggy". Naturalness allows us to rank alternatives when defining tailored mutation operators. When defining a literal replacement operator, for instance, we rank our choices in terms of the ones that are the most unnatural, or most surprising, to a language model and drop the rest. It also allows us to constrain where we apply our mutation operators and rank the operators to apply at those locations. In each of these cases, the intuition is that highly-ranked mutants are more likely to be similar to mistakes a developer might make and therefore produce mutants that are coupled with some of the defects that traditional operators do not reach.

A. Terminology

As is standard, a mutation operator is a rewriting rule. A mutant is a program variant produced by the application of a mutation operator. We consider only mutants produced by a single application of a mutation operator, not those produced by $k > 1$ applications, so called higher order mutants [10]. As a result, we sometimes use mutant and the application of a mutation operator interchangeably in the following. A *tailored mutant* is a mutant produced by a tailored mutation operator.

We use the term "program location" or simply "location" to denote a node in a fine-grained control flow graph (CFG), where, instead of a basic block, each node represents a single statement. At the source code level, each program location refers to one or more code tokens. In the current implementation of our system, we generate CFGs at an intraprocedural level, which means that we have one CFG for each method in the program. A sample CFG can be seen in Figure 1.

B. Tailored Mutation Operators

Tailored mutation operators can be built from commits or from recurrent syntactic patterns; they can perturb API protocols; they can operate on ASTs. A concrete example of an AST operator would be one that replaced function overloads, like `return solve(min, max)` with `return solve(f,min,max)` where `f` is type compatible and in scope. As proof-of-concept, we consider tailored operators that replace only one token at a time, like identifier names. A tailored operator can replace an

identifier name with another name used in similar contexts or even simply another name in scope. In this paper, we introduce identifier and literal replacement operators. These operators do not preserve semantics, but respect the type system and produce compilable mutants.

Our identifier operator applies to function calls and uses of variables, parameters and fields. When applied to a function call, it replaces the function name with the name of another function that has the same signature, respecting the scope of the receiver object of the original method call. When applied to uses of variables, parameters, or fields, it replaces the name at that usage with another name of a variable, parameter, or field that is in scope and compatible. A replacement variable name is compatible if it is type-compatible while a replacement field name is compatible if the alternate name is another field of the containing object.

Traditional mutation operators include an operator for literals that selects from a small, fixed set of type-compatible literals, such as -1, 0, and 1 for an integer literal, or the empty string or `null` for a string literal. While such a pre-defined set is effective in some cases, it misses specific replacements that cause subtle defects and, moreover, it generates a large number of trivial mutants. For example, replacing a dereferenced string literal with a null reference or a number literal that represents an array index with a negative number, raises an exception as soon as that mutant is executed, rendering it as trivial.

Mutating literals defines a potentially unbounded set of mutants; we exploit the naturalness of software to truncate and rank the infinite possibilities, by considering literal replacements that are common within a code base. To mutate literals, we use a project-specific, context-specific approach by mutating literals and variable usages to literals that are commonly used within that code base at a similar context. This process begins by looking for all the trigrams in the rest of the code base that have the same prefix as the current token being mutated. This step ensures that common literals of the code base are added to the list of candidates. Additionally, for literal mutation we add as candidates all primitive variables of the correct type. Finally, we remove redundant target numeric literals that have equivalent values (*e.g.* `-0` and `0` or `10e-1` and `0.1`).

Our evaluation shows that these tailored mutation operators capture a wide variety of real defects that humans introduce into the code. They also produce vastly more mutants, exacerbating the cost of mutation analysis. Next, we discuss how we address this problem by judiciously deciding where to apply which operators.

C. Selecting Locations

To exploit our new mutants, we must sample them efficiently. Here we explore selection policies that allow us to select a smaller set of mutants, without sacrificing their effectiveness. Since the number of mutants within a program location are usually limited, we first explore two methods for selecting mutation program locations. Selecting these locations allows us to pick a smaller set of mutants. As a baseline, we use the effectiveness of a *fully random* selection process. This process randomly selects a subset of the mutants from the pool of all available mutants.

Random Location First The first policy that we consider is to select a random program location first and then to select a mutation operator within that location. This two-stage random process removes any bias because of an imbalance on the number of mutants across locations, *i.e.* it has the effect that it down-weights mutants that appear in more "crowded" locations. To compute the effectiveness of this selection policy, we resort to a Monte Carlo simulation and average the results across simulations.

Minimum CFG Distance Selection Selecting locations at random is reasonable, but may not be optimal. We are interested in mutating as many relevant code paths as possible to make sure that all possible cases have been mutated. Additionally, we need to pick locations that maximize the marginal utility of each mutation. If we chose two nearby mutants there is a higher chance that the "fates" of those two mutants are correlated. Both of these considerations suggest that we should pick highly diverse locations to maximize the expected utility for each mutant execution. To achieve this we explore an optimization-based approach for selecting mutants.

One reasonable heuristic is to choose mutants in locations so as to minimize the total distance between each CFG node (*i.e.* location) to the closest node with a selected mutant. In other words, we wish to place mutants such that they are near as many statements as possible in the CFG. This heuristic implicitly selects mutants to increase coverage metrics. We formulate this as an optimization problem over sets, *i.e.*, let $\mathbb{L}$ be the set of all possible mutant locations at which at least one mutation is possible. Then we define an objective function $O : 2^{\mathbb{L}} \rightarrow \mathbb{R}$ over a potential set of locations

$$O(L) = \sum_{g \in G} \sum_{n \in N_g} \min_{l \in L} d(n,l), \quad (1)$$

and select a set $L^*_\kappa \subseteq \mathbb{L}$ of CFG nodes (locations) by solving the optimization problem

$$L^*_\kappa = \arg\min_{L} O(L) \quad (2)$$

subject to the constraint $|L| = \kappa$, where κ is a user-selected maximum number of mutants, G is the set of CFGs for all methods in the program, and N_g is the set of all locations (nodes) in the CFG g. This optimization essentially selects κ CFG nodes (in L^*_κ) to minimize the average distance between all nodes to their closest node in the set of selected nodes.

This optimization problem especially convenient because it is *submodular* [11][1]. Submodularity is a diminishing returns property, which says intuitively that if we consider adding one more location l to an existing set L, the benefit of adding l diminishes as the set L gets bigger. Submodularity has found application in wide range of similar problems to ours, including placement of sensors in sensor networks [12] and identifying

[1]Many more references and tutorials are available at http://submodularity.org by Andreas Krause and Carlos Guestrin.

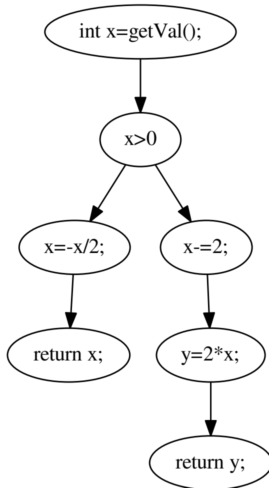


Fig. 1: An example control flow graph similar to those used in this work. Each node represents a single statement. The optimization objective (2) tries to pick M nodes (locations) to minimize the sum of the distances of all nodes to the nearest selected nodes.

the most influential nodes in a social network [13]. More formally, let T be an arbitrary set and let $f : 2^T \to \mathbb{R}$ map sets $S \subseteq T$ to the real numbers. We say that f is *submodular* if for all $S \subseteq S' \subseteq T$ (implying $|S| \leq |S'| \leq |T|$) and for all $x \in T - S'$, we have

$$f(S \cup \{x\}) - f(S) \geq f(S' \cup \{x\}) - f(S'). \tag{3}$$

which implies that adding an extra element x to a larger set S' yields a smaller increase compared to adding it to a smaller set S. More precisely, we show that the function $-O$ is submodular. For any $L \subseteq L' \subseteq \mathbb{L}$ and $l' \in \mathbb{L}$ with $l' \notin L'$, first consider a single CFG $g \in G$ and a single node $n \in N_g$. Then we have

$$\begin{aligned}\left[\min_{l \in L} d(n,l) - \min_{l \in L \cup \{l'\}} d(n,l)\right] &= \min\left\{0, \min_{l \in L} d(n,l) - d(n,l')\right\} \\ &\geq \min\left\{0, \min_{l \in L'} d(n,l) - d(n,l')\right\} \\ &= \min_{l \in L'}\left[d(n,l) - \min_{l \in L' \cup \{l'\}} d(n,l)\right].\end{aligned}$$

Now we can simply sum this inequality over all CFGs $g \in G$ and all nodes $n \in N_g$ to obtain the desired result, namely, that $O(L) - O(L \cup \{l\}) \geq O(L') - O(L' \cup \{l\})$.

Solving the optimization problem (2) requires maximizing a submodular function under a cardinality constraint. Although this problem is NP-hard, we resort to a greedy approximation, as is typical in the literature, for which strong approximation guarantees exist (see [14] for details). At each step i we greedily pick the location that *minimizes* the sum of the distances of each CFG node to the nearest node that belongs to the set of selected nodes.

To measure the distance $d(s,t)$ between two CFG nodes s and t, we use the smaller of the length of shortest path from s to t, or the length of shortest path from t to s (recall that the CFG is directed). Note that the distance between two nodes in mutually exclusive branches is infinite, since there is no path connecting them. Therefore, optimizing our objective implicitly maximizes branch coverage. Additionally, this selection process implicitly follows the intuition that branching nodes (*e.g.* `if` conditionals) should be selected early, since selecting such a node reduces the distance to nodes within both branches.

To illustrate Equation 2, consider the example graph in Figure 1. To find L_1^* we consider all nodes in the graph and select $L_1^* = \{$`x>0`$\}$. The sum of the distances of all nodes to the `x>0` node is $O(L_1^*) = 10$, which minimizes O subject to $\kappa = 1$. If we were to pick a node in any of the branches during the first step, O would have been infinite since there is no path connecting nodes of two separate branches. Next, to find L_2^*, we select $L_2^* = \{$`y=2*x;`$\} \cup L_1^*$ achieving $O(L_2^*) = 6$. The next step would include selecting either node `x=-x/2;` or node `return x;` to reduce $O(L_3^*)$ to 4. Note that although our optimization objective does *not* explicitly optimize for branch coverage or branching conditions, it implicitly covers our intuition that conditions should be mutated early and that high branch coverage should be achieved. Additionally, note that statements within basic blocks that contain more statements will be preferred to blocks with fewer statements. Finally notice that because of the submodularity of $-O$, adding extra nodes to the set of selected nodes has diminishing returns ($O(L_1^*) - O(L_2^*) = 4$, $O(L_2^*) - O(L_3^*) = 2$, $O(L_3^*) - O(L_4^*) = 1$, *etc.*).

D. Picking Mutation Operators at Location

Once a location is selected, we still need to pick the "best" mutant from the set of all available mutants at the specified location. One obvious approach is to pick a random mutant. We use this policy as our baseline.

Scoring Mutation Operators by Naturalness Recent work in *naturalness of code* [9] allows us to measure how probable a specific token is given information from the rest of the code within each project. Furthermore, Ray *et al.* [7] has recently found that unnatural code is on average more buggy. Therefore, mutating code to make it *less* natural implies a clear way of generating bugs.

Inspired from the NATURALIZE framework [15], [16] that used probabilistic models of source code to suggest alternative names for variables, methods and classes, we use the same technique to score tailored mutation operators. Note, that although Allamanis *et al.* suggest names that *increase* the naturalness of the code, here we prioritize mutants the *decrease* its naturalness. To measure the "naturalness" of each mutation, we use an n-gram language model. A language model (LM) is a probability distribution over strings. Given any string $x = x_0, x_1 \dots x_M$, where each x_i is a token, a LM assigns a probability $P(x)$.

Language models allow us to score different mutants based on the probability they assign to different productions of code. Given the sequence of code tokens $y_0 \dots y_M$ the score of a mutant in location l that is mutates the current token to t is

given by

$$S(t,l) = \log \frac{P(a_0 \dots a_M)}{P(y_0 \dots y_M)} = \sum_{i=l}^{l+n} \log \left(\frac{P(a_i | a_{i-1} \dots a_{i-n+1})}{P(y_i | y_{i-1} \dots y_{i-n+1})} \right) \quad (4)$$

where $a_i = y_i$ if $i \neq l$ otherwise $a_i = t$ and n refers to the size of the n-gram LM. S is the log-ratio of the probability of the mutated code to the probability of the unmutated code and is measures how more (or less) probable the token t is in location l compared to the current token y_l. Since $S(t,l)$ is a log-ratio, it suggests how many orders of magnitude more (or less) probable the mutated code is compared to the existing code. In our work, for a given program location, we rank mutants in an ascending order (*i.e.* from the least natural to the most natural). Since traditional mutants do not have a naturalness score and are a minimal set, they are placed first in the selection list.

E. Implementation

We implemented our approaches using shared infrastructure to ensure that our results accurately reflect differences between the mutation approaches rather than differences in their implementations. More specifically, we developed a mutation tool that builds on top of the Major mutation framework [17]. Our tool generates traditional and tailored mutants and collects mutation analysis data.

Our implementation that computes the control flow graph (CFG), for each method in our dataset, relies on the abstract syntax tree (AST) representation of Eclipse's JDT. Within such a computed CFG, each node consists of a single statement or a conditional expression (for branches). For simplicity, our implementation treats `try-catch` blocks as `if` statements. Some statements in a program do not belong to any explicitly declared method but rather a (static) initializer; in such cases, our implementation creates a separate CFG for the initializer, containing the set of sequential statements that exist in that initializer.

III. Evaluation

This section details the evaluation of our new approaches to mutant generation and mutant selection with respect to effectiveness and efficiency. We determine the effectiveness of our mutant generation approach by measuring how many mutants it generates that are *coupled* with real faults, following the approach of Just *et al.* [2].

First, we define two concepts. When T is a test suite that kills the mutant m while $T - \{t\}$ does not, then t is a *triggering test case* for m, and m is *coupled* to that test. When t detects the defect d, we further say that m is coupled to the defect d.

To perform this evaluation, we used Defects4J [18]. Defects4J contains 357 defects from 5 Java programs with the state of the code *before* and *after* a bug fixing commit. It additionally contains all the unit tests of that software project. The unit test(s) that were created or modified to detect the defect are triggering tests. Because some mutants trigger compiler bugs, we use 258 of these defects across all projects, spanning all projects.

Methodology First we generate all mutants for all possible program locations within the class(es) affected by the bug fixing commit. For the traditional mutation operators, we use the operations that are commonly used in the literature. Those are the mutation operators described in Just *et al.* [2]. A detailed list can be found in Table I. These traditional mutation operations are considered a sufficient set [19], [20]. Then, for each defect, using the Major framework [17] we run a full mutation analysis executing all the non-triggering tests, collecting information about each mutant (*i.e.* whether it lived or was killed by the tests). Then, we re-run a mutation analysis using only the triggering test(s). The mutants that were killed by the triggering test(s) but not from the non-triggering tests are the *coupled* mutants for that defect.

There are various potential threats to validity. First, Defect4J contains 5 Java programs and may not be fully representative of faults in other software systems. However, as the creators of the dataset argue, it contains a diverse set of software and defects. Additionally results using this data have been correlated with other datasets [2]. This gives us confidence that our results will generalize in other codebases. Another potential threat is the use of the specific subset of traditional operators. Although this subset can be enlarged, we follow Just *et al.* who argued that these traditional mutation operators are sufficient and effective [2].

Verifiability Our tailored mutation operators are implemented in the Major mutation framework [17] which is publicly released. Likewise, the Defects4J dataset is publicly available, allowing third-party verification of the results.

Metrics We are interested in the efficiency of the mutants. We measure efficiency of a set of mutants by the number of defects that have at least one mutant coupled with them. A mutant is coupled with a defect if it is killed by a triggering test but *not* by a non-triggering test. The coupling suggests that the given mutant is representative and relevant to the real defect.

We measure the effectiveness of mutants within different defect scopes. We define the scope of a defect as the part of the code where that the bug fixing commit made changes. In specific, we consider three increasingly narrower scopes:

- Class Scope: contains mutants only within the class(es) where the bug fixing commit affected.
- Method Scope: contains all mutants within the method(s) that the bug fixing commit affected.
- Line Scope: contains all mutants within the line(s) that the bug fixing commit changed.

The notion of defect scopes is important when measuring effectiveness. When a software engineer is interested in a specific unit (*e.g.* a method), she will use mutation testing only within the scope of that unit. As the defect scope gets narrower, the mutants should be increasingly relevant to the defect.

A. Mutant Effectiveness

In this section, we are interested in the effectiveness of the mutants and the mutation operators. Figure 2 presents the

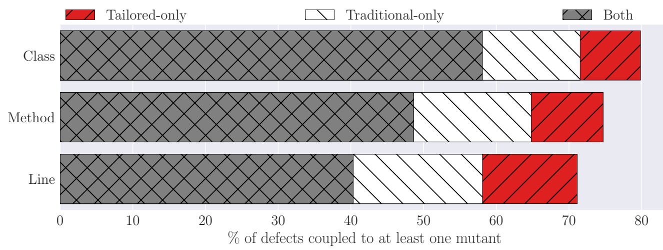


Fig. 2: The percent of Defect4J defects coupled with at least one mutant at each scope. About 22% of the defects are coupled with either *only* tailored mutants or *only* traditional mutants. This increases to 33% when considering only mutants at the line-scope.

defects that are coupled to at least one mutant at different scope. Thanks to the tailored mutants at class scope we can detect 8% more defects, compared to only using traditional mutants. This increases to 14% when considering mutants only within the line scope of each defect. The gray area suggests that a large number of tailored mutants are coupled with defects that are also coupled with traditional mutants. There is also a part of traditional mutants which are uniquely coupled with some defects. Thus, *tailored mutants increase the efficiency of mutation testing*.

Tailored mutants and traditional mutants allow us to apply mutations in different and distinct locations. In our data for all CFG nodes that are covered by at least one mutant, 69% of those locations are covered *only* by a tailored mutant, 17% *only* by a traditional mutant and the rest of the locations by both types of mutants.

We are also interested in the coupling of mutation operators with real defects. For a defect d, the set of coupled mutation operators is the set of mutation operators that can produce a mutant that is coupled to d. Additionally, we say that a mutation operator is *uniquely* coupled if it is the only operator that generates coupled mutants for a given defect. Table I presents the coupling statistics for the mutation operators. The variable name replacement mutation operator achieves the highest coupling and achieves equal unique coupling to the traditional relational operator replacement (ROR) operator. Tailored method replacement achieves the third best unique coupling. Table I also shows the average kill rate of the mutants produced by each operator, using only the non-triggering tests. Averages are computed per defect for each scope; we exclude from the average any operators that do not generate a mutant within a defect scope. If a mutation operator is representative for real defects, we expect as the defect scope gets narrower, the kill rate to decrease, implying a less tested region of code. Table I suggests that both tailored and traditional operators exhibit such behavior. Regarding the tailored mutation operators, the larger number of generated mutants in combination with a higher ratio of mutants detected by the non-triggering test suite suggests that tailored mutation operators are a noisier source, which reinforces the need for proper sampling approaches.

Qualitative Evaluation Tailored mutants mutate the code in ways that are in some cases more tailored and related the original defect. Figure 3 presents a sample of the tailored

Closure 111

```
- return topType;
+ return topType.isAllType()?
+   getNativeType(ARRAY_TYPE) : topType;
```

Mutations: `isAllType→isNominalType, isAllType→isNullType`

Math 34

```
- return chromosomes.iterator();
+ return getChromosomes().iterator();
```

Mutations: `getChromosomes→getChromosomeList`

Math 41

```
- for (int i = 0; i < weights.length; i++) {
+ for (int i = begin; i < begin + length; i++) {
```

Mutations: `begin→0`

Math 70

```
- return solve(min, max);
+ return solve(f, min, max);
```

Mutations: `min→max, max→min`

Math 75

```
- return getPct((Comparable<?>) v);
+ return getCumPct((Comparable<?>) v);
```

`getCumPct→getPct`

Time 21

```
- byNameKeyCache.put(setLoc[2],
+ byNameKeyCache.put(setEn[2],
  new String[] {setLoc[2], setLoc[1]});
```

Mutations: `setEn→setLoc, setLoc→setEn`

Fig. 3: Snippets of real faults and their fixes in Defect4J coupled *only* to tailored mutants. We also present sample of the tailored mutations that detect the fault. The formatting of some snippets has been changed to fit the page.

mutants coupled at the line scope where no traditional mutant was coupled within that scope. Variable name replacements (Math 70, Time 21 in Figure 3) capture confusions of variable usage, that sometimes have similar names and function. Method call replacement is able to introduce bugs that may arise from faulty usage of similar functions. All such mutations were not previously possible with the traditional set of mutation operators.

Figure 4 shows some examples of real faults that are not detected by either natural or traditional mutants, but could be detected by other unexplored kinds of tailored operators. Mutation operators that refine conditions with special cases (Closure 102), argument shuffling (Time 4), change of types when implicit type conversion may be happening (Math 30), statement location perturbation (Closure 102) and other complex pattern matching conditionals (Closure 112) are some of the classes where tailored mutation operators are needed to avoid an exponential explosion of mutants. Future research is required

TABLE I: Coupled and uniquely coupled mutation operators to defects at line scope (Ordered by the number of uniquely coupled defects). A (uniquely) coupled mutation operator is an operator that generates (uniquely) coupled mutants. The table shows the average number of mutants (per defect) and the percent of mutants that were killed by the non-triggering tests at each defect scope. When computing the averages for an operator, we ignore defects where the operator could not be applied within the defect scope.

Mutation Operator	Type	Total	Unique	Average Number of Mutants per Defect class	method	line	Average Kill Rate of Non-Triggering Tests class	method	line
Variable Name Replacement (VAR)	tailored	109	13	1512	184	48	77.5	69.0	52.7
Relational Operator Replacement (ROR)	traditional	76	13	127	18	6	62.6	48.7	40.8
Method Call Replacement (MCR)	tailored	67	8	592	65	24	86.3	72.3	54.5
Statement Deletion (STD)	traditional	28	7	38	5	2	81.2	71.0	38.5
Literal Value Replacement (LVR)	traditional	67	6	104	17	5	74.4	62.3	41.5
Conditional Operator Replacement (COR)	traditional	70	4	93	14	6	68.2	56.5	48.0
Natural Literal Replacement (NLR)	tailored	40	1	96	22	7	77.3	62.4	51.1
Arithmetic Operator Replacement (AOR)	traditional	25	0	128	39	17	79.1	65.9	61.5
Unary Operator Replacement (ORU)	traditional	3	0	6	3	3	88.6	77.9	50.0
Logic Operation Replacement (LOR)	traditional	1	0	10	6	2	79.6	92.9	0.0
Shift Operator Replacement (SOR)	traditional	1	0	5	3	3	65.8	75.0	33.3

Closure 102

```
- return len > 0;
+ return len > 0 && s.charAt(0) != '0';
```

Closure 102

```
- removeDuplicateDeclarations(root);
 if (MAKE_LOCAL_NAMES_UNIQUE) {
    [abbreviated]
 }
+ removeDuplicateDeclarations(root);
```

Closure 122

```
- if (comment.getValue().indexOf("/* @") != -1 ||
-    comment.getValue().indexOf("\n * @") != -1) {
+ Pattern p = Pattern.compile(
+    "(/|(\n[ \t]*))\\*[ \t]*@[a-zA-Z]");
+ if (p.matcher(comment.getValue()).find()) {
```

Math 30

```
- final int n1n2prod = n1 * n2;
+ final double n1n2prod = n1 * n2;
```

Time 4

```
- newPartial=new Partial(iChronology, newTypes, newValues);
+ newPartial=new Partial(newTypes, newValues, iChronology);
```

Fig. 4: Snippets of real faults and their fixes (in Defect4J) with no coupled mutants that could be detected by other kinds of tailored mutation operators. The formatting of some snippets has been changed to fit the page.

to investigate if code naturalness or other methods can provide a reasonable set of such operators.

The effectiveness of natural literal mutation Natural literal mutation approaches the problem of mutating variables and literals into other literals in a significantly different way compared to traditional literal value replacement operators. There is an unbounded number of literals to which a primitive variable or literal can be mutated. Natural literal replacement operators take a project-specific, context-specific approach to expand the traditional literal mutations by suggesting mutations that make the code similar to other code locations that have appeared within the project. The effectiveness of the natural literal replacement operation is indicated by the fact that 15.8% of defects are coupled to at least one natural literal mutation. This number increases to 22.9% and 28.9% when considering mutants within method and line scope.

For example, in Lang 41 the string literal "`[]`" is mutated to "`[`" within the code `arrayPrefix.append("[]");` because it has been seen within similar context. This mutation produces a mutant that is coupled with a real defect at the line scope. However, natural literal mutations have fewer uniquely coupled defects than the traditional literal mutation (Table I). We believe that this is because some contexts are relatively unique and infrequent within a single project and no alternative literals can be suggested. Investigating expanding literal mutations from a larger corpus of code is left for future work. Although literal mutations do not seem to be as effective on their own as traditional literal value replacements, they still seem to be a good complement for increasing the breadth of literal mutations and their overall efficiency.

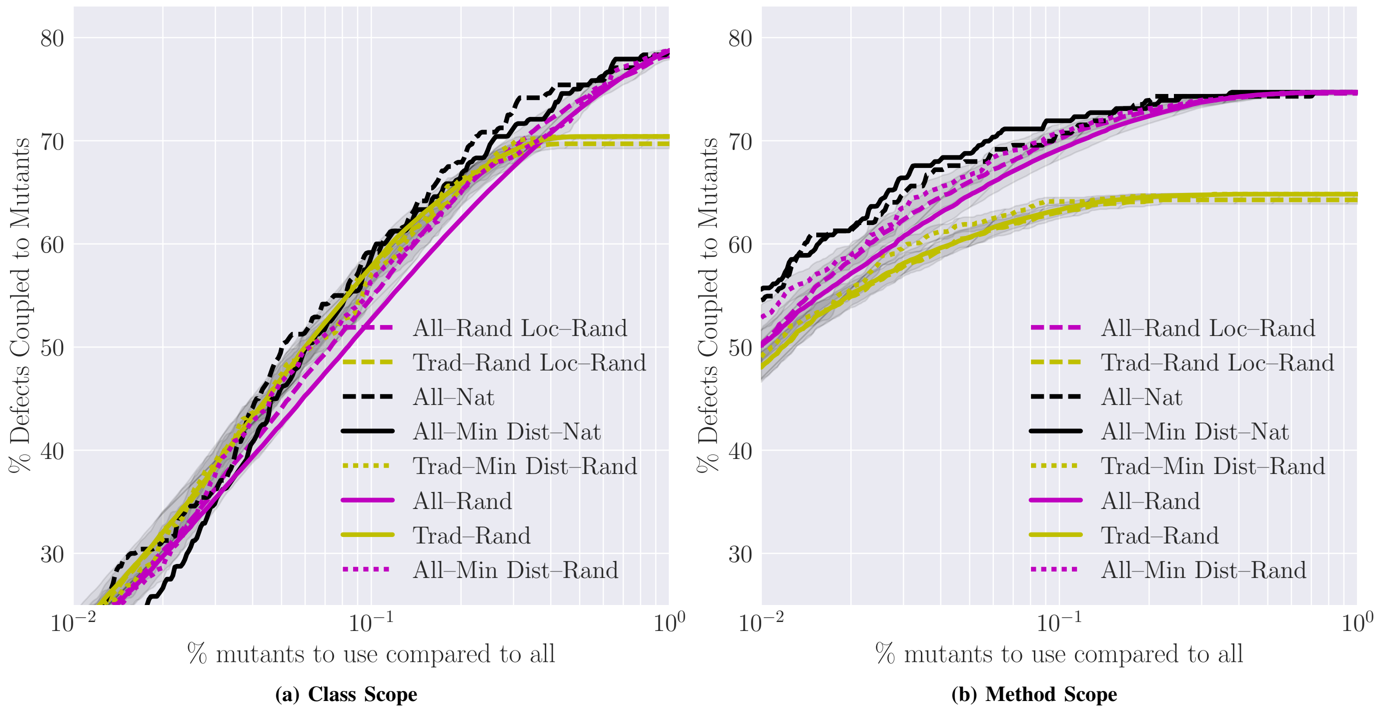
B. Mutant Selection

The increased number of mutants raises the problem of selecting mutation operations while maintaining the efficiency of the mutation analysis. This section evaluates various approaches for mutant selection. As a baseline we use a random selection policy, where we pick a mutant at random. In this case, the expected effectiveness can be computed analytically by calculating the probability that at least one coupled mutant has been killed by the triggering test(s), which is given by

$$P(\kappa,\lambda,|\mathbb{M}|) = \begin{cases} 1 & \text{if } |\mathbb{M}|-\kappa<\lambda \\ 1-\frac{(|\mathbb{M}|-\kappa)!(|\mathbb{M}|-\lambda)!}{|\mathbb{M}|!(|\mathbb{M}|-\kappa-\lambda)!} & \text{otherwise} \end{cases} \quad (5)$$

where $\mathbb{M}$ is the set of all available mutants in the scope, $\kappa \leq |\mathbb{M}|$ is the number of mutants we are currently selecting and λ is the number of mutants in $\mathbb{M}$ that are killed only by the triggering test (*i.e.* are coupled mutants).

Figure 5 plots the effectiveness of the mutants as more mutants are selected with different selection policies. As



(a) Class Scope

(b) Method Scope

Fig. 5: Mutant effectiveness of traditional ("Trad") and all ("All") mutants. The *x* axis (in log-scale) is the mutant budget (in relative terms to the total number of tailored and traditional mutants of each defect) that is executed during mutation analysis. One may notice that when we use both traditional and tailored mutants at the class scope, the effectiveness is increased but with the extra cost of executing more mutants. However, when limiting ourselves to the method scope of a defect, including tailored mutants always helps. The graphs show different methods for selecting mutants. For the selection processes that contain stochasticity ("Rand"), the shaded region around each line indicate one standard deviation from the expected efficiency. Selection processes that use our submodular objective are denoted by "Min Dist". We do *not* show results for line defect scope since selection methods behave in a very similar way because there are few mutants within a given line of code.

previously discussed, selecting both tailored and traditional mutants achieves the highest efficiency (*y*-axis). However, this requires to incur the cost of potentially executing more mutants. Figure 5a suggests that for a small numbers of mutants — within class scope — just using traditional mutants is the most efficient possible choice. However, as the budget of mutants to execute becomes larger, using both tailored and traditional mutants achieves the best effectiveness. This result holds *only* when considering class-level defect scope. When we narrow the scope (*i.e.* consider mutants only within the method or line scope) a naïve random selection of both tailored and traditional mutants seems (Figure 5b for method scope, but the results are similar within line scope) to always provide better effectiveness, regardless of the mutant selection method. This results suggest an actionable recommendation for practitioners using mutation analysis. *When interested in a single method (or line) using both tailored and traditional mutants provides better efficiency irrespectively of the number of mutants that one may wish to execute.*

Importance of Location in Mutation Testing Mutation location plays an important role in mutation testing, since it is correlated with various coverage metrics. In the Defects4J dataset, we find that the mutants are *not* evenly distributed across locations in the control flow graph (CFG) of each program. Figure 6 shows the distribution of the number of mutants per CFG node. We find that some CFG nodes have hundreds of mutants, while others have only a few. We further

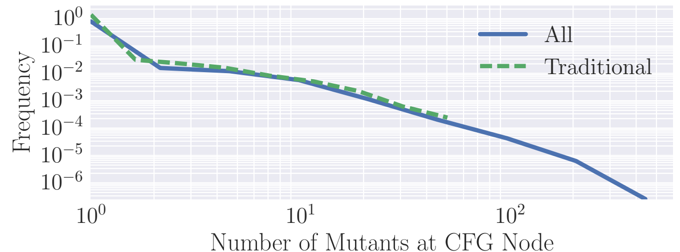


Fig. 6: The distribution of number of mutants per control flow graph (CFG) node. Each node in the CFG is either an executable or non-executable statement. About 85% of the CFG nodes are associated with at least one mutant. The graph shows a heavy-tailed distribution of mutants to locations, showing that few nodes have hundreds of mutants, while most of the nodes have only a few. Note the log-log scale, which is customarily used to plot such distributions.

note that although we have introduced the tailored mutants, still about 15% of the CFG nodes are *not* associated with any mutant.

Comparing Effectiveness of Selection Policies In Figure 5, we plot the percent of mutants that one can use (*x*-axis) compared to using all the available mutants *vs.* the percent of bugs that are coupled with at least one of the selected mutants. The *x*-axis (in log scale), essentially can be interpreted as the budget of mutant executions that one may be willing to incur during mutation analysis.

Figure 5 show two trends. First, location-based methods perform better than a simple baseline random selection. Additionally, there is an improvement when using our CFG distance

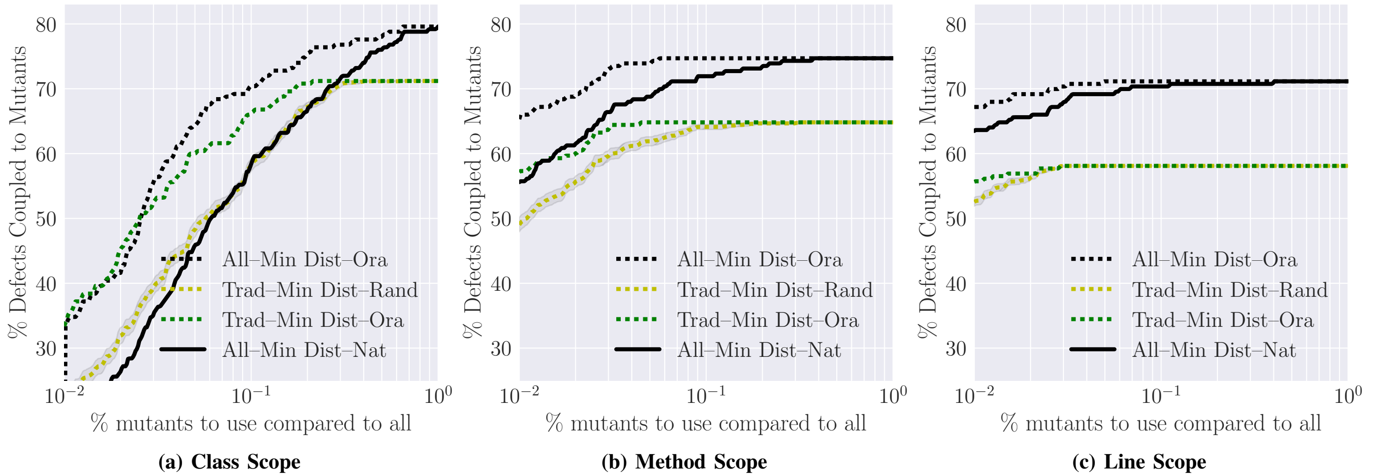


Fig. 7: Mutant effectiveness of traditional ("Trad") and all ("All") mutants for a given budget. The *x* axis (in log-scale) is the budget (in relative terms compared to using all tailored and traditional mutants of a defect) that are executed during mutation analysis, as in Figure 6. The graphs show the performance of the location-based selection for traditional and all mutants when selecting mutants with out optimization-based approach that minimized Equation 2. For each defect scope, we show the performance of the method when using code unnaturalness *vs.* using an oracle to select the optimal mutant at each location. The oracle is the upper bound that the CFG distance minimization can achieve.

optimization and our naturalness-based scoring, compared to a random selection for a given location. The differences seem to diminish when considering only line scope (not shown). This is reasonable since the number of locations within a line are quite limited (most usually we would have exactly one location). We also observe that our selection method achieves better results within a method scope. This implies that within the class scope we have a suboptimal selection method (Equation 2 implies that methods with more CFG nodes are prioritized) and that further research is required for selecting the methods first.

The goal of selecting mutants is to achieve a similar efficiency with a smaller number of mutants. Our selection methods achieve this. For example, in Figure 5b to achieve 70% of efficiency, one needs to use both tailored and traditional mutants. If a developer resorts to simple random selection, then she will need to execute 13% of all mutants. In contrast, only 6% of all mutants are needed when preferring the mutants returned by our selection method that combines our location optimization and naturalness-based selection, which reduces the number of mutants that need to be executed by about 50%. Similarly, for an effectiveness of 60% our selection method requires about half the mutants compared to a naïve random approach. These results suggest that location and naturalness-based selection of mutants is a promising approach for reducing the number of mutants required to achieve a given effectiveness.

The upper bound of location-based selection Figure 7 plots the effectiveness of the best selection processes previously discussed. The graphs also include an "oracle" selection process. This refers to a selection process, where we first pick a location optimizing the objective in Equation 2 and then at each location an oracle picks the best possible mutant (*i.e.* if a coupled mutant exists, it is picked. Otherwise a random non-coupled mutant is selected). Obviously, this is *not* a real-life option but it provides an upper bound on the efficiency of our location-based optimization selection method. The gap seen in Figure 7 suggests that improvements in the mutant selection at a given location can yield substantial improvements to the efficiency of the location-based methods. It also suggests that source code naturalness is only one indicator of the effectiveness of a mutant. We believe that this has to do with two factors. First, simple errors that may be captured by a local model (such as the *n*-gram) are most often easily removed before the code is even committed or tested. The remaining bugs, should contain more semantics-related which is harder to capture with an *n*-gram LM. Second, it could be that a large number of defects that are too unnatural represent bugs that are caught early in the development process.

IV. Related Work

Mutation Analysis The huge computational cost of mutation analysis is a widely acknowledged and studied problem. Selective mutation testing, first studied by Mathur [21], aims to select a subset of all mutants that is representative of the entire set. A large body of related work primarily focused on the determination of sufficient mutation operators in an attempt to reduce the number of generated mutants. Wong and Mathur [22] found that mutants generated with only 2 out of 22 mutation operators achieved effectiveness results that are comparable to the full set of mutants. Offutt *et al.* [19] determined a sufficient set of 5 mutation operators that only incurred a negligible loss in effectiveness in their experiments. Similarly, Namin *et al.* [20] analyzed an exhaustive set of 108 mutation operators and concluded that 28 are sufficient. More recently, however, Zhang *et al.* [4] showed that random selection of mutants is as effective as operator-based mutant selection, and a study by Kurtz *et al.* [6] suggests that there exists no subset of mutation operators that can generate effective mutants for all programs.

Our work is motivated by recent findings and differs from the large body of related work in two ways. First, rather than sampling from a vast, highly redundant pool of mutants, our approach starts with a small set of provably effective and

non-redundant [23], [24] mutation operators, and augments it with mutation operators tailored towards the program under test. Second, prior work has evaluated the effectiveness of a sampling approach exclusively on mutants whereas our methodology considers the coupling of mutants with real faults.

Prior work has intensively addressed the scalability problems of mutation analysis. In contrast, the effectiveness, *i.e.*, the correlation between mutant detection and real fault detection, is less studied. Andrews *et al.* [1] and Just *et al.* [2] showed that such a correlation indeed exists but the latter study also showed that the set of commonly used mutation operators is not sufficient—27% of real faults are not coupled to the mutants they generate. Our work aims to close this gap by providing tailored mutants that are coupled to more real faults.

Code Naturalness Recently, machine learning and NLP methods have been applied to source code text with fruitful applications. Originally, Hindle *et al.* [9] showed that source code is predictable and "natural", by using *n*-gram LMs. Later, others [25], [26], [27], [28] made improvements to the language models of code. Applications of these models include source code autocompletion, learning coding conventions and suggesting names [15], [16], predicting program properties [29], extracting code idioms [30], code migration [31], [32] and code search [33]. Related to our work is the work of Cambell *et al.* [34] and Bhatia and Singh [35] that provide better localization of compile-time syntax-only errors using code language models. Recently, Ray *et al.* [7] showed that buggy code has on average lower entropy scores assigned by cache *n*-gram models.

V. Conclusion

In this work, we introduce the idea of tailored mutation operators, which extend traditionally used mutation operators by adapting the code transformations they produce using information from elsewhere in the project. We presented examples of tailored mutation operators that replace variable names, method calls and literals. These new operators increase the coupling of mutants to real defects compared to traditional operators. Introducing tailored mutants increases effectiveness but increases the number of mutants that need to be executed. To alleviate this problem, we present new mutant selection methods. We select program locations within a control flow graph first by employing submodular optimization, a well studied theoretical framework for optimization over sets, and then we select mutants within that location, based on the naturalness of the mutation, following recent work that shows that buggy code is likely to be *un*-natural [7].

Promising future directions include the exploration of alternative methods for selecting mutants within a given program location. Additionally, using more advanced models of code naturalness may improve upon the performance of the *n*-gram language model. Modification to the location selection method and its submodular optimization approach, *e.g.* using machine learning, may provide additional improvements.

Acknowledgments

This work was supported by Microsoft Research through its PhD Scholarship Programme. Charles Sutton and Miltiadis Allamanis were supported by the Engineering and Physical Sciences Research Council [grant number EP/K024043/1].

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 402–411.

[2] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 654–665.

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

[4] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, pp. 435–444.

[5] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 511–522.

[6] R. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gokce, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016 (to appear).

[7] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," *ICSE*, 2016.

[8] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2014, pp. 189–200.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[10] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[11] F. Bach, "Learning with submodular functions: A convex optimization perspective," *Foundations and Trends in Machine Learning*, vol. 6, no. 2-3, pp. 145–373, 2013.

[12] A. Krause, A. Singh, and C. Guestrin, "Near-optimal sensor placements in gaussian processes: Theory, efficient algorithms and empirical studies," *Journal of Machine Learning Research (JMLR)*, vol. 9, pp. 235–284, 2008.

[13] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 137–146.

[14] A. Krause and D. Golovin, "Submodular function maximization," in *Tractability: Practical Approaches to Hard Problems*, L. Bordeaux, Y. Hamadi, and P. Kohli, Eds. Cambridge University Press, 2014.

[15] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[16] ——, "Suggesting accurate method and class names," in *Symposium on the Foundations of Software Engineering (FSE)*, 2015.

[17] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[18] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 437–440.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[20] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 351–360.

[21] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, 1991, pp. 604–605.

[22] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[23] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[24] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability (STVR)*, vol. 25, no. 5-7, pp. 490–507, 2015.

[25] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, 2013, pp. 207–216.

[26] C. J. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning (ICML)*, 2014.

[27] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 532–542.

[28] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 269–280.

[29] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015, pp. 111–124.

[30] M. Allamanis and C. Sutton, "Mining Idioms from Source Code," in *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[31] S. Karaivanov, V. Raychev, and M. T. Vechev, "Phrase-based statistical translation of programming languages," in *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 173–184. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661148

[32] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Migrating code with statistical machine translation," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 544–547.

[33] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of The 32nd International Conference on Machine Learning*, 2015, pp. 2123–2132.

[34] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 252–261.

[35] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *arXiv preprint arXiv:1603.06129*, 2016.