Learning How to Mutate Source Code from Bug-Fixes

Michele Tufano*, Cody Watson†, Gabriele Bavota‡, Massimiliano Di Penta‡, Martin White*, Denys Poshyvanyk*

*College of William and Mary, Williamsburg, Virginia, USA
Email: {mtufano, cawatson, mgwhite, denys}@cs.wm.edu
†Università della Svizzera italiana (USI), Lugano, Switzerland
Email: {gabriele.bavota}@usi.ch
‡University of Sannio, Benevento, Italy
Email: dipenta@usisannio.it

Abstract—Mutation testing has been widely accepted as an approach to guide test case generation or to assess the effectiveness of test suites. Empirical studies have shown that mutants are representative of real faults; yet they also indicated a clear need for better, possibly customized, mutation operators and strategies. While some recent papers have tried to devise domain-specific or general purpose mutator operators by manually analyzing real faults, such an activity is effort- (and error-) prone and does not deal with an important practical question as to how to really mutate a given source code element. We propose a novel approach to automatically learn mutants from faults in real programs. First, our approach processes bug fixing changes using fine-grained differencing, code abstraction, and change clustering. Then, it learns mutation models using a deep learning strategy. We have trained and evaluated our technique on a set of 787k bugs mined from GitHub. Starting from code fixed by developers in the context of a bug-fix, our empirical evaluation showed that our models are able to predict mutants that resemble original fixed bugs in between 9% and 45% of the cases (depending on the model). Moreover, over 98% of the automatically generated mutants are lexically and syntactically correct.

Index Terms—mutation testing, deep learning, neural networks

I. INTRODUCTION

Mutation testing is a program analysis technique aimed at injecting artificial faults into the program’s source code or bytecode [1], [2] to simulate defects. Mutants (i.e., versions of the program with an artificial defect) can guide the design of a test suite, i.e., test cases are written or automatically generated by tools such as Evosuite [3], [4] until a given percentage of mutants has been “killed”. Also, mutation testing can be used to assess the effectiveness of an existing test suite when the latter is already available [5], [6].

A number of studies have been dedicated to understand the interconnection between mutants and real faults [7]–[14]. Daran and Thévenod-Fosse [9] and Andrews et al. [7], [8] indicated that mutants, if carefully selected, can provide a good indication of a test suite’s ability to detect real faults. However, they can underestimate a test suite’s fault detection capability [7], [8]. Also, as pointed out by Just et al. [13], there is a need to improve mutant taxonomies in order to make them more representative of real faults. In summary, previous work suggests that mutants can be representative of real faults if they properly reflect the types and distributions of faults that programs exhibit. Taxonomies of mutants have been devised by taking typical bugs into account [15], [16]. Furthermore, some authors have tried to devise specific mutants for certain domains [17]–[21]. A recent work by Brown et al. [22] leveraged bug-fixes to extract syntactic-mutation patterns from the diffs of patches. The authors mined 7.5k types of mutation operators that can be applied to generate mutants.

However, devising specific mutant taxonomies not only requires a substantial manual effort, but also fails to sufficiently cope with the limitations of mutation testing pointed out by previous work. For example, different projects or modules may exhibit diverse distributions of bugs and types, helping to explain why defect prediction approaches do not work out-of-the-box when applied cross-project [23].

Instead of manually devising mutant taxonomies, we propose to automatically learn mutants from existing bug fixes. Such a strategy is likely to be effective for several reasons. First, a massive number of bug-fixing commits are available in public repositories. In our exploration, we found around 10M bug-fix related commits on GitHub just in the last six years. Second, a buggy code fragment arguably represents the perfect mutant for the fixed code because: (i) the buggy version is a mutation of the fixed code; (ii) such a mutation already exposed a buggy behavior; (iii) the buggy code does not represent a trivial mutant; (iv) the test suite did not detect the bug in the buggy version. Third, advanced machine learning techniques such as deep learning have been successfully applied to capture code characteristics and effectively support several SE tasks tasks [24]–[33].

Stemming from such considerations, and being inspired from the work of Brown et al. [22], we propose an approach for automatically learning mutants from actual bug fixes. After having mined bug-fixing commits from software repositories, we extract change operations using an AST-based differencing tool and abstract them. Then, to enable learning of specific mutants, we cluster similar changes together. Finally, we learn from the changes using a Recurrent Neural Network (RNN) Encoder-Decoder architecture [34]–[36]. When applied to unseen code, the learned model decides in which location and what changes should be performed. Besides being able to learn mutants from an existing source code corpus, and
differently from Brown et al. [22], our approach is also able to determine where and how to mutate source code, as well as to introduce new literals and identifiers in the mutated code.

We evaluate our approach on 787k bug-fixing commits with the aim of investigating (i) how similar the learned mutants are as compared to real bugs; (ii) how specialized models (obtained by clustering changes) can be used to generate specific sets of mutants; and (iii) from a qualitative point of view, what operators were the models able to learn. The results indicate that our approach is able to generate mutants that perfectly correspond to the original buggy code in 9% to 45% of cases (depending on the model). Most of the generated mutants are syntactically correct (more than 98%), and the specialized models are able to inject different types of mutants.

This paper provides the following contributions:

- A novel approach for learning how to mutate source code from bug-fixes. To the best of our knowledge, this is the first attempt to automatically learn and generate mutants.
- Empirical evidence that our models are able to learn diverse mutation operators that are closely related to real bugs.
- We release the data and code to enable replication [37].

II. APPROACH

We start by mining bug-fixing commits from thousands of GitHub repositories (Sec. II-A). From the bug-fixes, we extract method-level pairs of buggy and corresponding fixed code that we call transformation pairs (TPs) (Sec. II-B1). TPs represent the examples we use to learn how to mutate code from bug-fixes (fixed → buggy). We rely on GumTree [38] to extract a list of edit actions (A) performed between the buggy and fixed code. Then, we use a Java Lexer and Parser to abstract the source code of the TPs (Sec. II-B2). Finally, for each set of TPs we use an encoder-decoder model to learn how to transform a fixed piece of code into the corresponding buggy version (Sec. II-C).

A. Bug-Fixes Mining

We downloaded the GitHub Archive [39] containing every public GitHub event between March 2011 and October 2017. Then, we used the Google BigQuery APIs to identify commits related to bug-fixes. We selected all the commits having a message containing the patterns: (“fix” or “solve”) and (“bug” or “issue” or “problem” or “error”).

We identified 10,056,052 bug-fixing commits for which we extracted the commit ID (SHA), the project repository, and the commit message.

Since we are aware that not all commit messages matching our pattern are necessarily related to corrective maintenance [40], [41], we assessed the precision of the regular expression used to identify bug-fixing commits. Two authors independently analyzed a statistically significant sample (95% confidence interval, for a total size of 384) of identified commits to judge whether the commits were actually referring to bug-fixing activities. Next, the authors met to resolve a few disagreements in the evaluation (only 13 cases). The evaluation results, available in our appendix [37], reported a true positive rate of 97%. The commits classified as false positives mainly referred to partial and incomplete fixes.

After collecting the bug-fixing commits, for each commit we extracted the source code pre- and post- bug-fixing (i.e., buggy and fixed code) by using the GitHub Compare API [42].

During this process, we discarded files that were created in the bug-fixing commit, since there is no buggy version to learn from, as the mutant would be the deletion of the entire source code file. In this phase, we also discarded commits that had touched more than five Java files, since we aim to learn from bug-fixes focusing on only a few files and not spread across the system, and as found in previous work [43], large commits are more likely to represent tangled changes, i.e., commits dealing with different tasks. Also, we excluded commits related to repositories written in languages different than Java, since we aim at learning mutation operators for Java code. After these filtering steps, we extracted the pre- and post-code from ~787k (787,178) bug-fixing commits.

B. Transformation Pairs Analysis

A TP is a pair (m_b, m_f) where m_b represents a buggy code component and m_f represents the corresponding fixed code. We will use these TPs as examples when training our RNN. The idea is to train the model to learn the transformation from the fixed code component (m_f) to the buggy code (m_b), in order to generate mutants that are similar to real bugs.

1) Extraction: Given a bug-fix bf, we extracted the buggy files (f_b) and the corresponding fixed (f_f) files. For each pair (f_b, f_f), we ran AST differencing between the ASTs of f_b and f_f using GumTree Spoon AST Diff [38], to compute the sequence of AST edit actions that transforms f_b into f_f.

Instead of computing the AST differencing between the entire buggy and fixed files, we separate the code into method-level pieces that will constitute our TPs. We first rely on GumTree to establish the mapping between the nodes of f_b and f_f. Then, we extract the list of mapped pairs of methods L = {(m_b1, m_f1), . . . , (m_bn, m_fn)}. Each pair (m_b, m_f) contains the method m_b (from the buggy file f_b) and the corresponding mapped method m_f (from the fixed file f_f). Next, for each pair of mapped methods, we extract a sequence of edit actions using the GumTree algorithm. We then consider only those method pairs for which there is at least one edit action (i.e., we disregard methods unmodified during the fix).

Therefore, the output of this phase is a list of TPs = {tp1, . . . , tp_k}, where each TP is a triplet tp = {m_b, m_f, A}, where m_b is the buggy method, m_f is the corresponding fixed method, and A is a sequence of edit actions that transforms m_b in m_f. We do not consider any methods that have been newly created or completely deleted within the fixed file, since we cannot learn mutation operations from them. Also, TPs do not capture changes performed outside methods (e.g., class name).
The rationale behind the choice of method-level TPs is manyfold. First, methods represent a reasonable target for mutation, since they are more likely to implement a single task. Second, methods provide enough meaningful context for learning mutations, such as variables, parameters, and method calls used in the method. Smaller snippets of code lack the necessary context. Third, file- or class-level granularity could be too large to learn patterns of transformation. Finally, considering arbitrarily long snippets of code, such as hunks in diffs, could make the learning more difficult given the variability in size and context [44, 45]. Note that we consider each TP as an independent fix, meaning that multiple methods modified in the same bug fixing activity are considered independently from one other. In total, we extracted ∼2.3M TPs.

2) Abstraction: The major problem in dealing with raw source code in TPs is the extremely large vocabulary created by the multitude of identifiers and literals used in the code of the ∼2M mined projects. This large vocabulary would hinder our goal of learning transformations of code as a neural machine translation task. Therefore, we abstract the code and generate an expressive yet vocabulary-limited representation. We use a combination of a Java lexer and parser to represent each buggy and fixed method within a TP, as a stream of tokens. First, the lexer (based on ANTLR [46, 47]) reads the raw code tokenizing it into a stream of tokens. The tokenized stream is then fed into a Java parser [48], which discerns the role of each identifier (i.e., whether it represents a variable, method, or type name) and the type of literals.

Each TP is abstracted in isolation. Given a TP \( tp = \{m_b, m_f, A\} \), we first consider the source code of \( m_f \). The source code is fed to a Java lexer, producing the stream of tokens. The stream of tokens is then fed to a Java parser, which recognizes the identifiers and literals in the stream. The parser then generates and substitutes a unique ID for each identifier/literal within the tokenized stream. If an identifier or literal appears multiple times in the stream, it will be replaced with the same ID. The mapping of identifiers/literals with their corresponding IDs is saved in a map (\( M \)). The final output of the Java parser is the abstracted method (\( \text{abstract}_f \)). Then, we consider the source code of \( m_b \). The Java lexer produces a stream of tokens, which is then fed to the parser. The parser continues to use map \( M \) for \( m_b \). The parser generates new IDs only for novel identifiers/literals, not already contained in \( M \), meaning, they exist in \( m_b \), but not in \( m_f \). Then, it replaces all the identifiers/literals with the corresponding IDs, generating the abstracted method (\( \text{abstract}_b \)). The abstracted TP is now the following 4-tuple \( \text{tpa} = \{\text{abstract}_b, \text{abstract}_f, A, M\} \), where \( M \) is the ID mapping for that particular TP. The process continues considering the next TP, generating a new mapping \( M \). Note that we first analyze the fixed code \( m_f \) and then the corresponding buggy code \( m_b \) of a TP since this is the direction of the learning process (from \( m_b \) to \( m_f \)).

The assignment of IDs to identifiers and literals occurs in a sequential and positional fashion. Thus, the first method name found will receive the ID \( \text{METHOD}_1 \), likewise the second method name will receive ID \( \text{METHOD}_2 \). This process continues for all method and variable names (\( \text{VAR}_X \) and literals (\( \text{STRING}_X, \text{INT}_X, \text{FLOAT}_X \)). Figure 1 shows an example of the TP’s abstracted code. It is worth noting that IDs are shared between the two versions of the methods and new IDs are generated only for newly found identifiers/literals. The abstracted code allows to substantially reduce the number of unique words in the vocabulary because we allow the reuse of IDs across different TPs. For example, the first method name identifier in any transformation pair will be replaced with the ID \( \text{METHOD}_1 \), regardless of the original method name.

Fig. 1: Transformation Pair Example.

At this point, \( \text{abstract}_b \) and \( \text{abstract}_f \) of a TP are a stream of tokens consisting of language keywords (e.g., if), separators (e.g., “(”, “;”) and IDs representing identifiers and literals. Comments and annotations have been removed.

Figure 1 shows an example of a TP. The left side is the buggy code and the right side is the same method after the bug-fix (changed the if condition). The abstracted stream of tokens is shown below each corresponding version of the code. Note that the fixed version is abstracted before the buggy version. The two abstracted streams share most of the IDs, except for the INT_2 ID (corresponding to the int value 0), which appears only in the buggy version.

There are some identifiers and literals that appear so often in the source code that, for the purpose of our abstraction, they can almost be treated as keywords of the language. For example, the variables i, j, or index are often used in loops. Similarly, literals such as 0, 1, −1 are often used in conditional statements and return values. Method names, such as getValue, appear multiple times in the code as they represent common concepts. These identifiers and literals are often referred to as “idioms” [22]. We keep these idioms in our representation, that is, we do not replace idioms with a generated ID, but rather keep the original text in the code representation. To define the list of idioms, we first randomly sampled 300k TPs and considered all their original source code. Then, we extracted the frequency of each identifier/literal used in the code, discarding keywords, separators, and comments. Next, we analyzed the distribution of the frequencies and focused on the top frequent words (outliers of the distribution). In particular, we focused on the top 0.005% of the distribution. Two authors manually analyzed this list and curated a set of 272 idioms. Idioms also include standard Java types such as String, Integer, common Exceptions, etc. The complete list of idioms is available on our online appendix [37].

Figure 1 shows the idiomized abstracted code at the bottom. The method name size is now kept in the representation.
and not substituted with an ID. This is also the case for the literal values 0 and 1, which are very frequent idioms. Note that the method name `populateList` is now assigned ID `METHOD_2` rather than `METHOD_3`. This representation provides enough context and information to effectively learn code transformations, while keeping a limited vocabulary ($|V| = \sim 430$). Note that the abstracted code can be mapped back to the real source code using the the mapping ($M$).

3) Filtering Invalid TPs: Given the extracted list of 2.3M TPs, we manipulated their code via the aforementioned abstraction method. During the abstraction, we filter out such TPs that: (i) contain lexical or syntactic errors (i.e., either the lexer or parser failed to process them) in either the buggy or fixed version of the code; (ii) their buggy and fixed abstracted code ($abstract_b, abstract_f$) resulted in equal strings. The equality of $abstract_b$ and $abstract_f$ is evaluated while ignoring whitespace, comments or annotations edits, which are not useful in learning mutants. Next, we filter out TPs that performed more than 100 atomic AST actions ($|A| > 100$) between the buggy and fixed version. The rationale behind this decision was to eliminate outliers of the distribution (the 3rd quartile of the distribution is 14 actions) which could hinder the learning process. Moreover, we do not aim to learn such large mutations. Finally, we discard long methods and focus on small/medium size TPs. We filter out TPs whose fixed or buggy abstracted code is longer than 50 tokens. We discuss this choice in the Section \[\Box\] and report preliminary results also for longer methods. After the filtering, we obtained $\sim 380k$ TPs.

4) Synthesis of Identifiers and Literals: TPs are the examples we use to make our model learn how to mutate source code. Given a $tp = \{m_b, m_f, A\}$, we first abstract its code, obtaining $tpa = \{abstract_b, abstract_f, A, M\}$. The fixed code $abstract_f$ is used as input to the model which is trained to output the corresponding buggy code (mutant) $abstract_b$. This output can be mapped back to real source code using $M$.

In the current usage scenario (i.e., generating mutants), when the model is deployed, we do not have access to the oracle (i.e., buggy code, $abstract_b$), but only to the input code. This source code can then be abstracted and fed to the model, which generates as output a predicted code ($abstract_p$). The IDs that the $abstract_p$ contains can be mapped back to real values only if they also appear in the input code. If the mutated code suggests to introduce a method call, `METHOD_6`, which is not found in the input code, we cannot automatically map `METHOD_6` to an actual method name. This inability to map back source code exists for any newly created ID generated for identifiers or literals, which are absent in the input code. Synthesizing new identifiers would involve extensive knowledge about the project, control and data flow information. For this reason, we discard the TPs that contain, in the buggy method $m_b$, new identifiers not seen in the fixed method $m_f$. The rationale is that we want to train our model from examples that rearrange existing identifiers, keywords and operators to mutate source code. Instead, this is not the case for literals. While we cannot perfectly map a new literal ID to a concrete value, we can still synthesize new literals by leveraging the type information embedded in the ID. For example, the (fixed) if condition in Figure 1 if (VAR_1.METHOD_2 () < INT_1) should be mutated in its buggy version if (VAR_1.METHOD_2 () < INT_2). The value of INT_2 has never appeared in the input code (fixed), but we could still generate a compilable mutant by randomly generating a new integer value (different from any literal in the input code). While in these cases the literal value is randomly generated, the mutation model still provides the prediction about which literal to mutate.

For such reasons, we create two sets of TPs, hereby referred as $TP_{ident}$ and $TP_{ident-\text{lit}}$. $TP_{ident}$ contains all TPs $tpa = \{abstract_b, abstract_f, A, M\}$ such that every identifier ID (VAR_, METHOD_, TYPE_) in $abstract_b$ is found also in $abstract_f$. In this set we do allow new literal IDs (STRING_, INT_, etc.). $TP_{ident-\text{lit}}$ is a subset of $TP_{ident}$, which is more restrictive, and only contains the transformation pairs $tpa = \{abstract_b, abstract_f, A, M\}$ such that every identifier and literal ID in $abstract_b$ is found also in $abstract_f$. Therefore, we do not allow new identifiers nor literals.

The rationale behind this choice is that we want to learn from examples (TPs) where the model is able to generate compilable mutants (i.e., generate actual source code, with real values for IDs). In the case of the $TP_{ident-\text{lit}}$ set, the model will learn from examples that do not introduce any new identifier and literal. This means that the model will likely generate code for which every literal and identifier can be mapped to actual values. From the set $TP_{ident}$ the model will likely generate code for which we can map every identifier but we may need to generate new random literals.

In this context it is important to understand the role played by the idioms in our code representation. Idioms help to retain transformation pairs that we would otherwise discard, and learn transformation of literal values that we would otherwise need to randomly generate. Consider again the previous example if (VAR_1.METHOD_2 () < INT_1) and its mutated version if (VAR_1.METHOD_2 () < INT_2). In this example, there are no idioms and, therefore, the model learns to mutate INT_1 to INT_2 within the if condition. However, when we want to map back the mutated (buggy) representation to actual source code, we will not have a value for INT_2 (which does not appear in the input code) and, thus, we will be forced to generate a synthetic value for it. Instead, with the idiomized abstract representation the model would treat the idioms 0 and 1 as keywords of the language and learn the exact transformation of the if condition. The proposed mutant will therefore contain directly the idiom value (1) rather than INT_2. Thus, the model will learn and propose such transformation without the need to randomly generate literal values. In summary, idioms increase the number of transformations incorporating real values rather than abstract representations. Without idioms, we would lose these transformations and our model could be less expressive.

5) Clustering: The goal of clustering is to create subsets of TPs such that TPs in each cluster share a similar list of AST actions. Each cluster represents a cohesive set of examples so
that a trained model can apply those actions to a new code. As previously explained, each transformation pair \( tp = \{m_i, f_j, A\} \) includes a list of AST actions \( A \). In our dataset, we found \( \sim 1,200 \) unique AST actions, and each TP can perform a different number and combination of these actions. Deciding whether the transformation pairs, \( t_{p1} \) and \( t_{p2} \), perform a similar sequence of actions and, thus, should be clustered together, is far from trivial. Possible similarity functions include the number of shared elements in the two sets of actions and the frequency of particular actions within the sets. Rather than defining such handcrafted rules, we choose to learn similarities directly from the data. We rely on an unsupervised learning algorithm that learns vector representations for the lists of actions \( A \) of each TP. We treat each list of AST actions \( A \) as a document and rely on \textit{doc2vec} \cite{le2014distributed} to learn a fixed-size vector representation of such variable-length documents embedded in a latent space where similarities can be computed as distances. The closer two vectors are, the more similar the content of the two corresponding documents. In other words, we mapped the problem of clustering TPs to the problem of clustering continuous valued vectors. To this goal, we use \( k \)-means clustering, requiring the number of clusters \( k \) into which to partition the data upfront. When choosing \( k \), we need to balance two conflicting factors: (i) maximize the number of clusters so that we can train several different mutation models and, as a consequence, apply different mutations to a given piece of code; and (ii) have enough training examples (TPs) in each cluster to make the learning possible. Regarding the first point, we target at least three mutation models. Concerning the second point, with the available TPs dataset we could reasonably train no more than six clusters, so that each of those contain enough TPs. Thus, we experiment on the dataset \( TP_{ident-lit} \) with values of \( k \) going from 3 to 6 at steps of 1 and we evaluate each clustering solution in terms of its Silhouette statistic \cite{rousseeuw1987silhouettes,rousseeuw1987silhouettes}, a metric used to judge the quality of clustering. We found that \( k = 5 \) generates the clusters with the best overall Silhouette values. We cluster the dataset \( TP_{ident-lit} \) into clusters: \( C_1, C_2, C_3, C_4, C_5 \).

C. Learning Mutations

1) Dataset Preparation: Given a set of TPs (i.e., \( TP_{ident}, TP_{ident-lit}, C_1, \ldots, C_5 \)) we use the instances to train our Encoder-Decoder model. Given a \( tpa = \{abstract_b, abstract_f, A, M\} \) we use only the pair \( \{abstract_f, abstract_b\} \) of fixed and buggy abstracted code for learning. No additional information about the possible mutation actions \( A \) is provided during the learning process to the model. The given set of TPs is randomly partitioned into: training (80%), evaluation (10%), and test (10%) sets. Before the partitioning, we make sure to remove any duplicated pairs \( \{abstract_f, abstract_b\} \) to not bias the results (i.e., same pair both in training and test set).

2) Encoder-Decoder Model: Our models are based on an RNN Encoder-Decoder architecture, commonly adopted in Machine Translation \cite{bahdanau2014neural,cho2014learning}. This model comprises two major components: an RNN Encoder, which \textit{encodes} a sequence of terms \( x \) into a vector representation, and an RNN Decoder, which \textit{decodes} the representation into another sequence of terms \( y \). The model learns a conditional distribution over a (output) sequence conditioned on another (input) sequence of terms: \( P(y_1, \ldots, y_m| x_1, \ldots, x_n) \), where \( n \) and \( m \) may differ. In our case, given an input sequence \( x = abstract_f = (x_1, \ldots, x_n) \) and a target sequence \( y = abstract_b = (y_1, \ldots, y_m) \), the model is trained to learn the conditional distribution: \( P(abstract_b|abstract_f) = P(y_1, \ldots, y_m|x_1, \ldots, x_n) \), where \( x_i \) and \( y_j \) are abstracted source tokens: Java keywords, separators, IDs, and idioms. The Encoder takes as input a sequence \( x = (x_1, \ldots, x_n) \) and produces a sequence of states \( h = (h_1, \ldots, h_n) \). We rely on a bi-directional RNN Encoder \cite{schuster1997bidirectional} which is formed by a backward and forward RNNs, which are able to create representations taking into account both past and future inputs \cite{maas2011learning}. That is, each state \( h_i \) represents the concatenation of the states produced by the two RNNs reading the sequence in a forward and backward fashion: \( h_i = \left[ h_i^f; h_i^b \right] \). The RNN Decoder predicts the probability of a target sequence \( y = (y_1, \ldots, y_m) \) given \( h \). Specifically, the probability of each output term \( y_j \) is computed based on: (i) the recurrent state \( s_i \) in the Decoder; (ii) the previous \( i-1 \) terms \( (y_1, \ldots, y_{i-1}) \); and (iii) a context vector \( c_i \). The latter practically constitutes the attention mechanism. The vector \( c_i \) is computed as a weighted average of the states in \( h \), as follows: \( c_i = \sum_{t=1}^{n} a_{it} h_t \) where the weights \( a_{it} \) allow the model to pay more attention to different parts of the input sequence. Specifically, the weight \( a_{it} \) defines how much the term \( x_i \) should be taken into account when predicting the target term \( y_j \). The entire model is trained end-to-end (Encoder and Decoder jointly) by minimizing the negative log likelihood of the target terms, using stochastic gradient descent.

3) Configuration and Tuning: For the RNN Cells we tested both LSTM \cite{hochreiter1997long} and GRU \cite{chung2014empirical}, founding the latter to be slightly more accurate and faster to train. Before settling on the bi-directional Encoder, we tested the unidirectional Encoder (with and without reversing the input sequence), but we consistently found the bi-directional one yielding more accurate results. Bucketing and padding was used to deal with the variable length of the sequences. We tested several combinations of the number of layers (1,2,3,4) and units (256, 512). The configuration that best balanced performance and training time was the one with 1 layer encoder, 2 layer decoder both with 256 units. We train our models for 40k epochs, which represented our empirically-based sweet spot between training time and loss function improvements. The evaluation step was performed every 1k epochs.

III. EXPERIMENTAL DESIGN

The evaluation has been performed on the dataset of bug fixes described in Section II and answers three RQs.

RQ1: Can we learn how to generate mutants from bug-fixes? RQ1 investigates the extent to which bug fixes described in Section II and answers three RQs.
The first performance metric we use is the Bilingual Evaluation Understudy (BLEU) score, a metric used to assess the quality of a machine translated sentence [55]. BLEU scores require reference text to generate a score, which indicates how similar the candidate and reference texts are.

The candidate and reference texts are broken into n-grams and the algorithm determines how many n-grams of the candidate text appear in the reference text. We report the global BLEU score, which is the geometric mean of all n-grams up to four. To assess our mutant generation approach, we first compute the BLEU score between the abstracted fixed code ($abstract_f$) and the corresponding target buggy code. This BLEU score serves as our baseline for comparison. We compute the BLEU score between the predicted mutant ($abstract_p$) and the target ($abstract_b$). The higher the BLEU score, the more similar $abstract_p$ is to $abstract_b$, i.e., the actual buggy code. To fully understand how similar our prediction is to the real buggy code, we need to compare the BLEU score with our baseline. Indeed, the input code (i.e., the fixed code) provided to our model can be considered by itself as a “close translation” of the buggy code, therefore, helping in achieving a high BLEU score. To avoid this bias, we compare the BLEU score between the fixed code and the buggy code (baseline) with the BLEU score obtained when comparing the predicted buggy code ($abstract_p$) to the actual buggy code ($abstract_b$). If the BLEU score between $abstract_p$ and $abstract_b$ is higher than that one between $abstract_f$ and $abstract_b$, it means that the model transforms the input code ($abstract_f$) into a mutant ($abstract_p$) that is closer to the buggy code ($abstract_b$) than it was before the mutation, i.e., the mutation goes in the right direction. In the opposite case, the predicted code represents a translation that is further from the buggy code when compared to the original input. To assess whether the differences in BLEU scores between the baseline and the models are statistically significant, we employ a technique devised by Zhang et al. [59]. Given the test set, we generate $m = 2,000$ test sets by sampling with replacement from the original test set. Then, we evaluate the BLEU score on the $m$ test sets both for our model and the baseline. Next, we compute the $m$ deltas of the scores: $\delta_i = model_i - baseline_i$. Given the distribution of the deltas, we select the 95% confidence interval (CI) (i.e., from the 2.5th percentile to the 97.5th percentile). If the CI is completely above or below the zero (e.g., 2.5th percentile > 0) then the differences are statistically significant.

**Prediction Classification.**

Given $abstract_f$, $abstract_b$ and $abstract_p$, we classify each prediction into one of the following categories: (i) perfect prediction if $abstract_p = abstract_b$ (the model converts the fixed code back to its buggy version, thus reintroducing the original bug); (ii) bad prediction if $abstract_p = abstract_f$ (the model was not able to mutate the code and returned the same input code); and (iii) mutated prediction if $abstract_p \neq abstract_b$ AND $abstract_p \neq abstract_f$ (the model mutated the code, but differently than the target buggy code). We report raw numbers and percentages of the predictions falling in the described categories.

**Syntactic Correctness.** To be effective, mutants need to be syntactically correct, allowing the project to be compiled and tested against the test suite. We evaluate whether the models’ predictions are lexically and syntactically correct by means of a Java lexer and parser. Perfect predictions and bad predictions are already known to be syntactically correct, since we established the correctness of the buggy and fixed code when extracting the TPs. The correctness of the predictions within the mutated prediction category is instead unknown. For this reason, we report both the overall percentage of syntactically correct predictions as well as the mutated predictions. We do not assess the compilability of the code.

**Token-based Operations.** We analyzed and classified models’ predictions also based on their tokens’ operations, classifying the predictions into one of four categories: (i) insertion if #tokens predictions > #tokens input; (ii) changes if #tokens prediction = #tokens input AND prediction ≠ input; (iii) deletions if #tokens prediction < #tokens input; (iv) none if prediction = input. This analysis aims to understand whether the models are able to insert, change or delete tokens.

**AST-based Operations.** Next, we focus on the mutated predictions. These are not perfect predictions, but we are interested in understanding whether the transformations performed by the models are somewhat similar to the transformations between the fixed and buggy code. In other words, we investigate whether the model performs AST actions similar to the ones needed to transform the input (fixed) code into the corresponding buggy code. Given the input fixed code $abstract_f$, the corresponding buggy code $abstract_b$, and the predicted mutant $abstract_p$, we extract with GumTreeDiff the following lists of AST actions: $A_{f-b} = actions(abstract_f \rightarrow abstract_b)$ and $A_{f-p} = actions(abstract_f \rightarrow abstract_p)$. We then compare the two lists of actions, $A_{f-b}$ and $A_{f-p}$, to assess their similarities. We report the percentage of mutated predictions whose list of actions $A_{f-p}$ contains the same elements and frequency of those found in $A_{f-b}$. We also report the percentage of mutated predictions when only comparing their unique actions and disregarding their frequency. In those cases, the model performed the same list of actions but possibly in a different order, location or frequency than those which led to the perfect prediction (buggy code).

**RQ2: Can we train different mutation models?** RQ2 evaluates the five models trained using the five clusters of TPs. For each model, we evaluate its performance on the corresponding 10% test set using the same analyses discussed for RQ1. In addition, we evaluate whether models belonging to different clusters generate different mutants. To this aim, we first concatenate the test set of each cluster into a single test set. Then, we feed each input instance in the test set (fixed code) to each and every mutation model $M_1, .. , M_5$, obtaining
five mutant outputs. After that, we compute the number of unique mutants generated by the models. For each input, the number of unique mutants ranges from one to five depending on how many models generate the same mutation. We report the distribution of unique mutants generated by the models.

RQ3: What are the characteristics of the mutants generated by the models? RQ3 qualitatively assesses the generated mutants through manual analysis. We first discuss some of the perfect predictions found by the models. Then, we focus our attention on the mutated predictions (neither perfect nor bad predictions). We randomly selected a statistically significant sample from the mutated predictions of each cluster-model and manually analyzed them. The manual evaluation assesses (i) whether the functional behavior of the generated mutant differs from the original input code; (ii) the types of mutation operations performed by the model in generating the mutant.

Three judges, among the authors, were involved in this analysis, and we required each instance to be independently evaluated by two judges. The judges were presented with the original input code and the mutated code. The judges defined the mutation operations types in an open-coding fashion. Also, they were required to indicate whether the performed mutation changed the code behavior. After the initial evaluation, two of the three judges met to discuss and resolve the conflicts in the evaluation results. We define a conflict as any instance for which two of the judges disagreed on either the changes in the functional behavior or the set of mutation operations assigned. We report the distribution of the mutation operators applied by the different cluster-models and highlight the differences.

IV. RESULTS

BLEU Score. The top part of Table I shows the BLEU scores obtained by the two general models and compared with the baseline. The rightmost column represents the 2.5th percentile of the distribution of the deltas. Compared to the baseline, the models achieve a better BLEU score when mutating the source code w.r.t. the target buggy code. The differences are statistically significant, and the 2.5th percentile of the distribution of the deltas (+5.63 and +7.97), shows that the models’ BLEU scores are significantly higher than those obtained by the baseline. The observed increase in BLEU score indicates that the code mutated by our approach (abstract₂) is more similar to the buggy code (abstract₁) than the input code (abstract₀). Thus, the injected mutations push the fixed code towards a “buggy” state, exactly what we expect from mutation operators. While our baseline is relatively simple, improvements of few BLEU score points have been treated as “considerable” in neural machine translation tasks [57].

Prediction Classification. Table II shows the raw numbers and percentages of predictions falling into the three categories previously described (i.e., perfect, mutated, and bad predictions). The GMident generated 1,991 (17%) perfect predictions whereas GMident⁻lit 2,132 (21%) perfect predictions. We fed into the trained model a fixed piece of code, which the model has never seen before, and the model was able to perfectly predict the buggy version of that code, i.e., to replicate the original bug. No information about the type of mutation operations to perform nor mutation locations are provided to the model. The fixed code is its only input. It is also important to note that, for the perfect predictions of the GMident⁻lit model, we can transform the entire abstracted code to actual source code by mapping each and every ID to their corresponding value. The perfect predictions generated by GMident can be mapped to actual source code but, in some cases, we might need to randomly generate new literal values.

Finally, GMident and GMident⁻lit are not able to mutate the source code in 3,548 (30%) and 2,644 (26%) cases, respectively. While the percentages are non-negligible, it is still worth noting that overall, in 69% and 73% of cases, the models are able to mutate the code. These instances of bad predictions can be seen as cases where the model is unsure on how to properly mutate the code. There are different strategies that could be adopted to force the model to mutate the code (e.g., penalize during training predictions that are equal to the input code, modify the inference step, or using beam search and select the prediction that is not equal to the input).

Syntactic Correctness. Table III reports the percentage of syntactically correct predictions performed by the model. More than 98% of the model predictions are lexically and syntactically correct. When focusing on mutated predictions, the syntactic correctness is still very high (>96%). This indicates that the model is able to learn the correct syntax rules from the abstracted representation we use as input/output of the model. While we do not report statistics on the compilability of the mutants, we can assume that the ~20% perfect predictions generated by the models are compilable, since they correspond to actual buggy code that was committed to software repositories. This means that the compilability rate of the mutants generated by our models is at least around 20%. This is a very conservative estimation that does not consider the mutated

Table I: BLEU Score

| Model     | abstract₁ - abstract₀ (mutation) | abstract₂ - abstract₀ (mutation) | 2.5th percentile δ = mutation - baseline |
|-----------|---------------------------------|---------------------------------|------------------------------------------|
| GMident   | 71.85                           | 76.68                           | +5.63                                    |
| GMident⁻lit| 70.07                           | 78.92                           | +7.97                                    |
| M₁        | 67.18                           | 82.16                           | +17.01                                   |
| M₂        | 51.58                           | 50.96                           | +1.01                                    |
| M₃        | 81.89                           | 83.15                           | +0.94                                    |
| M₄        | 67.04                           | 78.87                           | +12.47                                   |
| M₅        | 65.68                           | 77.73                           | +13.51                                   |

Table II: Prediction Classification

| Model     | Perfect pred. | Mutated pred. | Bad pred. | Total |
|-----------|---------------|---------------|-----------|-------|
| GMident   | 1,991 (17%)   | 6,020 (52%)   | 3,548 (31%) | 11,559 |
| GMident⁻lit| 2,132 (21%)   | 5,240 (52%)   | 2,644 (27%) | 10,016 |
| M₁        | 1,348 (45%)   | 1,500 (49%)   | 190 (6%)   | 3,038 |
| M₂        | 65 (9%)       | 635 (91%)     | 1 (0%)     | 710   |
| M₃        | 392 (33%)     | 967 (33%)     | 1,603 (54%)| 2,962 |
| M₄        | 721 (29%)     | 1,453 (57%)   | 358 (14%)  | 2,532 |
| M₅        | 366 (34%)     | 681 (63%)     | 33 (3%)    | 1,080 |
predictions. Brown et al. [22] achieved a compilability rate of 14%. Moreover, “the majority of failed compilations (64%) arise from simple parsing errors” [22], whereas we achieve a better estimated compilability and a high percentage of syntactically correct predictions.

Token-based Operations. Table IV shows the classification of predictions based on the token-based operations performed by the models. $GM_{ident}$ and $GM_{ident−lit}$ generated predictions that resulted in the insertion of tokens in 1% of the cases, changed nodes in 5% and 3% of the cases, and deletion of tokens in 63% and 69%, respectively. While most of the predictions resulted in token deletions, it is important to highlight that our models are able to generate predictions that insert and change tokens. We investigated whether these results were in-line with the actual data, or whether this was due to a drawback of our learning. We found that the operations performed in the bug-fixes we collected are: 72% insertion, 8% deletion, and 20% changes. This means that bug-fixes mostly tend to perform insert operations (e.g., adding an if statement to check for an exceptional condition), which means that when learning to inject bugs by mutating the code, it is expected to observe a vast majority of delete operations (see Table IV).

AST-based Operations. Table V reports the percentage of mutated predictions that share the same set or list of operations that would have led to the actual buggy code. $GM_{ident}$ and $GM_{ident−lit}$ generate a significant amount of mutated predictions which perform the same set (16% and 24% respectively) or the same type and frequency (14% and 21%) of operations w.r.t. the buggy code. This shows that our models can still generate mutated code that is similar to the actual buggy code.

Summary for RQ1. Our models are able to learn from bug-fixes how to mutate source code. The general models generate mutants that perfectly correspond to the original buggy code in ~20% of the cases. The mutants generated are mostly syntactically correct (>98%) and with an estimated compilability rate of at least 20%.

RQ2: Can we train different mutation models? We present the performance of the cluster models $M_1$,...,$M_5$ based on the metrics introduced before. Each model has been trained and evaluated on the corresponding cluster of TPs, with respective sizes of $C_1 = 30,385$, $C_2 = 7,016$, $C_3 = 29,625$, $C_4 = 25,320$, and $C_5 = 10,798$.

BLEU Score. Table II shows the BLEU scores obtained by the five models. The BLEU scores for these models (mutation column) are relatively high, between 77.73 and 83.15 (with exception of model $M_2$), meaning that the mutated code generated by such models is a very close translation of the actual buggy code. Looking at the distribution of the deltas, we can notice that all the 2.5th percentiles are greater than zero, meaning that the models achieve a BLEU score which is statistically better than the baselines. Even in the case of $M_2$, for which the global BLEU score is slightly lower than the baseline, when the comparison is performed over 2,000 random samples, it outperforms the baseline.

Prediction Classification. Table II shows the raw numbers and percentages of predictions falling in the three categories we defined. Model $M_1$ achieves the highest percentage of Perfect predictions (44%), followed by model $M_5$ (33%) and model $M_4$ (28%). This means that, given a fixed code, it is very likely that at least one of the models would predict the actual buggy code, as well as other interesting mutants. At the same time, the percentages of Bad predictions decreased significantly (except for $M_3$) w.r.t. the general models.

The high percentage of bad predictions for $M_3$ can be partially explained by looking at the actual data in the cluster. The TPs in $C_3$ exhibits small transformations of the code. This is also noticeable from Table II which shows a baseline BLEU score of 81.89 (the highest baseline value), which means that the input fixed code is already a close translation of the corresponding buggy code. This may have led the model to fall in a local minimum where the mutation of the fixed code is the fixed code itself. Solutions for this problem may include: (i) further partitioning the cluster into more cohesive sub-clusters; (ii) allowing more training times/epochs for such models; (iii) implementing changes in the training/inference that we discussed previously.

Syntactic Correctness. Table III reports the percentage of syntactically correct predictions performed by the models. Overall, the cluster model results are in-line with what was found for the general models, with an overall syntactic correctness between 94.63% and 98.18%. When focusing only on the mutated predictions, we still obtain very high syntactic correctness, between 91% and 97%. In terms of compilability, we could expect even better results for these models, given the higher rate of perfect predictions (which are likely to be compilable) generated by the cluster models.

Token-based Operations. Table IV shows the classification
Table V reports the percentage of the distribution of unique mutants. We showed that clusters models generate a high percentage of perfect predictions (between 9% and 45%) with a syntactic correctness between 94% and 98%. Even when the models generate mutants that are not perfect predictions, they usually apply a similar set of operations w.r.t. the buggy code. Furthermore, the trained models generate diverse mutants.

**RQ3: What are the characteristics of the mutants generated by the models?** Figure 2 shows examples of perfect and mutated predictions generated by the general models, as well as diverse mutants generated by the cluster models for the same input code. At the top, each example shows the input code (fixed) followed by the generated mutated code.

The first example is a perfect prediction generated by the general model. The top line is the abstracted fixed code fed to the model, and the bottom line represents the output of the model, which perfectly corresponds to the target buggy code. The fixed code first removes the element at index from VAR_2, assigning it to the VAR_1, and then, if the newly defined variable is not null, it invokes the method get and returns its value, otherwise it returns null. The general model was able to apply different transformations of the code to generate the original buggy code, which invokes all the methods in sequence and returns the value. If the removed element is null, the buggy code will throw an exception when invoking the method get. This transformation of the code does not fit in any existing mutation operator category.

Next, we report an interesting case of mutated prediction. In this case, we used the mapping $M$ to automatically map back every identifiers and literals, showing the ability to generate actual source code from the output of the model. The model replaced the $equals()$ method call with an equality expression ($==$). This shows how the model was able to learn common bugs introduced by developers when comparing objects. Note that the method name $equals$ is an idiom, which allowed the model to learn this transformation.

Finally, the bottom part of Figure 2 shows the five mutants generated by the cluster models for the same fixed code (F) provided as input. In this case, we used the mapping $M$ to retrieve the source code from the output of the models. We selected this example because it shows both interesting mutations and some limitations of our approach. $M_1$ was not able to generate a mutant and returned the same input code (bad prediction). $M_2$ generated a mutant by removing the entire method body. While this appears like a trivial mutation, it is still meaningful as the method is not supposed to return a value, but only perform computations that will result in some side-effects in the class. This means that the test suite should carefully evaluate the state of the class after the invocation of the mutant. Mutants generated by $M_3$ and $M_4$ are the most interesting. They both introduce an infinite-loop, but in two different ways. $M_3$ deletes the increment of the $rowCount$.
variable, whereas $M_4$ resets its value to 1 at each iteration. Finally, $M_5$ changes the if condition and introduces an infinite loop similarly to the model $M_3$. However it also deletes the variable definition statement for $rowCount$, making the mutant not compilable. All the predictions (including perfect, mutated, and diverse) are available in our appendix [37].

In the manual evaluation, three judges analyzed a total of 430 samples (90, 82, 86, 89, and 83 from $M_1$, $M_2$, $M_3$, $M_4$, $M_5$, respectively). For each sample instance, the judges agreed that the mutated code appeared to have a different functional behavior w.r.t. the original input code. Only one case was debated, corresponding to a mutant which was created by deleting a print call. In this case, the functional behavior may or may not have been changed, depending on whether the output console is assessed as part of the behavior of the method. Thus, all the instances except one were evaluated as actual mutants that introduced a buggy behavior.

Figure 3 shows a heatmap of the frequency of mutation operations for each trained model. The intensity of the color represents the frequency with which a particular operation (specified by the row) was performed by the particular cluster model (columns). Due to space constraints, the rows of the heatmap contain only a subset of the 85 unique types of operations performed by the models, i.e., only those performed in at least 5% of the mutations by at least one model.

We also highlighted in red boxes the peculiar, most frequent operations performed by each model. $M_1$ appears to focus on deletion of method calls; $M_3$ on deletion and replacement of an argument in a method call; $M_4$ mostly operates on if-else blocks and its logical conditions; $M_5$ focuses on deleting and replacing variable assignments. Finally, it is worth noting the large variety of operations performed by $M_2$, ranging from addition, deletion, and replacement of method calls, variable assignments, arguments, etc. This might also explain the lower BLEU score achieved by the latter model, which performs large and more complex operations w.r.t. the other models which tend to focus on a smaller set of operations. Differences among the mutation models can also be appreciated by the number of different mutation operations performed for each mutant. The models $M_1$, $M_2$, $M_3$, $M_4$, $M_5$ performed 1.19, 3.48, 1.42, 1.93, 2.02 average number of operations for each mutant, respectively.

Summary for RQ3. The mutation models are capable of performing a diverse set of operations to mutate source code.

V. THREATS TO VALIDITY

Construct validity. To have enough training data, we mined bug-fixes in GitHub, rather than using curated bug-fix datasets such as Defects4J, while still very useful but limited in size. To mitigate imprecisions in our datasets (i.e., commits not related to bug fixes), we manually analyzed a sample of the extracted commits. Moreover, we disregarded large commits (too many files/AST operations) that might refer to tangled changes.

Internal validity. In assessing whether the generated mutants change the behavior of the code, we analyzed the mutated method in isolation (i.e., not in the context of its system). This might have introduced imprecisions that were mitigated by assigning multiple evaluators to the analysis of each mutant.

External validity. We only focused on Java code. However, the learning process is language-independent and the whole infrastructure can be instantiated for different languages by replacing the lexer, parser and AST differencing tools. We only focused on methods having no more than 50 tokens. In our appendix 37 we report experimental results on larger methods (between 50 and 100 tokens) using the same configuration of the network and training epochs. The trained model was still able to generate ∼6% of perfect predictions. We are confident that with more training time and parameters’ tuning better results can be obtained for larger methods.

VI. RELATED WORK

Brown et al. [22] leveraged bug-fixes to extract syntactic-mutation patterns from the diffs of patches. Our approach is novel and differs from Brown et al. in several aspects:

• Rather than extracting all possible mutation operators from syntactic diffs, we automatically learn mutations from the data;
• Rather than focusing, in isolation, on contiguous lines of code changed in the diff, we learn the mutation in its context (e.g., method). This allows learning which type and variation of mutation operator is more likely to be effective, given the current context (i.e., methods, variables, scopes and blocks);
• Our approach is able to automatically mutate identifiers and literal by inserting idioms (based on what learned) in the new mutant. When the model suggests to mutate a literal with another unknown literal, it is generated randomly. Brown’s et al. approach does not contemplate the synthesis of new identifiers (cfr. Section 2.3 [22]);
• Rather than extracting a single mutation pattern, we can learn co-occurrences and combinations of multiple mutations;
• While the approach by Brown et al. randomly applies mutation operators to any code location unless the user specifies a rule for that, our approach automatically applies, for a given piece of code, the mutation(s) that according to the learning might reflect likely bugs occurring in such a location. While limiting mutants only to the most suitable ones for each location might not be always necessary, because one can apply as many mutants as possible to increase fault detection, this could lead to an overestimate of at test suite effectiveness or to more effort to unnecessarily augment a test suite. In a view of test suite optimization, an approach that learns where and how to mutate code is therefore desirable.

Currently, a direct comparison between the two approaches was not viable as the approaches have been developed for different programming languages (C vs. Java).

Different general-purpose mutation frameworks have been defined in the literature, including µJava [58], Jester [59], Major [10], Jumble [60], PIT [61], and javaLanche [62]. The main novelty of our work over those approaches is the automation of the learning and application of the mutation. Relevant to our work are also studies investigating the relationship between mutants and real faults. Andrews et al. [7], [8] showed that carefully selected mutants can provide
a good assessment of a test suite’s ability to catch real faults and hand-seeded faults can underestimate the test suite’s bug detection capability. Daran and Thévenod-Fosse [9] found that the set of errors produced by carefully selected mutants and real faults with a given test suite are quite similar, while Just et al. [13] reported that some types of real faults are not coupled to mutants and highlighted the need for new mutation operators. Finally, Chekham et al. [14] showed that strong mutation testing yields high fault revelation, while this is not the case for weak mutation testing. Our work builds on these studies, cementing an approach for learning mutants from real bug fixes. Hence, we avoid the need for manually selecting the mutants to inject and increase the chances of generating mutants representative of real bugs.

Allamanis et al. [63] generate tailored mutants, e.g., exploiting API calls occurring elsewhere in the project and show that tailored mutants are well-coupled to real bugs. Differently from them, we automatically learn how to mutate code from an existing dataset of bugs rather than using heuristics.

VII. CONCLUSION

We presented the first approach to automatically learn mutants from existing bug fixes. The evaluation we performed highlights that the generated mutants are similar to real bugs, with 9% to 45% of them (depending on the model) reintroducing in the fixed code (provided as input) the actual bug. Moreover, our models are able to learn the correct code syntax, without the need for syntax rules as input. We release the data and code to allow researchers to use it for learning other transformations of code [37]. Future work includes additional fine tuning of the RNN parameters to improve performance.

REFERENCES

[1] R. G. Hamlet, “Testing programs with the aid of a compiler,” TSE, vol. 3, no. 4, pp. 279–290, Jul. 1977.
[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” Computer, vol. 11, no. 4, pp. 34–41, April 1978.
[3] G. Fraser and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in SIGSOFT/SIEE ’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 15th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pp. 416–419.
[4] ——, “Whole test suite generation,” IEEE Trans. Software Eng., vol. 39, no. 2, pp. 276–291, 2013.
[5] S. Mouchawarh, L. C. Briand, Y. Labiche, and M. Di Penta, “Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments,” IEEE Trans. Software Eng., vol. 37, no. 2, pp. 161–187, 2011. [Online]. Available: https://doi.org/10.1109/TSE.2010.32
[6] L. C. Briand, M. Di Penta, and Y. Labiche, “Assessing and improving state-based class testing: A series of experiments,” IEEE Trans. Software Eng., vol. 30, no. 11, pp. 770–793, 2004.
[7] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in 27th International Conference on Software Engineering (ICSE 2005), 15–21 May 2005, St. Louis, Missouri, USA, 2005, pp. 402–411.
[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” IEEE Trans. Software Eng., vol. 32, no. 8, pp. 608–624, 2006.
[9] M. Daran and P. Thévenod-Fosse, “Software error analysis: A real case study involving real faults and mutations,” in Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSSTA 1996, San Diego, CA, USA, January 8-10, 1996, 1996, pp. 158–171.
[10] 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 2014 International Symposium on Software Testing and Analysis, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.
[11] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, “Assessing test case prioritization on real faults and mutants,” in Proceedings of the 34th International Conference on Software Maintenance and Evolution, ser. ICSE ’12. Piscataway, NJ, USA: IEEE Press, 2012, p. to appear.
[12] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (i),” in Automated Software Engineering (ASE), 2013 30th IEEE/ACM International Conference on. IEEE, 2015, pp. 201–211.
[13] 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 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 654–665.
[14] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, “An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption,” in Proceedings of the 39th International Conference on Software Engineering, ser. ICSE ’17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 597–608. [Online]. Available: https://doi.org/10.1109/ICSE.2017.61
[15] A. J. Offutt, A. Lee, G. Rothermel, R. H. UnIch, and C. Zapf, “An experimental determination of sufficient mutant operators,” ACM Trans. Softw. Eng. Methodol., vol. 5, no. 2, pp. 99–118, 1996.
[16] S. Kim, J. A. Clark, and J. A. McDermid, “Investigating the effectiveness of object-oriented testing strategies using the mutation method,” Softw. Test. Verif. Reliab., vol. 11, no. 3, pp. 207–225, 2001. [Online]. Available: https://doi.org/10.1002/stvr.238
[17] R. Jabbarvand and S. Malek, “android: an energy-aware mutation testing framework for android,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 208–219.
[18] K. Wang, “Maulloy: an automated mutation system for alloy,” Ph.D. dissertation, 2015.
[19] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, “Towards mutation analysis of android apps,” in 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), April 2015, pp. 1–10.
[20] A. J. Offutt and R. H. UnIch, “Mutation testing for the new century,” W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ch. Mutation 2000: Uniting the Orthogonal, pp. 34–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=571305.571314
[21] M. L. Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, “Enabling mutation testing for android apps,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 233–244.
[22] D. B. Brown, M. Vaughan, B. Liblit, and T. Reps, “The care and feeding of wild-caught mutants,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 511–522. [Online]. Available: http://dl.acm.org/citation.cfm?id=3106237.3106280
[23] T. Zimmermann, N. Nagappan, H. C. Galt, E. Giger, and B. Murphy, “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process,” in Proceedings of the 7th joint conference of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009, 2009, pp. 91–100.
[24] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, 2016, pp. 297–308.
[25] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, 2016, pp. 87–98.
[26] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Bug localization with combination of deep learning and information retrieval,” in Proceedings of the 25th International Conference on Program
