Using a Probabilistic Model to Predict Bug Fixes

Mauricio Soto
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
mauriciosoto@cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
clegoues@cs.cmu.edu

***Abstract*—Automatic Software Repair (APR) has significant potential to reduce software maintenance costs by reducing the human effort required to localize and fix bugs. State-of-the-art generate-and-validate APR techniques select between and instantiate various mutation operators to construct candidate patches, informed largely by heuristic probability distributions. This may reduce effectiveness in terms of both efficiency and output quality. In practice, human developers have many options in terms of how to edit code to fix bugs, some of which are far more common than others (e.g., deleting a line of code is more common than adding a new class). We mined the most recent 100 bug-fixing commits from each of the 500 most popular Java projects in GitHub (the largest dataset to date) to create a probabilistic model describing edit distributions. We categorize, compare and evaluate the different mutation operators used in state-of-the-art approaches. We find that a probabilistic model-based APR approach patches bugs more quickly in the majority of bugs studied, and that the resulting patches are of higher quality than those produced by previous approaches. Finally, we mine association rules for multi-edit source code changes, an understudied but important problem. We validate the association rules by analyzing how much of our corpus can be built from them. Our evaluation indicates that 84.6% of the multi-edit patches from the corpus can be built from the association rules, while maintaining 90% confidence.**

I. Introduction

Repairing bugs is one of the most resource-intensive tasks in software development [4], [38], [42]. Significant recent research effort has been dedicated to building automatic program repair (APR) tools that are able to address bugs in programs (e.g. [6], [18], [22], [23], [31], [39]–[41]). One well-known class of repair techniques follows a *generate-and-validate* approach (e.g., GenProg [40], Par [18], TrpAutoRepair [32], Prophet [23], HDRepair [5], Angelix [27], Nopol [43]), which takes as input a test suite, including at least one failing test case exposing a defect, and source code to be modified. These approaches then *generate* a large patch candidate space by applying mutation operators to the original source code and *validate* each by running the potentially patched program against the test suite, seeking a candidate that leads the program to pass all tests.

One key reason this is challenging is that the search space of possible edits that can be applied to the original source code is infinite. To tackle this problem, techniques limit themselves to a set of possible change types, which we refer to broadly as "Mutation operators". There is a broad diversity of such operators used in automatic program repair, including deleting or inserting statements [40], applying templates [18], transformation schemas [22], [23], or semantically-inferred code [27], [29], [43]. Given a potentially-faulty location (typically identified using off-the-shelf fault localization, e.g., Tarantula [14]), these approaches then use heuristics or heuristically-informed probability distributions to select between mutation operators to construct candidate patches. Even with these efforts, the search space for possible edits remains vast; not all bugs can be repaired; and many produced patches are of low quality.

Our intuition is that, in bug-fixing reality, human developers use certain mutation operators much more frequently than others (e.g., deleting a line is much more common than creating a new class), and we can use that knowledge to drive our search. In this paper, we therefore study and then simulate the behavior of human developers to create patches. Our key supposition is that our approach can support a richer search space constructed via more expressive mutation operators that are more likely to produce high-quality patches. Indeed, this idea has been leveraged manually in the past to create more human-acceptable mutation operators [18] and to inform patch *ranking* (rather than construction) [5], [23].

We mine bug fixing commits from the 500 most popular GitHub Java projects to model the selection probability of the possible mutation operators based on empirical data that describes how human programmers fix their code. We thus categorize, compare, and validate a superset of mutation operators in use in a number of state-of-the-art approaches [18], [23], [40], [41]. We then use this model to inform a repair approach that chooses from the set of possible operators based on these real-world probabilities. As a result, our work goes beyond prior work that leverages human bug fixes in a program repair context [5], [18], [23] by generalizing to a broader and more expressive set of mutation operators, and using a fully-automatically-mined model much earlier in the APR process, when patch candidates are actually created. Furthermore, we propose and initially validate a new approach for modeling multi-edit repairs based on mining previous rules from historical edit data, predicting from a given set of edits which operation should be applied next. This serves as a first step towards scalable traversal of the multi-edit patch space.

We evaluate the predictive power of our mined model on its own, in terms of its accuracy in predicting the operators used in real-world bug fixes. We demonstrate this algorithm both in terms of speed and quality with a full set of mutation

operators on a subset of real-world single-line defects [15], and we validate the quality of the patches found by our approach in comparison to several previous state-of-the-art techniques.

The primary contributions of this paper thus are:

- **Empirical model of single-edit repairs to Java code** mined from the 100 most recent bug fixes from the 500 most popular Java projects on GitHub. This study also provides a deeper understanding of which mutation operators are used to fix errors in source code, and the frequency of the mutation operators analyzed.
- **New Repair Approach**, which uses mutation operators from multiple state-of-the-art approaches and the above-mentioned models to choose between them, favoring those more commonly used by human developers. We validate the use of this model, as well as a good set of operators to use for expressive, high-quality repair.
- **Development and integration** of this approach for the Java programming language in a publicly-available tool that can apply the several different mutation operators, and the probabilistic model discussed in this study.
- **Comparative evaluation** of the new repair approach on real world bugs from well-known open source projects (independent of those used to inform model construction), as compared to several previous techniques. We independently evaluate patch quality using a held-out test suite. We conclude a technique based on a model informed by human behavior, finds patches more quickly than prior techniques that do not; that the patches are of comparatively higher quality; and that patch creation benefits from a diverse set of candidate mutation operators.
- **Multi-edit repair approach** A proposed technique, and an initial validation thereof, to construct bug-fixing patches requiring several mutations by mining and then applying association rules from a large set of historical bug fixes.
- **Code and data** that produced and were produced by our experiments, available at https://github.com/squaresLab/ProbabilisticModelSaner2018, to support reproducibility and extension.

The rest of this paper proceeds as follows: We first outline generate-and-validate repair and, in particular, categorize the mutation operators used in state-of-the-art approaches (Section II). Section III describes our approach to building probabilistic models of human edits for both single- and multi-edit bug fixes. Section IV describes our evaluation. Section VI describes related work; Section VII concludes.

II. Generate-and-validate repair and operators

In this section, we outline generate and validate repair at a high level (Section II-A), to provide the necessary background to understand this study.[1] We then detail and categorize the mutation operators used in prior approaches for syntactic program repair and that we model, mine, and leverage in our approach (Section II-B).

[1]We focus on relevant background in this section, and describe other techniques for automatic program repair in Section VI.

A. Paradigm

Figure 1 outlines the general paradigm for generate-and-validate repair. This approach begins with two elements: buggy source code and a test suite. The test suite is comprised of passing test cases to describe the correct program behavior, and failing test cases to expose the behavior to be repaired.

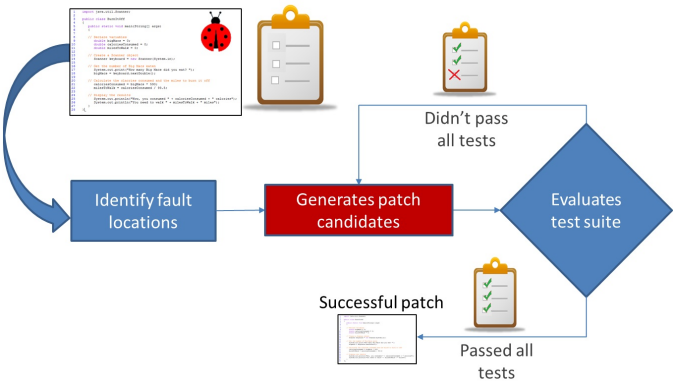

Fig. 1. Generate-and-validate repair approach.

A generic generate-and-validate APR approach first localizes the error to a smaller set of candidate program statements (typically using an off-the-shelf statistical approach [14]). It then constructs one or more candidate patches that seek to fix the buggy behavior without breaking any previously-correct behavior. Techniques vary in (1) the mutation operators that they consider, (2) how they select between those operators, and how they identify or construct the fix code used to instantiate them, and (3) how they traverse the search space otherwise. Most mutation operators (with the exception of deletion) require new code to fill in "holes." For example, an operator that replaces one statement with another must select replacement code; an operator that wraps a statement in a null check must select the object to be checked against null. Syntactic approaches (e.g., [5], [18], [21], [23]) typically select this code from within the same function, module, file, or program. Semantic approaches instead use synthesis to construct fix code. In both cases, heuristics inform code selection. For example, some syntactic approaches choose between fix code uniformly at random or using other set probability distributions [18], [21]; others used learned models to do so [23]. Semantic approaches use heuristics to decide which program components should be made available to the synthesis engine [27], [43].

Once constructed of one or more such mutations, each candidate patch is applied to the input program, which is then run on one or more of the input test cases for evaluation. If a patch leads the program to pass all the test cases in the test suite, including those originally witnessing the bug, it is presented as a likely repair for the bug. If not, the APR algorithm typically creates more candidate patches until it reaches a pre-defined resource limit. Here, too, search strategies vary: some techniques are explicitly single-edit [32], [41], [43];

others use random search strategies, like a random walk [6] or genetic programming [5], [18], [40].

Patch quality is an important concern in APR research [33]: Ideally, created patches will generalize beyond the test cases used to inform their construction [36], and conform to other non-functional standards of acceptability [9], [18]. Although assessing patch quality is an unsolved research problem [28], one mechanism for objectively evaluating functional patch correctness is to evaluate generated patches on a held-out test suite [21], [36], separate from the tests used to construct the patches. If the generated patch allows the program to pass all held-out tests, it is more likely to *generalize* to the desired but unwritten specification. If the patched program fails any held-out tests, the patch is said to *overfit* to the test suite that guided the repair.

B. Generate-and-validate mutation operators

We categorize mutation operators used from a cross section of state-of-the-art approaches into two groups:

a) Statement-Edit mutations: One family of repair approaches, including GenProg [40], TrpAutoRepair [32], and AE [41], creates candidate patches by applying coarse-grained mutation operators (e.g. *append*, *delete*, or *replace*) at the statement level. These prior techniques historically target the C programming language, where a statement is a grammar nonterminal corresponding intuitively to blocks, simple statements that terminate with a semicolon, or compound statements corresponding to control flow or loops. In Java, statements conceptually map to similar program elements, e.g. blocks, while loops, or single-line method calls. In these approaches, the statements being appended or replaced typically come from within the project being modified. This is grounded in the notion that source code has a high level of redundancy [12].

b) Template-based mutations: Another family of approaches instantiates predetermined templates, more complex than those in the first family, at applicable code locations. This family includes PAR [18], SPR [22], and Prophet [23].

PAR is the product of a study of a large number of human created patches, from which human annotators abstracted 10 different templates to cover the most commonly-used changes in bug-fixing practice. The 10 considered templates are detailed in the top section of Figure 2. In the interest of completeness, we also include six extra templates mentioned on the PAR website.[2] These extra templates provide new mutation operators drawn from human edits, that help us compare to and generalize the other approaches; they are shown in the middle segment of Figure 2. SPR and Prophet use a set of transformation schemas, shown in the bottom section of Figure 2.

The SPR/Prophet transformation schema can be mapped to certain PAR templates. For example, *Condition Introduction* can be seen as a superset of *Range Checker*, *Collection Size Checker*, *Class Cast Checker*, and *Null Checker*. *Condition Refinement* includes *Expression Adder and Remover*. *Insert Initialization* can be generalized from *Object Initializer*, *Upper Bound Setter* and *Lower Bound Setter*; *Conditional Control Flow Introduction* can be seen as a subset of *Sequence Exchanger*; *Value Replacement* can be seen as a superset of *Method Replacer*, *Parameter Replacer*, *Castee Mutator* and *Expression Changer*; and *Copy and Replace* can be matched to *Expression Adder*. These operators similarly generalize those used in semantics-based approaches, which replace expressions used either in conditions or on the right-hand-side of assignments (the operators are the same; the difference lies in how the fix code is selected/constructed).

[2]https://sites.google.com/site/autofixhkust/home/fix-templates

PAR fix templates	
Null Checker	Parameter Adder and Remover
Parameter Replacer	Expression Adder and Remover
Method Replacer	Collection Size Checker
Expression Replacer	Range Checker
Object Initializer	Class Cast Checker
PAR "extra" templates	
Caster Mutator	Lower Bound Setter
Castee Mutator	Upper Bound Setter
Sequence Exchanger	Off-by-one Mutator
SPR transformation schema	
Condition Refinement	Insert Initialization
Copy and Replace	Condition Control Flow Introduction
Value Replacement	Condition Introduction

Fig. 2. (Top) PAR fix templates. (Middle) PAR "extra" templates. (Bottom) SPR transformation schemas. We use the templates in the top and middle portions of this table as representative of the class of Template-based mutations.

The templates used in the program modification tool Kali [33] also correspond to subsets of certain PAR templates or their extensions. For example, *Redirect Branch* can be seen as a subset of *Expression Changer*, and *Insert Return* and *Remove Statement* are subsets of *Expression Adder and Remover* accordingly. Similarly, many other operators from the field of mutation testing [30], as used in APR [5], [6] can be seen as subsets of the extensions of the PAR templates.

To summarize, these approaches have significant similarities between them. We use the PAR templates to represent this category because PAR (1) broadly includes the other techniques' mutation operators, (2) provides a concrete description of how the code is changed, enabling replication, and (3) explicitly targets Java (SPR, Prophet and Kali target C), reducing the extent to which we must apply subjective judgment to re-implement and use in our context.

III. Program repair via a probabilistic edit model

In this section, we describe how we mine a model of human bug-fixing edits from a large set of popular Java projects. The intuition is to use this model to apply human knowledge to the automatic program repair process, creating patches inspired by what human developers do; the model is used explicitly in the patch creation step of a generate-and-validate repair process. To do this, we select a corpus of popular GitHub projects and identify their most recent bug fixing commits (Section III-A).

We identify mutation operator and replacement incidence in this dataset (Section III-B) to construct a two level probabilistic model used in a novel repair technique (Section III-C). Finally, we analyze the corpus to extract association rules of edits that happen together often, to potentially inform multi-edit source code repairs (Section III-D).

A. Selecting the corpus

We cloned the 500 most-starred Java projects on GitHub as of August 2016 and identified the most recent 100 bug fixing commits per each project. If the project had fewer than 100 bug fixing commits, we analyzed as many as found. Identifying such commits is a difficult problem [3]. We apply a regular expression to each commit message that looks for words such as *"fix", "bug", "issue", "problem",* etc. following guidelines from prior work [35]. We further only include commits that exclusively modify Java source code, since we focus on such bugs. We restrict attention to commits that modify a maximum of three files to exclude big merges, and because large commits are more likely to include changes unrelated to a bug fix [11], [16]. To our knowledge, this is the largest set of bug-fixing commits mined to inform program repair to date.

B. Identifying mutations in developer commits

For each considered commit, we refer to the code before the fix as the *"before-fix"* version and the code after as the *"after-fix"* version. We seek to identify the changes performed between the before- and after-fix versions, match them to our considered mutation operators (Section II-B), and count how many times each operator is used in the edits in our corpus. We used Gumtree [7], a source code tree differencing framework to identify deletions and insertions, as augmented by a component of QACrashFix [10], which allows it to more accurately account for replacements. These tools create an AST representation of each program file, both before- and after-fix, and produce a set of changes performed between them.

The changes output by these tools do not all directly map to the template- and statement-based operators we consider. That is, there is no one-to-one correspondence between the list of changes and the mutation operator identification. We thus greedily attempt to match the identified changes to the studied mutation operators. We seek each of the mutation operators that can match a given set of edits. For example, to identify a *Null Checker* application, for each action describing a commit, we check if the manipulated node is an *IfStatement*. If so, we check whether the action is a node insertion. If so, we check if the condition in the inserted *IfStatement* is an *InfixExpression* that compares an *Expression* to a *NullLiteral*. If so, we count this sequence of actions as an instance of a *Null Checker* mutation operator. We created such an automated procedure for all the mutation operators. These strategies are necessarily heuristic, and we do not claim perfect soundness in our matching, instead aggregating results over a large dataset.

C. APR using a two-level probabilistic model

We propose a novel syntactic generate-and-validate repair technique that differs from prior work first, in the range of mutation operators considered (Section II-B) and second, in how it chooses between those operators and instantiates them. We instantiate this technique by extending an open-source implementation of GenProg [40] for Java.[3] We add the newly-considered mutation operators, and a mechanism that allows the tool to select between the mutation operators according to the probabilities described by a model.

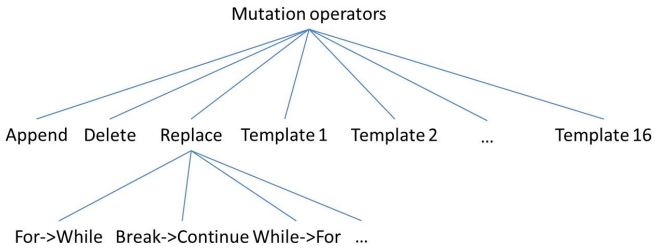

Fig. 3. Two level probabilistic model to inform operator selection and instantiation.

Our approach uses a two-level model (Figure 3) in operator selection/instantiation. The first level informs the selection of the given mutation operator, from a set of legal operators at a given potentially-faulty location (e.g. *Parameter replacer* cannot be applied to a *BreakStatement*). If the operator selected is *replace*, the second level informs the selection of the replacement code. To build both models, we perform an incidence count of each mutation operator and replacement observed in our dataset, matched as described in Section III-B, and then apply Laplace smoothing [34] with $\alpha = 1$ to account for 0 occurrences. These two models in detail are as follows:

1) Mutation operator probabilistic model: The *Mutation operator probabilistic model* describes the probabilities of choosing between the several different mutation operators at a particular fault location. To build this model, we count the incidence of each mutation operator observed in our dataset, matched as described in Section III-B. Mutation operator probability is then weighted accordingly. Figure 4 describes the distribution of mutation operators as mined from the corpus. Template-Based and Statement-Edit mutations contribute 29.26% and 70.74% of the studied edits, respectively.

2) Replacements probabilistic model: If the "Replacement" mutation operator is selected, the *Replacements probabilistic model* describes the probability of replacing one statement (*"replacee"*) with another (*"replacer"*), thus informing the selection of replacement fix code. We consider the 22 different types detailed by Eclipse JDT as direct subclasses of the class `Statement`, and the incidence with which each replaces another. For example: "What is the observed incidence of a `For` loop replacing a `While` loop?" Given 22 statement types, there are 484 possible combinations. Note that the observed probabilities are not reciprocal, e.g. that the probability of a `For` loop replacing a `While` loop is different from the probability of a `While` loop replacing a `For` loop. This model

[3]https://github.com/squaresLab/genprog4java

Mutation operator	Edits found (%)
Append	61.03
Sequence Exchance	15.76
Delete	9.10
Param Replacer	5.93
Param Add/Rem	3.15
Expression Repl	1.28
Method Replacer	1.12
Null Check	0.76
Castee Mutator	0.61
Replacement	0.60
Expression Add/Rem	0.37
Cast Check	0.11
Size Check	0.07
Range Check	0.04
Object Initializer	0.03
Caster Mutator	0.03
Lower Bound Set	0.01
Upper Bound Set	0.00
Off by One	0.00

Fig. 4. Mined distribution of mutation operators.

is built analogously to the mutation operator model, based on replacer/replacee statement incidence.

D. Multiple edit association rule mining

Although single-edit patches can repair many non-trivial bugs in real software, the majority of bug fixes in real software require multiple edits [37], [44]. The number of combinations of possible mutation operators to apply in a sequence increases exponentially with number of combined source code changes. As a first step towards mitigating this limitation, we propose an initial analysis of multi-edit source code changes by mining a more expressive model of common changes. In particular, we extract *association rules* to model chains of several edits, capturing the way humans create these kinds of fixes.

Association rules are if/then statements that show relationships between elements in a dataset which happen frequently together. We use the well-known association rule mining algorithm Apriori [1]. We mine association rules for the Mutation operators model (Section III-C) by analyzing mutation operator count in the studied commits. We develop rule sets at different *Confidence* levels, defined as:

$$conf(X \implies Y) = \frac{supp(X \cup Y)}{supp(X)}$$

Where X and Y are items in a transaction (mutation operators, in our context). Confidence is calculated according to its Support (*supp*), an indication of how frequently the set of mutation operators (item set) occurs in the corpus. Formally:

$$supp(X) = \frac{|\{t \in T; X \subseteq t\}|}{|T|}$$

Where X is the item set and t is each individual transaction in the database of transactions *T*. Apriori identifies the mutation operators that frequently happen together in a set of commits, iteratively extending them to larger item sets that appear often in the transactions as identified by these metrics.

IV. Evaluation

We evaluate our model both independently and as part of an automatic repair technique. We have four research questions:

- **RQ1:** How accurate are our mined models in predicting mutation operators and replacement code across a large dataset? (Section IV-A)
- **RQ2:** Which syntactic operators and selection models are most useful for repair? (Section IV-C)
- **RQ3:** How does our model-informed APR tool compare to the state-of-the-art in APR? (Section IV-D)
- **RQ4:** What are the most common multi-edit modification rules in practice? (Section IV-E)

All experiments are performed on a server consisting of 16 processors Intel(R) Xeon(R) CPU E5-2699 v3, with 2.30 GHz each processor, 46080 KB cache each, and 32 GB RAM memory, operating system Ubuntu 14.04.5 LTS. In addition to addressing our research questions as described in the above roadmap, we provide additional detail on the experimental setup for all repair experiments in Section IV-B.

A. Model generalization

	Replacements			Mutation operators		
Fold	#	Prob (%)	Eq (%)	#	Prob(%)	Eq (%)
1	155	70	25	17313	95	64
2	86	100	25	17232	93	63
3	81	82	23	16754	95	81
4	101	81	40	14159	94	9
5	99	90	27	17022	95	2
6	75	84	21	14945	95	58
7	79	86	24	12901	93	7
8	55	96	23	10552	95	27
9	82	88	8	14568	93	62
10	116	100	31	19140	95	6
Mean	92.9	87.7	24.7	15458.6	94.3	37.9
σ	27.4	9.3	8.0	2530.3	0.9	30.5

Fig. 5. Correctly predicted edits, across 10 folds of data. The replacement probabilistic model (center columns, left) predicts the correct replacement statement 87% of the time, an improvement of almost a factor of 4 over the equally-distributed equivalent. The mutations model (right columns, left) outperforms the equally-distributed equivalent by almost a factor of 3.

We begin by independently evaluating our models' (Section III-C) predictive accuracy, to validate the underlying intuition. We include all Statement-edit and Template-based mutations, as described in Section II-B.

1) Setup: Our high-level research question concerns whether our mined models are likely to identify the correct fix code across a large dataset, suggesting their potential utility in a repair context. We therefore compare the accuracy of the models to an *equally distributed* baseline, which selects a mutation (or replacement) uniformly at random. This is to contrast the accuracy of the priorities assigned by our model to the way

TABLE I
STUDIED SUBJECTS FROM DEFECTS4J.

Project	ID	Test cases	Defects considered
JFreechart	Chart	2205	4
Closure compiler	Closure	7927	25
Apache commons-lang	Lang	2245	13
Apache commons-math	Math	3602	18
Joda-Time	Time	4130	3

it is performed by current approaches, which pick mutation operators either uniformly at random, or using coarse grained heuristics [18], [21], [32], [41]. We measure how often each model correctly predicts the mutation operator (or replacer statement, for a given replacee statement), within the top 5 produced choices. Perfect accuracy (correctness in the Top-1 most likely choice) is unnecessary, since an APR approach can iterate through several different candidates before a fix; however, the noisy search problem suggests that a relatively tight bound (top-5) is likely appropriate.

To illustrate, consider a simple example. Assume the following simple bug-fixing patch from our corpus:

```
1 + if(i > l.size()) {
2       return l.get(i); //original faulty location
3 +   }
```

The developer has replaced the original `return` with an `if-then` statement that wraps it. In assessing the replacement model, we ask, assuming the selected mutation operator is *replace*, how often will an *IfStatement* be predicted as the replacer for a *ReturnStatement*? If it is returned in the first five instances in a model, the model correctly predicts this instance. For the equally distributed model, we select five operators/replacees uniformly at random.

We use 10-fold cross validation [19] to mitigate the risk of overfitting and avoid testing and training on the same data. We aggregate results over folds by summing the number of correctly predicted instances.

2) Results: Figure 5 shows results. The # columns show the number of attempted instances per fold. The "Prob." columns show the accuracy (correct prediction rate) of the mined model on that fold, for each model; "EqDist", the accuracy of the equally distributed model. For the Replacements model, the average accuracy of the Probabilistic model is 63% higher than the EqDist model; the difference is 56.4% for the Mutations model. Two sample t-tests indicate that both differences in means are statistically significant ($\alpha < 0.05$).

These results suggest that the probabilistic approach is significantly more accurate than an equally-distributed baseline (used in several syntactic APR approaches) in predicting bug-fixing edits. This suggests that the model, incorporated into an APR approach, may be more likely to produce correct patches. We directly address this question next.

B. Repair experiment setup

We have two high-level research concerns related to the utility of our learned models for repair: (1) Which syntactic operators and selection models are most useful for repair? (Section IV-C) and (2) How does our model-informed APR tool compare to the state-of-the-art in APR? (Section IV-D)

a) Validation of models in context: We first compare the utility of the various sets of syntactic mutation operators to identify which sets appears most useful in practice. We instantiate our tool with each of two models (the Equally Distributed baseline from Section IV-A and the learned Probabilistic models), and three different operator sets: (1) Statement-Edit mutations (2) Template-based mutations, and (3) All mutations. We evaluate each version for expressive power (in terms of bugs fixed), variants to repair (a machine- and test suite-independent proxy for time), and quality of the produced patches (evaluated as described below).

To diminish the degree of randomness in this experiment and control for the effect of the mutation probabilities specifically rather than inefficiency and noise in fault localization, we manually set the fault localization for each bug. Anecdotally, we observe that less-accurate fault localization increases the amount of time our tool needs to repair these defects, but does not decrease expressive power.

b) Comparison to previous techniques: Second, we compare our tool, using the best set of operators as identified in the first question, to four previously-proposed APR tools: GenProg [21], PAR [18], TrpAutoRepair [32], and Nopol [43]. These prior approaches provide coverage over several dimensions that differentiate program repair techniques. GenProg and TrpAutoRepair use the same statement-level edits, but differ in their search strategies (GenProg uses a genetic programming strategy; TrpAutoRepair, also known as RSRepair, a random walk). PAR uses GenProg's genetic programming search strategy, but a different set of templated mutation operators. Nopol is a semantics-guided generate-and-validate repair approach that thus uses a fix code identification strategy that is quite different from the operators we consider.[4]

The open-source implementation of GenProg for Java implements the first three approaches. We run these three techniques on the dataset using parameters described below. For Nopol, we use patch results released by the Nopol authors on this same dataset, and do not rerun their experiments. The Nopol authors have created patches for the same benchmark used in this study [24] and made their results publicly available.[5] We use results from the "March 2017" (most recent) release.

For these experiments, we focus on expressive power (bugs repaired) and patch quality rather than patch efficiency. Although important, it efficiency is not central to our claims, and, given our use of previously-released results, is difficult to evaluate in a controlled fashion in our context.

c) Dataset: We consider for repair a subset of the Defects4j [15] benchmark, a database and extensible framework

[4]We do not compare directly to SPR/Prophet [23], because they use machine learning to tune the probabilities and rank schema instantiation and are thus difficult to re-implement faithfully for Java; as established, their edit schemas are generalized by the extended PAR template set.

[5]https://github.com/Spirals-Team/defects4j-repair/tree/master/results/2017-march

Bug ID	Held-out Tests	Line Coverage	Statement Edits				Templates				Both			
			EqDist		Prob		EqDist		Prob		EqDist		Prob	
			Time	Gen?	Time	Gen?	Time	Gen?	Time	Gen?	Time	Gen?	Time	Gen?
Closure #10	472	60.2%	221.0	✓	179.5	✓	175.1	✓	121.3	✓	163.3	✓	157.4	✓
Closure #18	106	73.4%	–		–		36.2	✓	197.5	✓	45.0	✓	139.0	✓
Math #2	13	100.0%	–		–		109.4	✓	39.6	✓	109.4	✓	39.6	✓
Time #19	55	86.0%	94.1	✓	80.7	✓	–		–		135.1	✓	91.9	✓
Chart #1	93	74.4%	1.8	×	7.3	×	4.9	×	19.0	×	2.2	×	4.8	×

Fig. 6. Repair success of various models for three different operator sets. Columns 2 and 3 characterize the Evosuite-generated, held-out tests for each bug. For each bug, we report time in terms of variants evaluated to a repair (an average, if more than one patch is produced over the course of multiple random trials), and whether all produced patches generalize ("Gen?") to the held-out test suites (✓) or not (×). "–" indicates no patch was found in the given configuration.

of real bugs that enables reproducible studies in software testing and has been previously used to evaluate APR [24]. Table I characterizes the bugs from the dataset we consider. Because we compare to single-edit techniques, we restrict attention to a subset of the Defects4J bugs with single-line human patches. The number of bugs attempted per project varies based on how many such bugs are available in the dataset.

d) Patch quality: We evaluate patch quality using held-out test suites [21], [36]. We automatically generate a single held out suite for each buggy program version that any technique repairs. We use Evosuite [8], an established test suite generation tool for Java, to generate these held-out tests. We run EvoSuite with a 30-minute budget, using the human-repaired "after-fix" version of each Defects4J bug as the behavioral oracle. We use Cobertura[6] to calculate test suite coverage, again over the "after-fix" class that contains the human fix.

e) Settings: For the genetic programming based techniques (including ours), the population size is 40, and maximum generations is 10. For the other techniques, the maximum considered variant count is 400. We cap all runs at a timeout of 4 hours. These settings are consistent with prior work. We run 20 seeds per repair trial, in keeping with recommendations on the assessment of stochastic techniques [2].

C. Mutation operator and model utility

We evaluate our modified APR techniques, using each of three sets of operators and each of two models to select between them, on a subset of the bugs in Table I. Specifically, we consider the first 6 defects from each project. Figure 6 shows results. The first row (Closure #10) shows that the probabilistic model outperforms the equally-distributed model for all three edit sets. This happens as well for Math #2 and Time #19, except for the cases where no patch was found. By contrast, for bugs Chart #1 and Closure #18, the equally distributed approach finds a patch faster on average. Investigating this case manually, we find that these bugs are patched with mutations that are rarely applied by developers (Expression Replace, and Replacement accordingly), therefore it is more likely that the mutation operators will be chosen by a random selection than by the developer-informed model.

Our results suggest that considering all available mutation operators is preferable to restricting the mutation operator pool to just one of the categories. For the Statement-Edit category, 3 out of 6 were fixed faster when combining this kind of mutations with Template-Based mutations, and 3 out of 6 were slower. For the Template-Based category: 4 out of 8 were fixed faster when combining it with Statement-Edit mutations, 2 out of 8 were the same, and 2 out of 8 were slower. This suggests that the expressive power afforded by an increased set of operators may outweigh the commensurate increase in search space size, though it is safe to assume that this benefit must be supported by sufficiently accurate fault localization. In this sample, how quickly a model finds a patch appears independent of the mutation set.

In terms of quality assessment, the patches generated behave similarly within a given bug scenario. That is, for all bugs for which any approach found multiple patches, either all the patches generated by all configurations generalized to the held-out test suite, or none did.

From these results, we conclude that the probabilistic model using all available mutation operators (both Statement-Edit and Template-based) appears to maximize expressive power without an unacceptable loss to efficiency. In conjunction with our model assessment results (Section IV-A), we thus use all available mutations in our technique for the next experiment.

D. Comparison to other approaches

Next, we use our new APR approach to attempt to repair the defects described in Table I, comparing to other state-of-the-art approaches. Figure 7 shows results for the subset of bugs that our technique repairs (in the interest of space, we describe results for the remaining bugs in prose below). Figure 7 shows the number of unique patches found for each bug and how many of them generalized to a held-out test suite. The patches that did not generalize failed one or several tests in the held-out test suite. Of the 19 distinct patches created by our approach, 10 pass all held-out test suite (52.6%); 6.6% of GenProg's patches generalize; 22.2% of TRPAutoRepair's; 23.1% of PAR's; and 100% of the 5 Nopol patches generalize to the held-out test suite. Figure 8 shows our technique's patch for the Closure #18 bug; it is identical to the human patch.

A key takeaway from these results is that the probabilistic approach outperformed all the heuristic approaches. This is important because these approaches by construction select syntactic edits to perform according to some distribution. That is, these techniques are those that *could* benefit from a more informed edit distribution model. By contrast, Nopol produced 5 patches that all generalized to the held out test suite. However,

[6]http://cobertura.github.io/cobertura/

Bug ID	Held-out tests	Line Coverage	Prob. Model Found	Prob. Model Gen?	GenProg Found	GenProg Gen?	TrpAutoRepair Found	TrpAutoRepair Gen?	PAR Found	PAR Gen?	Nopol Found	Nopol Gen?
Chart # 1	93	74.4%	5	×	6	×	4	×	2	×	–	
Closure # 10	472	60.2%	2	✓	–		–		–		1	✓
Closure # 18	106	73.4%	1	✓	–		–		–		1	✓
Closure # 86	435	64.8%	2	✓	–		1	✓	–		–	
Lang # 33	85	92.3%	1	✓	–		–		1	✓	–	
Math # 2	13	100.0%	1	✓	–		–		1	✓	1	✓
Math # 75	25	92.5%	1	✓	–		–		1	✓	–	
Math # 85	11	94.7%	4	×	8	×	3	×	8	×	1	✓
Time # 19	55	86.0%	2	✓	1	✓	1	✓	–		1	✓

Fig. 7. Comparison between the probabilistic model-based repair and other state-of-the-art approaches. "–" indicates no patch found. The "Found" column indicate the number of patches found per bug over the multiple random trials. "Gen?" indicates whether all produced patches generalize to the held-out test suites (✓) or not (×). In these results, all produced patches for a bug, technique pair either generalized, or not.

```
-if(options.dependencyOptions.needsManagement()
-    && options.closurePass){
+if(options.dependencyOptions.needsManagement()){
   for (CompilerInput input : inputs) {
  // Forward-declare all the provided types, so that
  // they are not flagged even if they are dropped.
     for (String provide : input.getProvides()) {
       getTypeRegistry().forwardDeclareType(provide);
     }
   }
```

Fig. 8. A patch generated using the probabilistic model, identical to the developer patch; No other approach found this identical patch.

Nopol targets a specific bug type and does not use a heuristic approach. Our approach therefore presents an important benefit for a broader array of bug types, particularly those to which a semantics-based approach like Nopol doesn't apply. Overall, these results demonstrate the benefit of applying a probabilistic model for edit selection for those approaches to which such a selection process applies.

Our technique patched 9 of the 63 bugs in our evaluation. GenProg patched 9; Par, 16; Nopol, 27; and TRPAutoRepair, 8. There are 37 bugs for which at least one approach produced at least one patch. patch. From these, 19 were patched by only one approach; 10 were patched by two; 3 patched by three; 4 patched by four; and 1 patched by all five.

Many approaches can create several patches for each bug. Our approach created 19 distinct patches for the aforementioned 9 bugs. Of these, 10 (52.6%) pass the held-out test suites. Genprog created 46 distinct patches; 5 of them (10.9%) pass the test suites. TRPAutoRepair created 30 patches; 4 of them (13.3%) pass the test suites. PAR created 34 patches; 11 of them (32.4%) pass the test suites. Finally Nopol created 28 patches; 21 of them (75.0%) pass the test suites.

E. Association Rule Mining

In this section, we describe and evaluate the mutation operator association rules produced by mining human patches to identify edits that commonly occur together in human-generated patches (Section III-D). The goal of these models is to provide intuition regarding how to form multi-edit source code changes. Note that we create these association rules using strictly the Mutation Operator model corpus; the Replacements operator corpus is only informative when the "Replace" operator is chosen, and thus does not apply to the question of chaining together edits to produce larger patches.

1) Mutation operator rules: Below, we list the top 10 rules identified with 100% confidence in the dataset. This means that in 100% of the cases observed, every transaction that contained the antecedent of a rule also contained the consequent. A high threshold like 100% produces rules for APR that predict with high accuracy which edits to perform, given an initial set of edits. These rules are obtained with a 1% support, which means that each of these rules individually appear in at least 1% of all the transactions in the corpus. We show only the top association rules (the full set of rules is released with the code and data associated with this paper):

- Replace & Delete $\Longrightarrow$ Append
- Delete & AddNullCheck $\Longrightarrow$ Append
- Replace & SeqExchanger $\Longrightarrow$ Append
- Replace & ParamReplacer $\Longrightarrow$ Append
- Delete & CasteeMutator $\Longrightarrow$ Append
- Replace & Delete & ParamReplacer $\Longrightarrow$ Append
- Replace & AddNullCheck $\Longrightarrow$ Append
- Replace & Delete & SeqExchanger $\Longrightarrow$ Append
- Delete & ExpressionAdder $\Longrightarrow$ Append
- Delete & AddNullCheck & ParamReplacer $\Longrightarrow$ Append

The key observation to draw from these rules is that "Append" is the most common single edit mutation operator applied by developers. This behavior is reflected in the fact that it is the consequent in all the top mined rules. Overall, association rules provide an intuition of which common patterns of behavior developers use. These rules tell us which edits happen frequently together, supporting understanding of multi-edit source code changes, an understudied area that covers the majority of real patches.

2) Evaluation of Association Rules: To evaluate the effectiveness of the association rules in the context of the automatic program repair process, we first remove from the corpus human patches with fewer than three edits. This is because our mined rules all require at least two antecedents and one consequent. This removed 62.83% of the corpus. We validated the rules on the remaining 37.17% of patches as follows. First, we divide our corpus in 10 folds. For each fold, we build association rules

on the remaining nine, as described. Given the mined rules, we then we analyze how many testing patches (instances in the fold used as testing data) can be built by applying the learned rules. We categorize them as either *Fully covered*, *Partially covered*, or *Not covered*.

To illustrate via example, Table II shows three instances of patches in the testing data, and three rules. Instance 1 can be constructed by applying rules 1 and 3 and is thus classified as Fully covered. Rule 1 would cover the first three edits of Instance 2 instance, but there is no way to create the Replace ("Rep") mutation using the listed rules. This instance is classified as Partially covered. For Instance 3, even though Rule 3 contains two of the edits in the rule's antecedent, the instance does not contain the rule's consequent. The rules do not apply, and thus this instance is classified as Not covered.

TABLE II
PATCH INSTANCES (TOP); ASSOCIATED ASSOCIATION RULES (BOTTOM).

	Instances
1	Del; App; NullCheck; ObjInit
2	Del; App; NullCheck; Rep
3	App; NullCheck; CastMut
	Rules
1	Del $\wedge$ App $\rightarrow$ NullCheck
2	App $\wedge$ ParamRep $\rightarrow$ Rep
3	App $\wedge$ NullCheck $\rightarrow$ ObjInit

Finally, we performed this analysis at 6 different confidence thresholds (50%, 60%, 70%, 80%, 90%, 100%) to analyze the tradeoff between ruleset expressive power and size. A high confidence produces a small number of very accurate rules (when the antecedent is present, it is very likely that the consequent will be present as well). Setting the confidence lower produces the opposite trade-off: a large set of rules (covering more instances) where if the antecedent is present, it is less likely that the consequent will be present too. For each confidence threshold, we performed a standard 10-fold cross validation process with all the instances and all the rules for each fold and finally, we aggregate the results from all folds.

Figure 9 shows results. As expected, the number of rules created increases as confidence decreases. Note also that the number of Fully Covered instances increases as the confidence decreases, due to the fact that there are more rules, even though these rules are less accurate.

APR would benefit from having a small number of very accurate (high confidence) rules that would describe what edit to perform next after a series of edits, but at the same time, it needs rules that are flexible enough that they can generalize to a big portion of the patches. We find a good tradeoff at a confidence threshold of 90%. The 100% threshold provides very accurate rules, but can fully cover only 37.7% of the evaluation patches. By contrast, the 90% confidence threshold produces slightly more rules, but they are able to fully cover 84.6% of the patches.

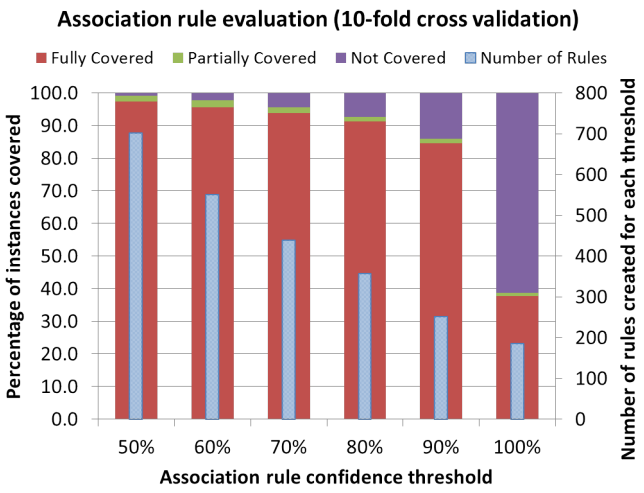

Fig. 9. The wide bars use the left vertical axis to describe the percentage of patches covered (Fully, Partially or Not) by the association rules. The thin bars use the right vertical axis to describe the number of rules created for each confidence threshold.

V. THREATS TO VALIDITY

Internal validity: Regarding possible errors in our implementation and experiments, to run our comparison with Genprog, we used an open-source implementation of GenProg targeting Java. We release our code, as well as our templates, independently-generated tests, and mined models for scrutiny and extension by other researchers, to mitigate the risk of errors in our implementation or approach. We also use 10-fold cross validation in assessing our model and association rules, to reduce the risk of training and testing on the same data.

External validity: It is possible that our results will not generalize to external datasets and to real bugs. To mitigate this concern, we build our model from well-known open-source programs, covering a diversity of applications, and distinct from the dataset from which the models were mined, and we evaluate our approach with bugs from an open-source framework.

There is also risk of producing low quality patches that would not generalize to a different description of desired program behavior. We attenuate this by assessing the quality of the generated patches with a held-out test suite. There is risk in the fact that we are manually giving the faulty location to the APR tool, since APR tools do not know in advance what the fault location is. This is for the purposes of evaluating the patch creating process only without the noise that fault localization might introduce.

Construct validity: Regarding the suitability of our evaluation metrics, we evaluate patch quality by running the generated patches on a second test suite created from a human patch, which is to a certain extent a biased measure since we can not guarantee that the human created patch is perfect [36]. Nevertheless, we consider this to be a best known practice, since this way we provide an alternative to subjectively asking a biased human developer whether he/she considered the patch to be correct or not.

We also rely on Evosuite [8] as our test suite generation mechanism for the held-out test suite used for evaluation, and we acknowledge that the test suites created by this tool may not be perfect, nor provide full coverage for all cases. Nonetheless, this is a state-of-the-art test suite generation tool that mitigates the risk of bias in manually constructing evaluation test suites.

VI. Related Work

There have been previous efforts to create an edit model based on human behavior. Soto et al. [37] built a probabilistic model to describe the replace operator only. They do not evaluate the model in the context of a repair tool, and the model is based on an instance count of statements rather than a more accurate analysis of AST differences, which our model was built from. HDRepair [5] uses fix history to assess patch suitability and fitness in the context of a genetic programming search strategy. The fitness of the generated fix candidates is determined by the frequency with which the changes included in a given patch occur in the corpus, using a graph-based representation of the bug fixes. Similarly, Prophet [23] uses a probabilistic model of a subset of our considered mutation operators built on the history of 8 different projects to rank candidate patches. Our approach follows this intuition to mimic human behavior; unlike the previous work, we apply this knowledge when actually creating patch candidates rather than when evaluating them, which reduces the search space at creation time.

Zhong and Su [44] perform an empirical study of real bug fixes on six projects, studying the incidence of three mutation operators, among other questions about the applicability of APR. Martinez and Monperrus [25] similarly study mutation operator incidence across 14 projects. Our work considers a broader set of mutation operators over a larger corpus, the largest, to the best of our knowledge, studied in existing work. To counter the risk of overfitting to a small set of training projects as performed before, our current study trains the model over 500 projects, covering a diversity of domains.

Par [18] describes a manual set of 10 templates of common behavior to create patches, showing that such templates result in patches of higher human-adjudged acceptability than statement-edit-based patches. Our study takes into consideration a superset of these templates, provides steps towards accounting for multi-edit source code changes, an understudied problem, and, importantly, mines and models these operators and their incidence automatically (rather than manually).

There exist a broad array of APR techniques proposed, especially recently; we survey many of them in Section II, focusing on heuristic or syntactic generate-and-validate techniques. Semantic-based techniques use semantic analysis or reasoning [20], [26], [27], [29], or semantic search [17] to construct candidate patches. Similarly, synthesis-based repair is a family of techniques that uses constraints to build patches following the description of the constraints [13], [39]. These constraints may be specifications created by developers, formal verification, invariants, etc. [13], [39]. Such techniques typically use synthesis to construct repairs, with a different mechanism for both constructing and traversing the search space, and our approach is thus less immediately comparable.

VII. Conclusion

In this paper, we analyze the way in which current state-of-the-art automatic program repair approaches select mutation operators to create candidate patches. We analyze, categorize and compare the mutation operators being used by state-of-the-art approaches. We analyzed the last 100 bug fixing commits from the 500 most-starred Java projects on GitHub, which is the largest corpus analyzed to date, to the best of our knowledge. We picked the most-stared projects because stars in Github are seen as proxies for popularity, which usually entail well-maintained, well-stablished, mature projects, that are more likely to have more and better quality bug fixes. We created a two-level probabilistic model describing the likelihood of selecting bug-fixing mutation operators, according to the observed incidence of their use by human developers. All the data gathered, tools and test suites used in this study are publicly available for open peer review and scrutiny.

We evaluated our approach in several ways: by performing an internal evaluation of its predictive power, and by running an APR tool using our model on 63 bugs from Defects4j. Finally we compare the quality of the patches generated against patches created for these same bugs by other state-of-the-art approaches. We measure both efficiency of patch production, and generalizability of the patches to a held-out test suite. Note that in Figure 6, for the majority of bugs, when using the probabilistic model we obtained better results than their equally distributed counterpart. Overall, we found that automatic program repair appears to benefit from having a diverse mutation operator pool; future improvements in fault localization thus will serve to further benefit from more precise models of mutation selection.

Multi-edit source code changes are understudied, but cover a large portion of bug fixes in software systems. To move towards advancing this area, we constructed a set of association rules that describe how subsets of mutation operators are applied together. Overall, our initial analysis of how to chain single-edit changes by following human behavior provides a first step towards an efficient mechanism for traversing the large search space of multi-edit repairs.

All code and data associated with these experiments are available for replication and extension: https://github.com/squaresLab/ProbabilisticModelSaner2018.

Acknowledgments

This work partially supported by the National Science Foundation (CCF-1563797) and the Air Force Research Laboratory (AFRL Contract No. FA8750-15-2-0075). Any findings or recommendations are those of the author(s) and do not necessarily reflect the views of the sponsoring agencies. The authors additionally wish to thank the anonymous reviewers, whose comments were particularly constructive.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases*, VLDB'94, pages 478–499, 1994.

[2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering*, ICSE'11, pages 1–10, 2011.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, ESEC/FSE'09, pages 121–130, 2009.

[4] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, 2013.

[5] X.-B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering*, SANER'16, 2016.

[6] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, ICST'10, pages 65–74, 2010.

[7] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, ASE'14, pages 313–324, 2014.

[8] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Foundations of software engineering*, ESEC/FSE '11, pages 416–419, 2011.

[9] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, ICSM'10, pages 1–10. Institute of Electrical and Electronics Engineers, 2010.

[10] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Automated Software Engineering*, pages 307–318, 2015.

[11] K. Herzig and A. Zeller. The impact of tangled code changes. In *Working Conference on Mining Software Repositories*, MSR'13, pages 121–130, 2013.

[12] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, ICSE'12, pages 837–847, 2012.

[13] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, PLDI'11, pages 389–400, 2011.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, ICSE'02, pages 467–477, 2002.

[15] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, ISSTA'14, pages 437–440, 2014.

[16] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *International Conference on Software Engineering*, ICSE'11, pages 351–360, 2011.

[17] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *International Conference On Automated Software Engineering*, ASE'15, pages 295–306, 2015.

[18] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, ICSE'13, pages 802–811, 2013.

[19] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *14th International Joint Conference on Artificial Intelligence*, IJCAI'95, pages 1137–1143, 1995.

[20] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Foundations of software engineering*, ESEC/FSE 2017, pages 593–604, 2017.

[21] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012.

[22] F. Long and M. Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference/International Symposium on the Foundations of Software Engineering*, FSE'15, pages 166–178, 2015.

[23] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, 2016.

[24] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Springer Empirical Software Engineering*, 21, 2016.

[25] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. In *Empirical Software Engineering*, ESE'15, pages 176–205, 2015.

[26] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering*, ICSE'15, pages 448–458, 2015.

[27] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, ICSE'16, pages 691–701, 2016.

[28] M. Monperrus. A critical review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the problem statement and the evaluation of automatic software repair. In *International Conference on Software Engineering*, ICSE '14, pages 234–242, 2014.

[29] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering*, ICSE'13, pages 772–781, 2013.

[30] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. The class-level mutants of mujava. In *2006 International Workshop on Automation of Software Test*, AST'06, pages 78–84, 2006.

[31] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, SOSP'09, pages 87–102, 2009.

[32] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance*, ICSM'13, pages 180–189, Sept. 2013.

[33] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, ISSTA'15, pages 24–36, 2015.

[34] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. 2010.

[35] A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk. . . In *International Conference on Software Engineering*, ICSE'06, pages 18–20, 2006.

[36] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *European Software Engineering Conference/International Symposium on the Foundations of Software Engineering*, ESEC/FSE'15, pages 532–543, 2015.

[37] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Mining Software Repositories*, MSR'16, pages 512–515, 2016.

[38] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, 2002.

[39] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, ISSTA'10, pages 61–72, 2010.

[40] W. Weimer, M. Dewey-Vogt, C. Le Goues, and S. Forrest. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, ICSE'12, pages 3–13, 2012.

[41] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 356–366, 2013.

[42] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *International Workshop on Mining Software Repositories*, MSR'07, page 1, 2007.

[43] J. Xuan, M. Martinez, F. Demarco, M. Clment, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, pages 34–55, 2016.

[44] H. Zhong and Z. Su. An empirical study on real bug fixes. In *International Conference on Software Engineering*, ICSE'15, pages 913–923, 2015.