CODIT: Code Editing with Tree-Based Neural Machine Translation

Saikat Chakraborty
Columbia University
New York, USA
saikatc@cs.columbia.edu

Miltiadis Allamanis
Microsoft Research
Cambridge, UK
miallama@microsoft.com

Baishakhi Ray
Columbia University
New York, USA
rayb@cs.columbia.edu

***Abstract*—The way developers edit day-to-day code tends to be repetitive, often using existing code elements. Many researchers have tried to automate repetitive code changes by learning from specific change templates which are applied to limited scope. The advancement of Neural Machine Translation (NMT) and the availability of vast open-source evolutionary data opens up the possibility of automatically learning those templates from the wild. However, unlike natural languages, for which NMT techniques were originally devised, source code and its changes have certain properties. For instance, compared to natural language, source code vocabulary can be significantly larger. Further, good changes in code do not break its syntactic structure. Thus, deploying state-of-the-art NMT models without adapting the methods to the source code domain yields sub-optimal results.**

To this end, we propose a novel Tree based NMT system to model source code changes and learn code change patterns from the wild. We realize our model with a change suggestion engine: CODIT and train the model with more than 30k real-world changes and evaluate it on 6k patches. Our evaluation shows the effectiveness of CODIT in learning and suggesting patches. CODIT also shows promise generating bug fix patches.

I. INTRODUCTION

Developers often edit source code to add new features, fix bugs, or maintain existing functionality (*e.g.,* API updates, refactoring, *etc*.). Recent research has shown that these edits are often repetitive [1]–[3]. Moreover, the code components (*e.g.,* token, sub-trees, *etc*.) used to build the edits are often times taken from the existing codebase [4], [5]. However, manually applying such repetitive edits can be tedious and error-prone [6]. Thus, it is important to automate code changes, as much as possible, to reduce the developers' burden.

There is a significant amount of industrial and academic work on automating code changes. For example, modern IDEs support specific types of automatic changes (*e.g.,* refactoring, adding boiler-plate code [7], [8], *etc*) Many research tools aim to automate some types of edits, *e.g.,* API related changes [9]–[14], refactoring [15]–[18], *etc*. Researchers have also proposed to automate generic changes by learning either from example edits [19], [20] or from similar patches applied previously to source code [2], [3], [21].

While the above lines of work are promising and have shown initial success, they either rely on predefined change templates or require domain-specific knowledge: both are hard to generalize in the larger context. However, all of them leverage, in someway, common edit patterns. Given that a large amount of code and its change history is available thanks to software forges like GitHub, Bitbucket, *etc*., a natural question arises: *Can we learn to predict general code changes by learning them in the wild?*

Recently there has been a lot of interest in using machine learning techniques to model and predict source code from real world [22]. However, modeling changes is different from modeling general code generation, since modeling changes is conditioned on the previous version of the code. In this work, we research whether machine learning models can capture the repetitiveness and statistical characteristics of code edits that developers apply daily. Such models can automate code changes, bug fixes, and other software-evolution-related tasks.

To this end, we use the general framework of Neural Machine Translation (NMT) to learn from past code changes and suggest future edits. In natural language processing (NLP), NMT is widely used to translate text from one language (*e.g.,* English) to another (*e.g.,* Spanish). NMT seem to be a natural fit for modeling code changes since it can learn the transformation (*i.e.* edits) from an original to a changed version of the code. Essentially, these models learn the probability distribution of changes and assign higher probabilities to plausible code edits and lower probabilities to less plausible ones. In fact, Tufano *et al.* [23], [24] show an initial promise of using standard sequence-to-sequence translation model (seq2seq) for predicting abstract code changes.

In this work, our goal is to predict concrete code changes. To do that, we design an encoder-decoder-based NMT model that works in two steps. First, it learns and suggests structural changes in code using a tree-based translation model. Structural changes are suggested in the form of Abstract Syntax Tree (AST) modifications. Tree-based models, unlike their token-based counterparts, capture the rich structure of code and produce syntactically correct patches. Then, we concretize each code fragment generating the tokens conditioned on the AST generated in the first step. Here, given the type of each leaf node in the syntax tree, our model recommends concrete tokens of the correct type while respecting scope information. We combine these two models to realize CODIT, a code change suggestion engine, which takes a code fragment as input and generates potential edits of that snippet.

Synthesizing patches (or code in general) is a tough problem [25]. When we view code generation as a sequence of token generation problem, the space of the possible actions

becomes too large. Existing statistical language modeling techniques (*e.g.,* Raychev *et al.* [11], Tu *et al.* [26]) endorse the action space with a probability distribution, which effectively reduces the action space significantly since it allows to consider only the subset of probable actions. The action space can be further reduced by relaxing the problem of concrete code generation to some form of abstract code generation, *e.g.,* generating code sketches [27], abstracting token names [23], *etc*. For example, Tufano *et al.* reduce the effective size of the action space to $3.53 \cdot 10^{10}$ by considering abstract token names [23]. While considering all possible ASTs allowed by the language's grammar, the space size grows to $1.81 \cdot 10^{35}$. In this work, a probabilistic grammar further reduces the effective action space to $3.18 \cdot 10^{10}$, which is significantly lower than previous methods. Thus, the reduction of the action space allow us to search for code more efficiently.

In this work, we collect a new dataset — *Code-Change-Data*, consisting of 44372 patches from 48 open-source GitHub projects collected from Travis Torrent [28]. Our experiments show CODIT achieves 15.81% patch suggestion accuracy in the top 10 suggestions; this result outperforms a Seq2Seq based model by 98.16%. CODIT can also predict 33.61% of structural changes accurately within the top 10 suggestion. We also evaluate CODIT on *Pull-Request-Data* proposed by Tufano *et al.* [24]. Our evaluation shows that CODIT suggests 14.36% of correct patches in the top 10 outperforming Seq2Seq-based model by 108.2%. Further evaluation on CODIT's ability to suggest bug-fixing patches in Defects4J shows that CODIT suggests 16 complete fixes and 9 partial fixes out of 80 bugs in Defects4J. In summary, our key contributions are:

- We propose a novel tree-based code editing machine learning model that leverages the rich syntactic structure of code and generates syntactically correct patches. To our knowledge, we are the first to model code changes with tree-based machine translation.
- We collect a large dataset of 44k real code changes.
- We implement our approach, CODIT, and exhaustively evaluate the viability of using CODIT for suggesting patch templates, concrete changes, and bug fixes.

We will release our code and data for broader dissemination.

II. BACKGROUND

Modeling Code Changes. Generating source code using machine learning models has been explored in the past [26], [29]–[31]. These methods model a probability distribution $p(c|\kappa)$ where c is the generated code and κ is any contextual information upon which the generated code is conditioned. In this work, we generate *code edits*. Thus, we are interested in models that predict code given its previous version. We achieve this using NMT-style models, which are a special case of $p(c|\kappa)$, where c is the new and κ is the previous version of the code. NMT allows us to represent code edits with a single end-to-end model, taking into consideration the original version of a code and defining a conditional probability distribution of the target version. Similar ideas have been explored in NLP for paraphrasing [32].

Neural Machine Translation Models. NMT models are usually a cascade of an *encoder* and a *decoder*. The most common model is sequence-to-sequence (seq2seq) [33], where the input is treated as a sequence of tokens and is encoded by a sequential encoder (*e.g.,* biLSTM). The output of the encoder is stored in an intermediate representation. Next, the decoder using the encoder output and another sequence model, *e.g.,* an LSTM, generates the output token sequence. In NMT, several improvements have been made over the base seq2seq model, such as *attention* [33] and *copying* [34]. Attention mechanisms allows decoders to automatically search information within the input sequence when predicting each target element. Copying, a special form of an attention mechanism, allows the model to directly copy elements of the input to the target. We employ both attention and copying mechanisms in this work.

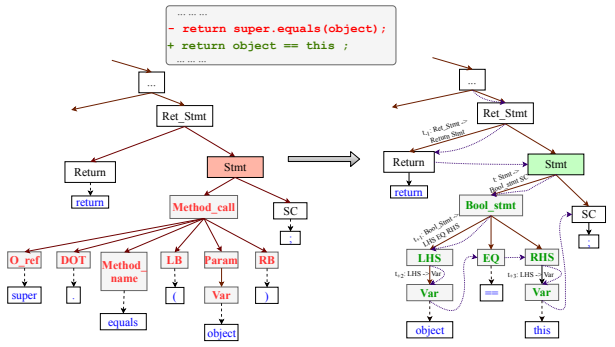
Grammar-based modeling. Context Free Grammars (CFG) has been used to describe the syntax of programming languages [35] and natural language [36], [37]. A CFG is a tuple $G = (N, \Sigma, P, S)$ where N is a set of non-terminals, Σ is a set of terminals, P is a set of production rules in the form of $\alpha \rightarrow \beta$ and $a \in N$, $b \in (N \cup \Sigma)^*$, and S is the start symbol. A sentence (*i.e.* sequence of tokens) that belongs to the language defined by G can be parsed by applying the appropriate derivation rules from the start symbol S. A common technique for generation of utterances is to expand the left-most, bottom-most non-terminal until all non-terminals have been expanded. Probabilistic context-free grammar (PCFG) is an extension of CFG, where each production rule in associated with a probability, *i.e.* is defined as (N, Σ, P, Π, S) where Π defines a probability distribution for each production rule in P conditioned on α.

III. MOTIVATING EXAMPLE

Figure 1 illustrates an example of our approach. Here, the original code fragment `return super.equals(object)` is edited to `return object == this`. CODIT takes these two code fragments along with their context, for training. While suggesting changes, *i.e.*, during test time, CODIT takes as input the previous version of the code and generates its edited version.

CODIT operates on the parse trees of the previous (t_p) and new (t_n) versions of the code, as shown in Figure 1(a) (In the rest of the paper, a subscript or superscript with p and n correspond to previous and new versions respectively). In Figure 1, changes are applied only to the subtree rooted at the $Method\_call$ node. The subtree is replaced by a new subtree (t_n) with `Bool_stmt` as a root. The deleted and added subtrees are highlighted in **red** and **green** respectively.

While modeling the edit, CODIT first predicts the structural changes in the parse tree. For example, in Figure 1(a) CODIT first generates the changes corresponding to the subtrees with dark nodes and red edges. Next the structure is concretized by generating the token names (terminal nodes). This is realized by combining two models: (i) a tree-based model predicting



(a) **Example of correctly suggested change by CODIT along with the source and target parse trees. The deleted and added nodes are marked in red and green respectively.**

(b) **Sequence of grammar rules extracted from the parse trees.**

(c) **Token generation. Token probabilities are conditioned based on terminal types generated by tree translator (see figure 1(a))**

Fig. 1: **Motivating Example**

the structural change (see IV-A) followed by a (ii) a token generation model conditioned on the structure generated by the tree translation model (see IV-B).

Tree Translator. The tree translator is responsible for generating structural changes to the tree structure. A machine learning model is used to learn a (probabilistic) mapping between t_p and t_n. First, a tree encoder, encodes t_p computing a distributed vector representation for each of the production rules t_p sequentially yielding the distributed representation for the whole tree. Then, the tree decoder uses the encoded representations of t_p to sequentially select rules from the language grammar to generate t_n. The tree generation starts with the `root` node. Then, at each subsequent step, the bottom-most, left-most non-terminal node of the current tree is expanded. For instance, in Figure 1(a), at time step `t`, node `Stmt` is expanded with rule **`Stmt` → `Bool_Stmt SC`**. When the tree generation process encounters a terminal node, it records the node type to be used by the token generation model and proceeds to the next non-terminal. In this way, given the LHS rule sequences of Figure 1(b) the RHS rule sequences is generated.

Token Generator: The token generator generates concrete tokens for the terminal node types generated in the previous step. The token generator is a standard seq2seq model with attention and copying [33] but constrained on the token types generated by the tree translator. To achieve this, the token generator first encodes the token string representation and the node type sequence from t_p. The token decoder at each step probabilistically selects a token from the vocabulary or copies one from the input tokens in t_p. However, in contrast to traditional seq2seq where the generation of each token is only conditioned on the previously generated and source tokens, we additionally condition on the token type that has already been generated by the tree model. Figure 1(c) shows this step: given the token sequence of the original code `< super . equals ( object ) >` and their corresponding token types (given in dark box), the new token sequences is generated `< object == this >`.

IV. Tree-based Neural Translation Model

We decompose the task of predicting code changes in two stages: First, we learn and predict the structure (syntax tree) of the edited code. Then, given the predicted tree structure, we concretize the code. We factor the generation process as

$$P(c_n|c_p) = P(c_n|t_n, c_p)P(t_n|t_p)P(t_p|c_p), \quad (1)$$

and our goal is to find $\hat{c}_n$ such that $\hat{c}_n = argmax_{c_n} P(c_n|c_p)$. Here, c_p is the previous version of the code and t_p is its parse tree, whereas c_n is the new version of the code and t_n its parse tree. Note that parsing a code fragment is unambiguous, *i.e.* $P(t_p|c_p) = 1$. Thus, our problems take the form

$$\hat{c}_n = \arg\max_{c_n, t_n} \underbrace{P(c_n|t_n, c_p)}_{\mathscr{M}_{token}} . \underbrace{P(t_n|t_p)}_{\mathscr{M}_{tree}} \quad (2)$$

Equation 2 has two parts. First, it estimates the changed syntax tree $P(t_n|t_p)$. We implement this with a tree-based encoder-decoder model (section IV-A). Next, given the predicted syntax tree t_n, we estimate the probability of the concrete edited code with $p(c_n|t_n, c_p)$ (Section IV-B).

A. Tree Translation Model ($\mathscr{M}_{tree}$)

The goal of $\mathscr{M}_{tree}$ is to model the probability distribution of a new tree (t_n) given a previous version of the tree (t_p). For any meaningful code the generated tree is syntactically correct. We represent the tree as a sequence of grammar rules generations following the CFG of the underlying programming language. The tree is generated by iteratively applying CFG expansions at the left-most bottom-most non-terminal node (`frontier_node`) starting from the start symbol.

For example, consider the tree fragments in Figure 1(a). Figure 1(b) shows the sequence of rules that generate those trees. For example, in the right tree of Figure 1(a), the node `Ret_Stmt` is first expanded by the rule: `Ret_Stmt`→`Return Stmt`. Since, `Return` is a terminal node, it is not expanded any further. Next, node `Stmt` is expanded with rule: `Stmt`→`Bool_Stmt SC`. The tree is further expanded with `Bool_Stmt`→`LHS EQ RHS`, `LHS`→`Var`, and `RHS`→`Var`. During the tree generation process, we apply these rules to yield the tree fragment of the next version.

In particular, the tree is generated by picking CFG rules at each non-terminal node. Thus, our model resembles a Probabilistic Context-Free Grammar (PCFG), but the probability of each rule depends on its surroundings. The neural network models the probability distribution, $P(R_k^n|R_1^n, ...R_{k-1}^n, t_p)$: At

time k the probability of a rule depends on the input tree t_p and the rules $R_1^n, ...R_{k-1}^n$ that have been applied so far. Thus, the model for generating the syntax tree t_n is given by

$$P(t_n|t_p) = \prod_{k=1}^{\tau} P(R_k^n|R_1^n, ...R_{k-1}^n, t_p) \quad (3)$$

Encoder: The encoder encodes the sequence of rules that construct t_p. For every rule R_i^p in t_p, we first transform it into a single learnable distributed vector representation $\boldsymbol{r}_{R_i^p}$. Then, the LSTM encoder summarizes the whole sequence up to position i into a single vector $\boldsymbol{h}_i^p$.

$$\boldsymbol{h}_i^p = f_{LSTM}(\boldsymbol{h}_{i-1}^p, \boldsymbol{r}_{R_i^p}) \quad (4)$$

This hidden vector contains information about the particular rule being applied and the previously applied rules. Once all the rules in t_p are processed, we get a final hidden representation ($\boldsymbol{h}_\tau^p$). The representations at each time step $(\boldsymbol{h}_1^p, \boldsymbol{h}_2^p, ..., \boldsymbol{h}_\tau^p)$ are used in the decoder to generate rule sequence for the next version of the tree. The parameters of the LSTM and the rules representations $\boldsymbol{r}_{R_i^p}$ are randomly initialized and learned jointly with all other model parameters.

Decoder: Our decoder has an LSTM with an attention mechanism as described by Bahdanau *et al.* [33]. The decoder LSTM is initialized with the final output from the encoder, *i.e.* $\boldsymbol{h}_0^n = \boldsymbol{h}_\tau^p$. At a given decoding step k the decoder LSTM changes its internal state in the following way,

$$\boldsymbol{h}_k^n = f_{LSTM}(\boldsymbol{h}_{k-1}^n, \boldsymbol{\psi}_k), \quad (5)$$

where $\boldsymbol{\psi}_k$ is computed by the attention-based weighted sum of the inputs $\boldsymbol{h}_j^p$ as [33] in , *i.e.*

$$\boldsymbol{\psi}_k = \sum_{j=1}^{\tau} softmax(\boldsymbol{h}_k^{n^T} \boldsymbol{h}_j^p) \boldsymbol{h}_j^p \quad (6)$$

Then, the probability over the rules at the kth step is:

$$P(R_k^n|R_1^n, ...R_{k-1}^n, t_p) = softmax(W_{tree} \cdot \boldsymbol{h}_k^n + \mathbf{b}_{tree}) \quad (7)$$

At each timestep, we pick a derivation rule R_k^n following equation (7) to expand the `frontier_node` (n_f^t) in a depth-first, left-to-right fashion. When a terminal node is reached , it is recorded to be used in $\mathscr{M}_{token}$ and the decoder proceeds to next non-terminal. In Equation (7), W_{tree} and $\mathbf{b}_{tree}$ are parameters that are jointly learned along with the LSTM parameters of the encoder and decoder.

B. Token Generation Model ($\mathscr{M}_{token}$)

We now focus on generating a concrete code fragment c a sequence of tokens $(x_1, x_2, ...)$. For the edit task, the probability of an edited token x_k^n depends not only on the tokens of the previous version $(x_1^p, ..., x_m^p)$ but also on the previously generated tokens $x_1^n, ..., x_{k-1}^n$. The next token x_k^n also depends on the token type (θ), which is generated by $\mathscr{M}_{tree}$. Thus,

$$P(c_n|c_p, t_n) = \prod_{k=1}^{m'} P(x_k^n|x_1^n, ..., x_{k-1}^n, \{x_1^p, ..., x_m^p\}, \theta_k^n) \quad (8)$$

Here, θ_k^n is the node type corresponding to the generated terminal token x_k^n. Note that, the token generation model can be viewed as a conditional probabilistic translation model where token probabilities are conditioned not only on the context but also on the type of the token (θ_*^*). Similar to $\mathscr{M}_{tree}$, we use an encoder-decoder. The encoder encodes each token and corresponding type of the input sequence into a hidden representation with an LSTM (figure 1(c)). Then, for each token (x_i^p) in the previous version of the code, the corresponding hidden representation (s_i^p) is given by: $\boldsymbol{s}_i^p = f_{LSTM}(\boldsymbol{s}_{i-1}^p, enc([x_i^p, \theta_i^p]))$. Here, θ_i^p is the terminal token type corresponding to the generated token x_i^p and $enc()$ is a function that encodes the pair of x_i^p, θ_i^p to a (learnable) vector representation.

The decoder's initial state is the final state of the encoder. Then, it generates a probability distribution over tokens from the vocabulary. The internal state at time step k of the token generation is $\boldsymbol{s}_k^n = f_{LSTM}(\boldsymbol{s}_{k-1}^n, enc(x_i^n, \theta_k^n), \boldsymbol{\xi}_k))$, where $\boldsymbol{\xi}_k$ is the attention vector over the previous version of the code and is computed as in Equation (6). Finally, the probability of the kth target token is computed as

$$P(x_k^n|x_1^n, ..., x_{k-1}^n, \{x_1^p, ..., x_m^n\}, \theta_k^n) = softmax\left(W_{token} \cdot \boldsymbol{s}_k^n + \boldsymbol{b}_{token} + mask(\theta_k^n)\right) \quad (9)$$

Here, W_{token} and $\boldsymbol{b}_{token}$ are parameters that are optimized along with all other model parameters. Since not all tokens are valid for all the token types, we apply a mask that deterministically filters out invalid candidates. For example, a token type of `boolean_value`, can only be concretized into **`true`** or **`false`**. Since the language grammar provides this information, we to create a mask ($mask(\theta_k^n)$) that returns a $-\infty$ value for masked entries and zero otherwise. Similarly, not all variable, method names, type names are valid at every position. We refine the mask-based on the variables, method names and type names extracted from the scope of the change. In the case of method, type and variable names, CODIT allows $\mathscr{M}_{token}$ to generate a special `<unknown>` token. However, the `<unknown>` token is then replaced by the source token that has the highest attention probability (*i.e.* the highest component of $\boldsymbol{\xi}_k$), a common technique in NLP.

V. IMPLEMENTATION

Our tree-based translation model is implemented as an edit recommendation tool, CODIT. CODIT learns source code changes from a dataset of patches. Then, given a code fragment to edit, CODIT predicts potential changes that are likely to take place in the similar context. We implement CODIT extending OpenNMT [38] based on PyTorch. We now discuss CODIT's implementation in details.

Patch Pre-processing. We represent the patches as parse tree format and extract necessary information (*e.g.,* grammar rules, tokens, and token-types) from them.

Parse Tree Representation. As a first step of the training process, CODIT takes a pool of patches as input and parses them. CODIT works at method granularity. For a method patch Δm, CODIT takes the two versions of m: m_p and m_n. Using GumTree, a tree-based code differencing tool [39], it identifies the edited AST nodes. The edit operations are represented as insertion, deletion, and update of nodes *w.r.t.* m_p. For example, in Figure 1(a), **red** nodes are identified as deleted nodes and **green** nodes are marked as added nodes. CODIT then selects the minimal subtree of each AST that captures all the edited nodes. If the size of the tree exceeds a maximum size of max_change_size, we do not consider the patch. CODIT also collects the edit context by including the nodes that connect the root of the method to the root of the changed tree. CODIT expands the considered context until the context exceed a maximum tree size (max_tree_size). During this process, CODIT excludes changes in comments and literals. Finally, for each edit pair, CODIT extracts a pair (AST_p, AST_n) where AST_p is the original AST where change was applied, and AST_n is the AST after the changes. CODIT then converts the ASTs to their parse tree representation such that each token corresponds to a terminal node. Thus, a patch is represented as the pair of parse trees (t_p, t_n).

Information Extraction. CODIT extracts grammar rules, tokens and token types from t_p and t_n. To extract the rule sequence, CODIT traverses the tree in a depth-first pre-order way. From t_p, CODIT records the rule sequence (R_1^p, ..., R_τ^p) and from t_n, CODIT gets (R_1^n, ..., $R_{\tau'}^n$) (Figure 1(b)). CODIT then traverses the parse trees in a pre-order fashion to get the augmented token sequences, *i.e.* tokens along with their terminal node types: (x_*^p, θ_*^p) from t_p and (x_*^n, θ_*^n) from t_n. CODIT traverses the trees in a left-most depth-first fashion. When a terminal node is visited, the corresponding augmented token (x_*^*, θ_*^*) is recorded.

Model Training. We train the tree translation model ($\mathscr{M}_{tree}$) and token generation model ($\mathscr{M}_{token}$) to optimize Equation (3) and Equation (8) respectively using the cross-entropy loss as the objective function. Note that the losses of the two models are independent and thus we train each model separately. In our preliminary experiment, we found that the quality of the generated code is not entirely related to the loss. To mitigate this, we used top-1 accuracy to validate our model. We train the model for a fixed amount of n_{epoch} epochs using early stopping (with patience of $valid_{patience}$) on the top-1 suggestion accuracy on the validation data. We use stochastic gradient descent to optimize the model.

Model Testing. To evaluate the model, we first sort the patches in our dataset chronologically. Then, we take the first 75% from each project as training, 15% as validation, and the rest 10% as testing. This partitioning method reflects how a developer would use a model in real life: a model trained on past patches is used to predict future edits.

To test the model and generate changes, we use beam-search [40] to produce the suggestions from $\mathscr{M}_{tree}$ and $\mathscr{M}_{token}$. First given a rule sequence from the previous version of the tree, we generate K_{tree} different trees reflecting different structural changes. Then for each tree, we generate K_{token} different concrete code. Thus, we generate $K_{tree} \cdot K_{token}$ code fragments. We sort them based on their probability, *i.e.* $log(P(c_n|c_p, t_p)) = log(P(c_n|c_p, t_n) \cdot P(t_n|t_p))$. From the sorted list of generated code, we pick the top K suggestions.

TABLE I: **Summary of datasets used to evaluate CODIT**

Dataset	# Projects	# Train Examples	# Validtion Examples	# Test Examples
Code-Change-Data	48	33093	4448	6831
Pull-Request-Data [41]	3	8535	1073	1058
Defects4J-data [42]	6	22060	2537	117

VI. EXPERIMENTAL DESIGN

We evaluate CODIT for three different types of changes that often appear in practice: (i) code change in the wild, (ii) pull request edits, and (iii) bug repair. We study three different dataset, summarized in Table I.

(i) *Code-Change-Data* is real code change dataset collected from 48 open-source projects from GitHub. These projects also appear in TravisTorrent [28] and have at least 50 commits in Java files. For each project, we collected the revision history of the main branch. For each commit, we record the code before and after the commit for all Java files that are affected. In total we collect 2,41,976 file pairs and from them we extract a set of method pairs affected by the changes. We further remove any pairs, where the change is only in comments and literals. We set $max\_change\_size = 10$ and $max\_tree\_size = 20$. We end up with 44,382 total patches which are input to CODIT's preprocessing pipeline (Section V).

(i) *Pull-Request-Data* is provided by Tufano *et al.* [24] which contains source code changes from merged pull requests from three projects from Gerrit [43]. It consist of 10,666 method pairs. We used the method pairs as is for training and validating CODIT.

(iii) *Defects4J* dataset contains bug-repair data along with test cases to reproduce the bugs from six open-source Java project [42]. We use all the patches from the project history that are in our scope to train CODIT, and test using the bug-repair patches.

A. Evaluation Metric

To evaluate CODIT, we measure for a given code fragment, how accurately CODIT generates patches. We consider CODIT to correctly generate a patch if it exactly matches the original. CODIT produces the top K patches and we compute CODIT's accuracy by counting how many patches are correctly generated in top K. Note that this metric is stricter than semantic equivalence.

To evaluate bug-fix patches from Defects4J, CODIT generates patches. We then run the bug-triggering test cases for every patch. We set a predefined time budget for evaluation. If CODIT can pass the test cases with a newly generated patch within the time budget, we call it a potential patch, which we suggest to the developer for manual checking. Later, we manually investigate the passing patches and check if they are semantically similar to the developer-provided patches. We call

TABLE II: **Performance of CODIT suggesting concrete patches**

Datset	Number of Examples	Top 1 CODIT	Top 1 Seq2Seq	Top 1 Gain	Top 2 CODIT	Top 2 Seq2Seq	Top 2 Gain	Top 5 CODIT	Top 5 Seq2Seq	Top 5 Gain	Top 10 CODIT	Top 10 Seq2Seq	Top 10 Gain
Code-Change-Data	6831	423 (6.19%)	363 (5.31%)	16.5%	788 (11.51%)	410 (6.00%)	92.20%	860 (12.59%)	488 (7.14%)	76.23%	1080 (15.81%)	545 (7.97%)	98.16%
Pull-Request-Data	1058	55 (5.19%)	43 (4.06%)	27.9%	94 (8.88%)	52 (4.91%)	80.76%	129 (12.519%)	68 (6.42%)	89.70%	152 (14.36%)	73 (6.89%)	108.2%

AST_p to be *fixed* when at least one of the patch that passes all the bug-producing test cases, exactly matches the developer provided patch. When all AST_p corresponding to a single bug in Defects4J are fixed, we call that bug *fully fixed*. If CODIT can fix some of the AST_p of a bug, we call that *partially fixed*.

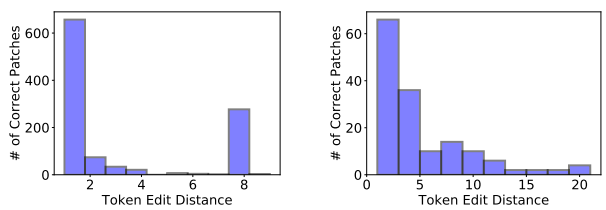
B. Baseline

Tufano *et al.* [24] first showed the potential of a Seq2Seq translation model to predict abstract code changes. Following their description, we build a Seq2Seq model on top of OpenNMT [38]. To validate our Seq2Seq implementation, we compare Seq2Seq performance on Tufano *et al.*'s abstract dataset (*Pull-Request-Data*) and get result similar to those reported in their paper. Since the original model was developed for an abstract dataset, they did not face the vocabulary explosion and the problem of `<unknown>` tokens, as we discussed in Section IV-B. To evaluate both on fair ground, we further enhance Seq2Seq to copy tokens from the source version of the code in case of an `<unknown>` token being generated by the model. In this way, both CODIT and Seq2Seq handle the vocabulary explosion problem in same way and can be compared fairly. To train our baseline, we use the same dataset as we used to train CODIT in each respective scenario.

VII. RESULTS

RQ1. How accurately can CODIT suggest concrete edits?

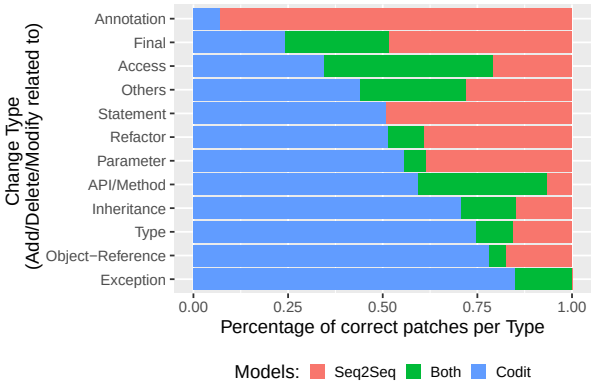
To answer this RQ, we evaluate CODIT's accuracy *w.r.t.* the evaluation dataset containing concrete patches. Table II shows the results: for *Code-Change-Data* dataset, CODIT can successfully generate 423 (6.19%), 788 (11.51%), 860 (12.59%), and 1080 (15.81%) patches at top 1, 2, 5, and 10 respectively. In contrast, the baseline Seq2Seq model (see Section VI), suggests 5.31%, 6.00%, 7.14%, and 7.97% correct patches. This shows CODIT outperforms Seq2Seq by 16.5%, 92.20%, 76.23%, and 98.16% in suggesting concrete patches at top 1, 2, 5, and 10 respectively. Similar results are also observed for *Pull-Request-Data* (Table II).



(a) ***Code-Change-Data*** (b) ***Pull-Request-Data***

Fig. 2: **Patch size (Token Edit-distance) histogram of correctly generated patches in different datasets.**

In general, both models perform better when generating small patches. For example, a large majority of the correctly generated patches have size of one (*i.e.* $\Delta_c = 1$, the token distance between c_p and c_n [44]). However, a non-trivial number of larger patches are also correctly generated. Figure 2 shows the histogram of the size of correctly predicted patches. For example, there are 292 and 50 correct patches with $\Delta_c > 4$ generated by CODIT for *Code-Change-Data* and *Pull-Request-Data* respectively. Among all the correctly generated patches by only CODIT from *Code-Change-Data*, 40, 57, and 604 patches only contain addition, deletion, or both addition and deletion respectively.



Change Type	**Description** (See Table III for examples)
API (354, 204, 38)	Modify API and Method Call (Example-1)
Type (97, 13, 20)	Change of Types (Example-2)
Parameter (48, 5, 33)	Add/delete/modify method parameters (Example-3)
Refactor (37, 7, 28)	Bracketing, variable/method renaming, etc. (Example-4)
Object-Reference (36, 2, 8)	Changes in Object or Class Reference (Example-5)
Statement (28, 0, 27)	Addition, Deletion, Change of statements. (Example-6)
Inheritance (24, 5, 5)	Add/delete abstract keyword, super() call, @Override (Example-7)
Exception (23, 4, 0)	Modify related to Catch, Throw, and other exceptions (Example-8)
Other (22, 14, 14)	Conditions, Values, Data-dependency, etc. (Example-9)
Final (21, 24, 42)	Add/delete final keyword (Example-10)
Access (10, 12, 6)	Modify Access Type (*e.g.,* public, private, protected) (Example-11)
Annotation (1, 0, 13)	Add/delete Nullable

Fig. 3: **Percentage of correct patches per type by CODIT, Seq2Seq, and both the models at top-10. The table describes each type of change. Left column shows the patch category and number of correctly generated patches in the form (CODIT-only, both, Seq2Seq-only).**

To further understand the kind of patches CODIT generates, we manually inspect the successfully generated patches at top 10 by each model from *Code-Change-Data*. We manually categorize these patches into 50 categories. Two authors

TABLE III: **Examples of different types of concrete patches generated by CODIT**

Example 1. API Change

```
return f.createJsonParser createParser(...)
```

Example 2. Type Change

```
void appendTo(StringBuffer StringBuilder buffer)
```

Example 3. Parameter: Add/Delete Method Parameter

```
1. testDataPath(false , true , true , true, false );
2. assertNotificationEnqueued(map ,key ,value ,hash)
```

Example 4. Refactoring: Modify Method Parameters Name

```
void visit(JSession x session , ...) throws Exception
{
    visit (((JNode) (x session)), ...);
}
```

Example 5. Object-Reference Change

```
setHeader(...,HandlerHelper HandlerUtils.construct())
```

Example 6. Statement: Add Statement

```
{...
    interruptenator.shutdown();
    Thread.interrupted();
}
```

Example 7. Inheritance: Abstracting a Method

```
public abstract void removeSessionCookies (...)
 {
     throw new android...MustOverrideException();
 }
```

Example 8. Exception Change: Add Try Block

```
 public void copyFrom( java.lang.Object arr){
+      try{
         android.os.Trace.traceBegin (...);
         ...
+      finally{
         android.os.Trace.traceEnd(...);
+      }
 }
```

Example 9. Other: Delete Unreferenced Variable

```
 public void testConstructor2NPE(){
 ...
 AtomicIntegerArray aa = new AtomicIntegerArray(a);
 shouldThrow () ;
 ...
 }
```

Example 10. Final: Make Method Parameter Immutable

```
public String print(final ReadableInstant inst){...}
```

Example 11. Access: Modify Access Type

```
 public protected AbstractInstant () { super(); }
```

Every cell shows an example of correctly suggested patches by CODIT. Top line is the patch category, followed by the actual patch. In the patch, **Red** tokens/lines are deleted and **Green** tokens/lines are added.

individually categorized them and then resolved any conflicts, by discussion. These 50 categories are then further combined into 13 coarse-grained patch types, as described in Figure 3 which gives detailed insights into the kind of patches CODIT generates, and how it compares with the baseline Seq2Seq.

We see that CODIT performs significantly better than Seq2Seq in *Exception Handling* (exact match of 27 *v.s.* 4). Most of these patches contain either structural changes (*e.g.,* addition to try-catch as shown in Example 8 in Table III) or changes in exception types. Here, structural/type information from $\mathscr{M}_{tree}$ help CODIT. Table IV shows some additional examples in this category: different exception/error types (*i.e.* `Exception`, `Error`, `RuntimeException`) are changed to `EOFException` although their usage differs. In the first three examples `EOFException` is used as a class reference, while for the others `EOFException` is used to initialize an object. These examples also illustrate CODIT's ability to generalize to different contexts and use-cases.

TABLE IV: **Examples of CODIT's ability to generalize in different use cases.** `Exception`, `Error`, `RuntimeException` **are modified to** `EOFException` **under different context.**

```
Ex:1. return Error EOFException.class ;
Ex:2. return Exception EOFException.class ;
Ex:3. return RuntimeException EOFException.class ;
Ex:4. return new Error EOFException(msg) ;
Ex:5. return new Exception EOFException(msg);
Ex:6. return new RuntimeException EOFException(msg);
```

CODIT also performs very well compared to Seq2Seq in suggesting *API related* changes—558 *v.s.* 242 correct patches at top 10. The main reason behind CODIT's success is its additional knowledge about the structure of the patch and the type of each predicted token. The type information significantly reduces the search space of tokens within each type. In contrast, Seq2Seq predicts API names from the probability distribution over all possible tokens, which is a much larger set of possibilities. Thus, as shown in Example 1 in Table III, CODIT correctly modifies the API name `createJsonParser` to `createParser`. Such additional type information gives CODIT advantages for *Type Changes* (110 *v.s.* 33 for CODIT *v.s.* Seq2Seq) and *Object Reference Changes* (38 *v.s.* 10). See example 2 and 5 in Table III respectively.

Both CODIT and Seq2Seq perform reasonably well when adding or deleting *parameters to method calls* (Example 3 in Table III), *refactorings* (Example 4), and *addition/deletion of statements* (Example 6). For refactoring, although both models have comparable performance, Seq2Seq mostly generated changes related to adding/deleting brackets, while CODIT produced a lot of variable/method renamings. Consider Example 4 where `x` is renamed to `session` both the formal parameter and the usage in the body. Note that, since CODIT uses a tree-based model it is good at capturing long-distance dependencies allowing the token-level model to focus on predicting tokens, *e.g.,* such that it can rename the same variable similarly.

For suggesting *Inheritance* related changes, most of the successful patches generated by Seq2Seq are inserting the `@Override` annotation. Instead, CODIT can additionally predict some non-trivial Inheritance-related patches. For example, in Example 7, CODIT does not only add the **`abstract`** keyword in the method signature, but also removes the body. Since CODIT is aware of code syntax, it learns that method declarations with an **`abstract`** keyword have a high probability of an empty method body.

For the *Annotation* and *Final* categories Seq2Seq outperforms CODIT. Since these changes are much simpler change patterns, we speculate that the added complexity of the tree model introduces additional uncertainties to the prediction. For example, in the case of annotations, the correct trees are not

predicted in top K_{tree} positions by $\mathscr{M}_{tree}$ and hence, CODIT failed to predict correct patches.

> **Result 1:** CODIT *suggests 12.59% correct patches for Code-Change-Data and 12.51% for Pull-Request-Data within top 5 and outperforms Seq2Seq by 76.23% and 89.70% respectively.*

Next, we evaluate CODIT's sub-components.

RQ2. How do different design choices affect CODIT's performance?

We investigate this RQ in three parts: evaluate standalone (i) the token generation model ($\mathscr{M}_{token}$), (ii) the tree translation model ($\mathscr{M}_{tree}$), and (iii) evaluate the combined model, *i.e.* CODIT, under different design choices.

Evaluating token generation model. Here we compare $\mathscr{M}_{token}$ with the baseline Seq2Seq model. Note that $\mathscr{M}_{token}$ generates a token given its structure. Thus, for evaluating the standalone token generation model in CODIT's framework, we assume that the true structure is given (emulating a scenario where a developer knows the kind of structural change they want to apply). Table V presents the results.

TABLE V: **Correct patches generated by the standalone token generation model when the true tree structure is known.**

Dataset	Total Correct Patches	
	Seq2Seq	standalone $\mathscr{M}_{token}$
Code-Change-Data	545 (7.97%)	1509 (22.09%)
Pull-Request-Data	73 (6.89%)	250 (23.63%)

While the baseline end-to-end Seq2Seq generates 7.97% (545 out of 6831) and 6.89% (73 out of 1058) correct patches for *Code-Change-Data* and *Pull-Request-Data* respectively at top 10, Table V shows that if the change structure (*i.e.* t_n) is known, the standalone $\mathscr{M}_{token}$ model of CODIT can generate 22.09% (1509 out of 6832) and 23.63% (250 out of 1058) for *Code-Change-Data* and *Pull-Request-Data* respectively, *i.e.* 177% and 243% performance gain. This result empirically shows that if the tree structure is known, NMT-based code change prediction significantly improves. In fact, this observation led us build CODIT as a two-stage model.

TABLE VI: $\mathscr{M}_{tree}$ **top-10 performance for different settings.**

Dataset	#Patches Correctly Produced*	
	Full Dataset	Filtered Dataset
Code-Change-Data	4204 / 6832 (61.54%)	1271 / 3782 (33.61%)
Pull-Request-Data	408 / 1058 (38.56%)	256 / 833 (30.73%)

* Each cell represents correctly predicted patches / total patches (percentage of correct patch) in the corresponding setting.

Evaluating tree translation model. Here we evaluate how accurately $\mathscr{M}_{token}$ predicts the structure of a change — shown in Table VI. $\mathscr{M}_{tree}$ can predict 61.54% and 38.56% of the structural changes in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, the outputs that are generated by $\mathscr{M}_{tree}$ are not concrete code, rather they a structural abstraction. Recently, Tufano *et al.* [24] employed a different form of abstraction: using a heuristic, they replace most of the identifiers including variables, methods and types with abstract names and transform previous and new code fragments to abstracted code templates. They reported 36% accuracy for the abstract template in the top 10 suggestions. Since, our abstraction mechanism differs significantly, directly comparing these numbers does *not* yield a fair comparison.

TABLE VII: **CODIT performance *w.r.t.* to the attention+copy mechanism @top-10 (K_{tree} =1, K_{token} =10).** Lower bound is without attention+copy. The upper bound evaluates with oracle copying predictions for `<unknown>`. For CODIT each `<unknown>` token is replaced by the source token with the highest attention.

Dataset	lower bound	upper bound	CODIT
Code-Change-Data	727 (10.64%)	1309 (19.16%)	1075 (15.73%)
Pull-Request-Data	81 (7.66%)	160 (15.12%)	138 (13.04%)

Note that, not all patches contain structural changes (*e.g.,* when a single token, such as a method name, is changed). For example, 3050 test patches of *Code-Change-Data*, and 225 test patches of *Pull-Request-Data* do not have structural changes. When we use these patches to train $\mathscr{M}_{tree}$, we essentially train the model to sometimes copy the input to the output and rewarding the loss function for predicting no transformation. Thus, to report the capability of $\mathscr{M}_{token}$ to predict structural changes, we also train a separate version of $\mathscr{M}_{tree}$ using only the training patches with at least 1 node differing between t_n and t_p. We also remove examples with no structural changes from the test set. This is our filtered dataset (Table VI). In the filtered dataset, $\mathscr{M}_{tree}$ predicts 33.61% and 30.73% edited structures from *Code-Change-Data* and *Pull-Request-Data* respectively. This gives us an estimate of how well $\mathscr{M}_{token}$ can predict *structural* changes.

Evaluating CODIT. Having $\mathscr{M}_{tree}$ and $\mathscr{M}_{token}$ evaluated separately, we will now evaluate our end-to-end combined model ($\mathscr{M}_{tree}$ + $\mathscr{M}_{token}$) focusing on two aspects: (i) effect of attention-based copy mechanism, (ii) effect of beam size.

First, we evaluate CODIT's attention-based copy mechanism to handle out-of-vocabulary words, as described in Section IV-B. Table VII shows the results.

TABLE VIII: **Performance of CODIT @top-10 for different $\mathbf{K_{tree}}$.** We take top 2 trees generated by $\mathscr{M}_{tree}$, expanding each into 10 different token sequences, *i.e.* 20 probable c_n. CODIT then chooses the top 10 prospective patches. This yields the best performing model.

Dataset	$K_{token} = 10$			
	$K_{tree} = 1$	$K_{tree} = 2$	$K_{tree} = 5$	$K_{tree} = 10$
Code-Change-Data	1075 15.73%	1080 15.81%	398 5.83%	518 7.58%
Pull-Request-Data	138 13.04%	152 14.37%	105 9.92%	126 11.91%

Recall that $\mathscr{M}_{token}$ generates a probability distribution over the vocabulary (Section IV-B). Since the vocabulary is generated using the training data, any unseen tokens in the test patches are replaced by a special `<unknown>` token. In our experiment, we found that a significant number (about 9% is *Code-Change-Data* and about 8% is *Pull-Request-Data*) of patches contain `<unknown>` tokens; this is undesirable since the generated code will not compile. Without attention+copy mechanism, CODIT can predict 727 (10.64%), and 81 (7.66%) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. However, if all the `<unknown>` tokens could be

replaced perfectly with the intended token, *i.e.* upper bound of the number of correct patches goes up to 1309 (19.16%) and 160 (15.12%) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. This shows the need for tackling the <unknown> token problem. To solve this, we replace <unknown> tokens predicted by $\mathscr{M}_{token}$ with the source token with the highest attention probability following Section IV-B. With this, CODIT generates 1075 (15.73%), and 138 (13.04%) correct patches from *Code-Change-Data* and *Pull-Request-Data* respectively (Table VII).

Second, we test two configurations of beam size, K_{tree} and K_{token} *i.e.* the number of trees generated by $\mathscr{M}_{tree}$ and number of concrete token sequences generated by $\mathscr{M}_{token}$ per tree (Section V). Table VIII shows results for different values of K_{tree} and K_{token}. For top-10 accuracy (*i.e.* $K = 10$, a user defined parameter), we get the best performing result using K_{tree} =2, K_{token} =10 on both datasets.

> **Result 2:** CODIT *yields the best performance with a copy-based attention mechanism and with tree beam size of 2.* $\mathscr{M}_{tree}$ *achieves 61.5% and 38.6% accuracy and* $\mathscr{M}_{token}$ *achieves 22.1% and 23.6% accuracy in Code-Change-Data and Pull-Request-Data respectively when tested individually.*

Finally, we evaluate CODIT's ability to fixing bugs.

RQ3. How accurately CODIT suggests bug-fix patches?

We evaluate this RQ with the state-of-the-art bug-repair dataset, Defects4J [42] using all six projects.

Training: We collect commits from the projects' original GitHub repositories and preprocess them as described in Section V. We further remove the Defects4J bug fix patches and use the rest of the patches to train and validate CODIT.

Testing: We extract the methods corresponding to the bug location(s) from the buggy-versions of Defects4J. A bug can have fixes across multiple methods. We consider each method as candidates for testing and extract their ASTs. We then filter out the methods that are not within our accepted tree sizes. In this way, we get 117 buggy method ASTs corresponding to 80 bugs. The rest of the bugs are ignored.

Here we assume that a fault-localization technique already localizes the bug [45]. In general, fault-localization is an integral part of program repair. However, in this paper, we focus on evaluating CODIT's ability to produce patches rather than an end-to-end repair tool. Since fault localization and fixing are methodologically independent, we assume that bug location is given and evaluate whether CODIT can produce the correct patch. Evaluation of CODIT's promise as a full-fledged bug repair tool remains for future work.

For a buggy method, we extract c_p. Then for a given c_p, we run CODIT and generate a ranked list of generated code fragments (c_n). We then try to patch the buggy code with the generated fragments following the rank order, until the bug-triggering test passes. If the test case passes, we mark it a potential patch and recommend it to developers. We set a specific time budget for the patch generation and testing. For qualitative evaluation, we additionally investigate manually the patches that pass the triggering test cases to evaluate the semantic equivalence with the developer-provided patches. Here we set the maximum time budget for each buggy method to 1 hour. We believe this is a reasonable threshold as previous repair tools (*e.g.,* Elixir [46]) set 90 minutes for generating patches. SimFix [47] set 5 hours as their time out for generating patches and running test cases.

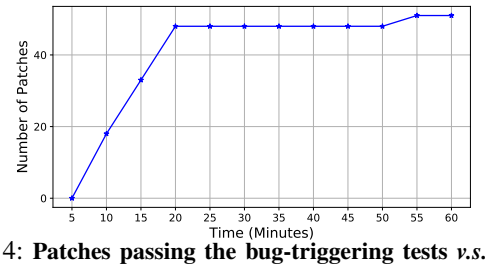


Fig. 4: **Patches passing the bug-triggering tests *v.s.* time.**

TABLE IX: **CODIT's performance on fixing Defects4J [42] bugs.**

Project	BugId	#methods to be patched	#methods in scope	#methods CODIT can fix	Patch Type
Chart	8	1	1	1	Api Change
	10	1	1	1	Method-Invocation
	11	1	1	1	Variable-Name-Change
	12	1	1	1	Api-Change
Closure	3†	2†	1†	1†	Method-Invocation†
	75†	2†	1†	1†	Return-Value-Change†
	86	1	1	1	Boolean-Value-Change
	92	1	1	1	Api-Change†
	93	1	1	1	Api-Change
Lang	4†	2†	1†	1†	Method-Invocation†
	6	1	1	1	Method-Parameter-Change
	21	1	1	1	Method-Parameter-Change
	26	1	1	1	Method-Parameter-Add
	30†	5†	1†	1†	Type-Change†
Math	6†	13†	1†	1†	Method-Parameter-Change†
	30	1	1	1	Type-Change
	46*	2*	2*	1*	Ternary-Statement-Change*
	49	4	4	4	Object-Reference-Change
	57	1	1	1	Type-Change
	59	1	1	1	Ternary-Statement-Change
	70	1	1	1	Method-Parameter-Add
	98	2	2	2	Array-Size-Change
Mockito	6*	20*	20*	2*	Api-Change*
	25†	6†	1†	1†	Method-Parameter-Add†
	30†	2†	1†	1†	Method-Parameter-Add†

Green rows are bug ids where CODIT can produce complete patch. Blue† rows are where CODIT can fix all the methods that are in CODIT's scope. Orange* rows are where CODIT could not fix all the buggy methods in CODIT's scope.

CODIT can successfully generate at least 1 patch that passes the triggering test case for 51 methods out of 117 buggy methods from 30 bugs, *i.e.* 43.59% buggy methods are potentially fixed. Figure 4 shows the number of patches passing the bug-triggering test case *w.r.t.* time. We see that, 48 out of 51 successful patches are generated within 20 minutes.

We further manually compare the patches with the developer-provided patches: among 51 potential patches, 30 patches are identical and come from 25 different bug ids (See Table IX). The bugs marked in green are completely fixed by CODIT with all their buggy methods being successfully fixed. For example, Math-49 has 4 buggy methods, CODIT fixes all four. For the bugs marked in blue†, CODIT fixes all

the methods that are in scope. For example, for Lang-4, there are 2 methods to be fixed, 1 of them are in CODIT's scope, and CODIT fixes that. However, for two other bugs (marked in orange*), CODIT produces only a partial fix. For example, in the case of Math-46 and Mockito-6, although all the methods are within scope, CODIT could fix 1 out of 2 and 2 out of 20 methods respectively. The 'Patch Type' column further shows the type of change patterns.

We further compare CODIT with baseline Seq2Seq model (Section VI-B). Seq2Seq model generates 5 complete and 2 partial patches. Among those 7 patches, CODIT generates 6. For Chart-1, Seq2Seq generates the fix, but CODIT could not. CODIT failed to generate this patch because, the correct tree that corresponds to the fixed code was not generated and tested by CODIT within the time limit we set.

One prominent bug repair approach [46], [48], [49] is to transform a suspicious program element following some change patterns until a patch that passes the test cases is found. For instance, Elixir [46] used 8 predefined code transformation patterns and applied those. In fact, CODIT can generate fixes for 8 bugs out of 26 bugs that are fixed by Elixir [46]. Nevertheless, CODIT can be viewed as a transformation schema which automatically learns these patterns without human guidance. We note that CODIT is *not* explicitly focused on bug-fix changes since it is trained with generic changes. Even then, CODIT achieves good performance in Defects4J bugs. Thus, we believe CODIT has the potential to complement existing program repair tools by customizing the training with previous bug-fix patches and allowing to learn from larger change sizes. Given time and space limitations, we leave this to future work.

Result 3: CODIT *generates complete bug-fix patches for 16 bugs and partial patches for 9 bugs in Defects4J.*

VIII. THREATS TO VALIDITY

External Validity. We built and trained CODIT on real-world changes. Like all machine learning models, our hypothesis is that the dataset is representative of real code changes. To mitigate this threat, we collected patch data from different repositories and different types of edits collected from real world.

Internal Validity. Similar to other ML techniques, CODIT's performance depends on hyperparameters. To minimize this threat, we tune the model with a validation set. To check for any unintended implementation bug, we frequently probed our model during experiments and tested for desirable qualities. In our evaluation, we used exact similarity as an evaluation metric. However, a semantically equivalent code may be syntactically different, *e.g.,* refactored code. We will miss such semantically equivalent patches. Thus, we give a lower bound for CODIT's performance.

IX. RELATED WORK

Modeling source code. Applying ML to source code has received increasing attention in recent years [22] across many applications such as code completion [11], [29], bug prediction [50]–[52], clone detection [53], code search [54], *etc*. In these work, code was represented in many form, *e.g.,* token sequences [26], [29], parse-trees [55], [56], graphs [52], [57], embedded distributed vector space [58], *etc*. In contrast, we aim to model code changes, a problem fundamentally different from modeling code. Yin *et al.* [59] proposed graph neural network-based distributed representation for code edits but their work focused on change representation than generation.

Machine Translation (MT) for source code. MT is used to translate source code from one programming language into another [60]–[63]. These works primarily used Seq2Seq model at different code abstractions. In contrast, we propose a syntactic, tree-based model. More closely to our work, Tufano *et al.* [23], [24], and Chen *et al.* [41] showed promising results using a Seq2Seq model with attention and copy mechanism. Our baseline Seq2Seq model is very similar to these models. However, Tufano *et al.* [23] experimented on transformed code templates where identifiers are replaced by abstract symbolic tokens. This vaguely resembles CODIT's $\mathscr{M}_{tree}$ that predicts syntax-based templates, but differs substantially as we discussed in the evaluation section. Gupta *et al.* used Seq2Seq models to fix C syntactic errors in student assignments [64]. However, their approach can fix syntactic errors for 50% of the input codes *i.e.* for rest of the 50% generated patches were syntactically incorrect which is never the case for CODIT because of we employ a tree-based approach.

Program Fixing. Automatic program repair is a well-researched field [49], [65]–[68]. There are two main directions: generate and validate [49], [69], [70], and synthesis-based strategies [71], [72] through learning. Researchers have applied conceptually similar strategies by searching for fixes from existing code bases [69], [73]. Le *et al.* [74] utilized the development history as an effective guide in program fixing. The key difference between Le *et al.* and this work is that instead of mining discrete change patterns, we devise a probabilistic model that learns and generalizes these patterns from data.

Automatic Code Changes. Modern IDEs [7], [8] provide support for automatic editings, *e.g.,* refactoring, boilerplate templates (*e.g.,* try-catch block) *etc*. There are many research on automatic and semi-automatic [75], [76] code changes as well: *e.g.,* given that similar edits are often applied to similar code contexts, Meng *et al.* [19], [21] propose to generate repetitive edits using code clones, sub-graph isomorphisms, and dependency analysis. Other approaches mine past changes from software repositories and suggest edits that were applied previously to similar contexts [2], [3]. In contrast, CODIT generates edits by learning them from the wild—it neither requires similar edit examples nor edit contexts. Romil *et al.* [20] propose a program synthesis-based approach to generate edits where the original and modified code fragments are the input and outputs to the synthesizer. Such patch synthesizers can be thought of as a special kind of model that takes into account additional specifications such as input-output examples or formal constraints. In contrast, CODIT is a statistical model

that predicts a piece of code given only historical changes and does not require additional input from the developer. Finally, there are domain-specific approaches, such as error handling code generation [77], [78], API-related changes [9]–[14], [27], automatic refactorings [15]–[18], *etc*. Unlike these work, CODIT focuses on general code changes.

X. CONCLUSION

We proposed and evaluated CODIT, a tree based model for suggesting eminent source code changes. CODIT's objective is to suggest changes that are similar to change patterns observed in the wild. We evaluate our work against 6831 real-world patches. The results indicate that treebased models are a promising tool for generating code patches They outperform popular seq2seq alternatives indicating that tree-based modeling is a better approach for modeling code changes. CODIT further shows promise in repairing bug fixes. We will explore CODIT's potential in fixing bugs in the future.

REFERENCES

[1] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 315–324, ACM, 2010.

[2] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 180–190, IEEE Press, 2013.

[3] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," MSR '15, ACM, 2015.

[4] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 492–495, ACM, 2014.

[5] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–317, ACM, 2014.

[6] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 367–377, IEEE, 2013.

[7] Microsoft, "Visual Studio (https://visualstudio.microsoft.com)."

[8] E. Foundation, "Eclipse IDE (https://www.eclipse.org)."

[9] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *ACM Sigplan Notices*, vol. 45, pp. 302–321, ACM, 2010.

[10] W. Tansey and E. Tilevich, "Annotation refactoring: inferring upgrade transformations for legacy applications," in *ACM Sigplan Notices*, vol. 43, pp. 295–312, ACM, 2008.

[11] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, pp. 419–428, ACM, 2014.

[12] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 382–385, IEEE, 2012.

[13] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Acm sigops operating systems review*, vol. 42, pp. 247–260, ACM, 2008.

[14] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 511–522, ACM, 2016.

[15] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 222–232, IEEE, 2012.

[16] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev, "Refactoring with synthesis," in *ACM SIGPLAN Notices*, vol. 48, pp. 339–354, ACM, 2013.

[17] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 211–221, IEEE Press, 2012.

[18] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 392–402, IEEE Press, 2015.

[19] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[20] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*, pp. 404–415, IEEE Press, 2017.

[21] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 502–511, IEEE Press, 2013.

[22] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, 2018.

[23] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," 2018.

[24] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," *arXiv preprint arXiv:1901.09102*, 2019.

[25] P. Flener, *Logic program synthesis from incomplete information*, vol. 295. Springer Science & Business Media, 2012.

[26] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *SIGSOFT FSE*, pp. 269–280, 2014.

[27] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," *arXiv preprint arXiv:1703.05698*, 2017.

[28] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[29] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847, IEEE, 2012.

[30] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, ACM, 2017.

[31] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang, "Building language models for text with named entities," 2018.

[32] J. Mallinson, R. Sennrich, and M. Lapata, "Paraphrasing revisited with neural machine translation," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, vol. 1, pp. 881–893, 2017.

[33] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[34] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, pp. 2091–2100, 2016.

[35] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[36] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[37] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[38] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-Source Toolkit for Neural Machine Translation," *ArXiv e-prints*.

[39] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 313–324, ACM, 2014.

[40] D. R. Reddy *et al.*, "Speech understanding systems: A summary of results of the five-year research effort," *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA*, 1977.

[41] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *arXiv preprint arXiv:1901.01808*, 2018.

[42] 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*, pp. 437–440, ACM, 2014.

[43] "Gerrit Code Review Database." https://www.gerritcodereview.com/, 2018. [Online; accessed 18-August-2018].

[44] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[45] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98, IEEE, 2007.

[46] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pp. 648–659, IEEE, 2017.

[47] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 298–309, ACM, 2018.

[48] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 24–36, ACM, 2015.

[49] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 802–811, IEEE Press, 2013.

[50] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 428–439, ACM, 2016.

[51] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 708–719, ACM, 2016.

[52] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[53] 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*, pp. 87–98, ACM, 2016.

[54] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 933–944, ACM, 2018.

[55] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 440–450, 2017.

[56] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, pp. 649–657, 2014.

[57] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 858–868, IEEE, 2015.

[58] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[59] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *arXiv preprint arXiv:1810.13337*, 2018.

[60] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 173–184, ACM, 2014.

[61] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 585–596, IEEE, 2015.

[62] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 457–468, ACM, 2014.

[63] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[64] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning.," in *AAAI*, pp. 1345–1351, 2017.

[65] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 266–276, ACM, 2014.

[66] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 3–13, IEEE, 2012.

[67] P. Liu, O. Tripp, and X. Zhang, "Flint: fixing linearizability violations," in *ACM SIGPLAN Notices*, vol. 49, pp. 543–560, ACM, 2014.

[68] F. Logozzo and T. Ball, "Modular and verified automatic program repair," in *ACM SIGPLAN Notices*, vol. 47, pp. 133–146, ACM, 2012.

[69] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, p. 54, 2012.

[70] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 166–178, ACM, 2015.

[71] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 772–781, IEEE, 2013.

[72] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, pp. 691–701, ACM, 2016.

[73] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pp. 162–168, IEEE, 2008.

[74] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven automated program repair," 2016.

[75] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 567–576, ACM, 2007.

[76] R. Robbes and M. Lanza, "Example-based program transformation," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 174–188, Springer, 2008.

[77] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 752–762, ACM, 2017.

[78] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1019–1028, ACM, 2010.