Learning the Relation between Code Features and Code Transforms with Structured Prediction

Zhongxing Yu, Matias Martinez, Zimin Chen, Tegawendé F. Bissyandé and Martin Monperrus

Abstract—To effectively guide the exploration of the code transform space for automated code evolution techniques, we present in this paper the first approach for structurally predicting code transforms at the level of AST nodes using conditional random fields (CRFs). Our approach first learns offline a probabilistic model that captures how certain code transforms are applied to certain AST nodes, and then uses the learned model to predict transforms for arbitrary new, unseen code snippets. Our approach involves a novel representation of both programs and code transforms. Specifically, we introduce the formal framework for defining the so-called AST-level code transforms and we demonstrate how the CRF model can be accordingly designed, learned, and used for prediction. We instantiate our approach in the context of repair transform prediction for Java programs. Our instantiation contains a set of carefully designed code features, deals with the training data imbalance issue, and comprises transform constraints that are specific to code. We conduct a large-scale experimental evaluation based on a dataset of bug fixing commits from real-world Java projects. The results show that when the popular evaluation metric *top-3* is used, our approach predicts the code transforms with an accuracy varying from 41% to 53% depending on the transforms. Our model outperforms two baselines based on history probability and neural machine translation (NMT), suggesting the importance of considering code structure in achieving good prediction accuracy. In addition, a proof-of-concept synthesizer is implemented to concretize some repair transforms to get the final patches. The evaluation of the synthesizer on the Defects4j benchmark confirms the usefulness of the predicted AST-level repair transforms in producing high-quality patches.

Index Terms—code transform, big code, machine learning, program repair.

1 Introduction

During the life-cycle of a computer program, its source code evolves under a sequence of transforms. Those code transforms are not random: they capture the evolution of the program (e.g., new features and bug fixes) and they also reflect the semantic constraints of the programming language (each version must compile, so not all transforms are acceptable). In other words, there is a probability distribution over the code transform space. In this paper, we address the problem of capturing the probability distribution of code transforms that underlies software evolution.

Code transform prediction is an important problem. It has implications in research areas dealing with automated code evolution, including, for example, program synthesis [1], program repair [2], super-optimization [3] and refactoring [4], and can be viewed as the foundation to achieve effective search. For instance, in program repair, the probability distribution of code transforms can not only enable a more focused exploration of the search space, but can also result in better patches [2].

The problem of computing the probability distribution of code transforms given a program is an unsolved problem. Yet, it is being indirectly investigated in the specific application domain of automated program repair [5]–[7]. For instance, the Prophet repair system [2] analyses a set of past commits extracted from version control systems in order to compute the likelihood of a given patch. Indeed, the fundamental issue behind automated code evolution is a representation problem: we need to identify the proper representation for both the program and code transforms. If the representation is too fine-grain (e.g., at the token level), one would need a tremendous amount of data or memory to capture the probability distribution. If the representation is too coarse-grain (e.g., at the statement level), the relation between code and code transforms becomes vague and unactionable, making it useless for driving automated code evolution.

In this paper, we propose an approach to learn the relation between code and code transforms. This approach is novel and effective. Its novelty lies in the representation of both programs and code transforms.

Representing programs. Programs are represented with a combination of abstract syntax trees (AST) and rich carefully engineered and powerful features. This representation has two advantages: (1) the learning algorithm has access to the full program information (all tokens), as well as to the AST tree structure and the AST node types; (2) the learning algorithm does not have to extract the probability from scratch, it can leverage the human knowledge encoded in the features to better and faster identify the signal from the noise in the learning data.

Representing code transforms. To make the code transforms to be precise enough to be automatically applied, we define AST-level code transforms, i.e., code transforms that are attached to specific AST nodes of the code. The AST-level code transforms are defined as follows. An 'edit' is a basic tree edit operation performed on nodes of the abstract syntax tree of the program before change. A 'diff' is the com-

- *Z. Yu is with Shandong University. E-mail: zhongxing.yu@sdu.edu.cn*
- *M. Martinez is with Universitat Politècnica de Catalunya. E-mail: matias.martinez@upc.edu*
- *Z. Chen and M. Monperrus are with KTH Royal Institute of Technology. E-mail: zimin@kth.se and monperrus@kth.se*
- *T. F. Bissyandé is with University of Luxembourg. E-mail: tegawende.bissyande@uni.lu*

 http://dx.doi.org/10.1109/TSE.2023.3275380

plete set of edits done in an atomic code change, as captured by a commit in a version control system. The AST-level code transform is an abstract view over a single edit or a group of several conceptually related edits. Our intuition is that this abstraction, compared with concrete AST edits, brings more accuracy and scalability for the arguably hard learning task of predicting code changes. We give the conceptual formal framework for defining this novel type of code transform.

Learning algorithm. The learning machinery is provided by structured prediction [8], in particular conditional random fields (CRFs) [9]. Structured prediction is a branch of machine learning that is well suitable for tree-based data such as abstract syntax tress. CRFs recently have been successfully used on programs, including, for example, automatic deobfuscation [10] and automatic renaming [11]. While there exist neutral network based approaches (*e.g.*, graph neural network [12]) that can also be used for structured prediction, CRF model has the following advantages: (1) CRF model is fully explainable and the chain of reasoning can be looked at through the associated graph; (2) CRF model exploits domain knowledge encoded into feature functions, thus it enables to capture human insight about independence, causality and other subtle relationships in the underlying graph structure; (3) CRF model has significantly fewer parameters which then can be estimated reliably from less data. In our case, the learned CRF model establishes a probabilistic model that captures how certain code transforms are applied to certain AST nodes, and can be used to predict code transforms for arbitrary new, unseen code snippets. While there exist works that build a model about code transforms by treating code as token sequences [13]–[15] or works that extract useful edit information from few highly similar edit instances [16]–[19], our established CRF model accounts the inherent structural dependencies between transforms applied to different code elements and is able to extract and assemble useful information from diverse (i.e., not that similar) example edit instances. The resultant probabilistic model thus is powerful in predicting our newly defined AST-level code transforms.

We instantiate our approach in the context of Java programs and code transforms used for repairing programs (hereafter referred to as repair transform for brevity). Our prototype system takes as input a set of past bug-fixing commits and produces a probabilistic model that can be used to predict the repair transforms needed for repairing a bug. We evaluate our approach on the *September 2015/GitHub* dataset offered by Boa [20], [21] (we hereafter call it Boa dataset for brevity) which contains 4,590,679 bug fixing commits, and measure to what extent our approach correctly predicts the repair transforms to be applied on the program version before the commit. We perform two series of experiments, one on “single-transform” diffs and one on "multiple-transform" diffs, and compare our model performance with that of two baselines which respectively use history probability and neural machine translation (NMT) to predict repair transforms. Intuitively speaking, “single-transform” diffs and "multiple-transform" diffs respectively refer to the case where one and multiple repair transforms are needed to change the buggy code into the correct code. For "single-transform" diffs, when the popular evaluation metric *top-3* is used [15], [22], our overall best performance model achieves 53% accuracy, and the history probability baseline and NMT baseline accuracies are 31% and 47% respectively. To our knowledge, the "multiple-transform" prediction problem is novel and has not been studied so far. It is arguably a harder prediction problem because the prediction space is orders of magnitude large. For "multiple-transform" diffs, when the popular evaluation metric *top-3* is used [15], [22], our best performance model achieves 41% accuracy, and the history probability baseline and NMT baseline accuracies are 0% and 36% respectively. We also systematically investigate the impact of configuration parameter, training data size, and feature functions on model performance. Overall, the results show that our established model achieves good prediction accuracy and consistently performs better than the two baselines. Compared to the baselines, our model takes into account the inherent code structure and thus achieves a better prediction performance.

In addition, to further illustrate the usefulness of the predicted AST-level repair transforms in producing the final patch, we implement a proof-of-concept synthesizer which concretizes 8 of the 16 considered repair transforms to generate patches. We evaluate the synthesizer on the widely used Defects4j (v1.2) benchmark, and compare the repair results with those obtained by 5 state-of-the-art deep learning (DL) based automatic program repair (APR) techniques, including CODIT [23], CoCoNut [24], DLFix [25], CURE [26], and Recoder [27]. The results show that by applying the proof-of-concept synthesizer to concretize the top 1 repair transform prediction by our model, the synthesizer correctly repairs 16 bugs, and a significant number of the repaired bugs are not repaired by the compared DL-based APR techniques. In particular, compared with CODIT, CoCoNut, DLFix, CURE, and Recoder, the proof-of-concept synthesizer repairs 13, 10, 4, 6, and 2 unique bugs respectively. In addition, as the predicted repair transforms are attached to specific AST nodes, the patch synthesis can proceed in a highly focused way. The synthesized patches thus will suffer less from the overfitting issue [28], and our evaluation confirms this. Overall, the evaluation results confirm the usefulness of the predicted AST-level repair transforms in producing high-quality patches.

To sum up, our contributions are:

- A conceptual framework for defining AST-level code transform and a novel approach to predict AST-level code transform based on structured prediction. Both of them are completely novel to the best of our knowledge.
- An instantiation of the approach for repair transform prediction for Java programs, which contains a set of carefully designed code features and deals with the training data imbalance and transform constraint issues that arise for repair transform prediction problems. The prototype implementation is publicly available at https://github.com/zhongxingyu/Seer.git.
- A large-scale and systematic experimental evaluation of the approach on the Boa dataset [20], [21]. The results show that our overall best performance model achieves good accuracy of code transform prediction, and consistently performs better than two baselines based on history probability and neural machine translation (NMT).
- An implementation of a proof-of-concept synthesizer

which concretizes some repair transforms to get the final patches, and an evaluation of the synthesizer on the Defects4j benchmark. The evaluation results demonstrate that high-quality patches can be obtained on top of the predicted AST-level repair transforms.

The remainder of this paper is structured as follows. We first use a working example to illustrate our approach in Section 2. Section 3 gives a necessary background about abstract syntax tree (AST) and conditional random fields (CRFs). Section 4 introduces the approach to structurally predict AST-level code transform using CRFs, followed by Section 5 which gives an instantiation of the approach in the context of repair transform for Java programs. Section 6 presents a detailed evaluation of the approach, including the proof-of-concept synthesizer. Finally, we discuss some closely related work in Section 7 and conclude the paper in Section 8.

2 Overview

In this section, using a working example, we provide an informal overview of our approach for predicting code transforms on AST nodes. Figure 1 gives a graphical overview. It shows the diff of Git commit ecc184b in project Jmist[1]. The problem involves two wrong invocations to method $y()$ that should be replaced by invocations to method $z()$. The code transform behind this diff is a replacement of a method invocation by another one, which is called a "`Meth-RW-Meth`" repair transform in this paper. In the diff of Figure 1, there are two instances of this code transform and they are applied to two different AST nodes.

The table at the top right hand side of Figure 1 shows the predictions of our model. Each prediction contains a set of predicates `Tran`(*T*, *N*), which evaluates to `true` when there exists code transform *T* on AST node *N* (see definition 4.5 for details). The prediction with the highest score 0.6 is composed of two "`Meth-RW-Meth`" code transforms on AST nodes identified with indexes 11 and 13. Since it is the actual repair changes to be made, it means that for this example, our approach successfully predicts the code transforms. Note that the prediction involves the locations to apply the code transforms: "`Meth-RW-Meth`" code transform points to the two AST nodes corresponding to the invocation of $y()$. We now outline how our approach achieves this.

Feature Extraction. Given the buggy code snippet, our approach first parses it to construct an AST and then extracts the following two types of features:

- The first type of feature is based on the characteristics of program elements. For instance, they can be whether the invocation $x()$ has overloaded methods and whether the invocation $y()$ is wrapped with an if-check when called in other statements. These features are engineered, and are related with code idioms, semantics not directly captured by the AST (e.g. method overloaded), and common usage.
- The second type of feature is based on the abstract syntax tree. All AST nodes are represented with special vertices, edges, and triangles that are used for structured prediction. For the specific example in Figure 1, an excerpt of this representation is shown on the bottom right hand side.

Code Transforms. To effectively guide the code evolution process, the code transforms are defined in terms of their changes on the AST structure. The AST nodes of the buggy code snippet are then annotated with labels that indicate the presence of a code transform.

Offline Model Learning. After extracting the features for all samples in a training dataset of patches and annotating AST nodes of the buggy code snippets with code transforms, our approach feeds them to a probabilistic model. More specifically, we learn a conditional random field from the data. The learning process makes use of the two types of features mentioned above, and establishes the relation between the features and the code transforms on the AST nodes. In particular, the CRF model allows us to conveniently make the establishment on top of the extracted two types of features using respectively observation-based and indicator-based feature functions. Put simply, observation-based feature functions facilitate us to establish the correlation between characteristics of program elements and the code transforms on program elements. For example, as overloaded methods are frequently mixed by developers, the correlation between "invocation $y()$ has overloaded methods" and "$y()$ should subject `Meth-RW-Meth` repair transform" should be high. Indicator-based feature functions instead facilitate us to establish the correlation between program structure and the code transforms on program elements through historical information on a large dataset. For instance, in case many instances of the node label edge <invo, var> (like edges ⑪ → ⑫ and ⑬ → ⑭ in Figure 1) in the dataset are associated with repair transform pair <`Meth-RW-Meth`, VOID> where VOID denotes that no repair transform is needed, then the correlation between <invo, var> and <`Meth-RW-Meth`, VOID> should be relatively high. To more accurately establish the correlation, we have designed various type related, usage related, and syntax related code element characteristics (Section 5.2.1) and considered structure-transform relation for AST vertices, edges, and triangles on top of an adequately large, representative dataset (Section 5.2.2).

The model is learned offline once and the learned weights for the corresponding feature functions reflect the strength of the correlation, and then the model can be used to do predictions for arbitrary, unseen buggy code snippets.

Prediction. Finally, using the extracted features for the new, unseen buggy code snippet, the already learned model assigns likely code transforms to AST nodes. Each assignment comes with a score representing the probability of the transform. For the buggy code snippet shown in Figure 1, the top-3 predictions by our trained model are shown in the table at the right hand side. The most likely prediction with the score of 0.6 says there is a need to apply two transforms "`Meth-RW-Meth`" (replace one invocation by another one) to AST nodes with indexes 11 and 13 respectively. This prediction is indeed correct. To repair this bug, we exactly need those two repair transforms suggested by the most likely prediction. With this prediction, we can then employ customized effective synthesis algorithm like [29], [30] to

1. https://github.com/bwkimmel/jmist/commit/ecc184bc08ee08159cdd79045c2ed0c4245ba59c

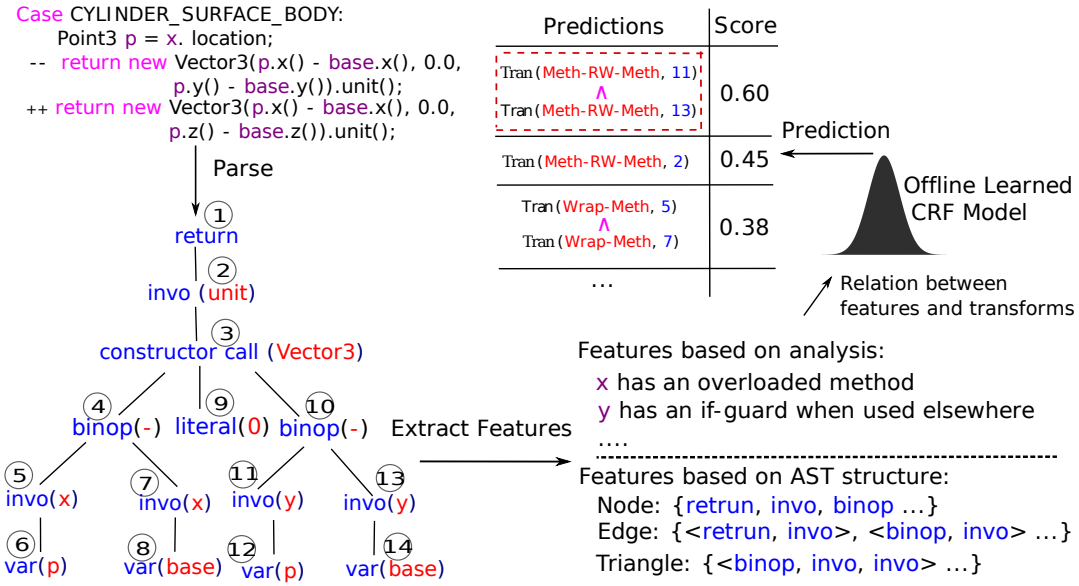

Fig. 1: Overview of our code transform prediction approach.

come up with the actual invocations to replace the two invocations $y()$ at AST nodes 11 and 13.

Key Points. We now emphasize the key points of our approach. First, our model performs structured prediction and hence predicts the code transforms for all AST nodes of the buggy code snippet given as input. We have a special code transform called EMPTY which means no code transform, and we call the other transforms actual code transforms. The EMPTY predictions are not shown in Figure 1, yet they are indeed outputs of the model. Second, our model does prediction of transforms on specific AST nodes, i.e., the prediction is in a targeted manner. For instance, there are three method invocations involved in the buggy code snippet (i.e., $unit()$, $x()$, $y()$), the most likely prediction attaches the repair transforms to the actual buggy invocation (i.e., the two calls to $y()$). Third, our model can effectively deal with the case when there need multiple actual repair transforms to different AST nodes. Those joint transforms are learned at training time and given as outputs at predication time, as shown in Figure 1.

3 Preliminaries

Before describing our approach in detail, we first provide the necessary background. Our prediction is at the level of AST nodes and we start by formally defining the AST.

Definition 3.1. (Abstract Syntax Tree). The abstract syntax tree (AST) for a code snippet is a tuple $< N, T, r, \delta, L, l, V, v >$ where N is a set of nonterminal nodes, T is a set of terminal nodes, $r \in N$ is the root node, $\delta : N \rightarrow (N \cup T)^*$ is a function that maps a nonterminal node to its children nodes, L is a set of node labels, $l : (N \cup T) \rightarrow L$ is a function that maps a node to its label, V is a set of node values, and $v : (N \cup T) \rightarrow (L \cup \epsilon)$ is a function that maps a node to its value (can be empty).

Labels of nodes correspond to the names of their production rules in the grammar, i.e., they encode the structure. Values of the nodes correspond to the actual tokens in the code. For instance, for the AST node identified with (10) in Figure 1, its label and value are "`BinaryOperator`" and "-" respectively.

We use conditional random fields (CRFs) [9] to do the learning. Before describing CRFs in detail, we first give the definition of clique and maximal clique, which are key for understanding CRFs.

Definition 3.2. (Clique and Maximal Clique). For an undirected graph $G = (V, E)$, a clique C is a set X of vertices of G such that every two distinct vertices are adjacent. A maximal clique is a clique that cannot be extended by including one more adjacent vertex.

Imagine an undirected triangle graph, then there are three 1-vertex cliques (the vertices, a special kind of clique), three 2-vertex cliques (the edges), and one 3-vertex clique (the triangle), and the 3-vertex clique is the maximal clique.

We now give the definition of CRFs. *Probabilistic graphical models (PGM) use a graph-based representation as the basis for compactly encoding a complex distribution over a high-dimensional space* [31], and CRFs belong to this formalism for expressing the dependence structure of entities. Traditionally, graphical models have been used to explicitly model the joint probability distribution $p(\mathbf{y}, \mathbf{x})$ over our observed knowledge about the entities (i.e., $\mathbf{x}$) and the predicted assignment of attributes for the entities (i.e., $\mathbf{y}$). This kind of model is called *generative* model. The limitation of the generative model is that it requires modeling the marginal probability $p(\mathbf{x})$, which can be difficult and computationally expensive as the dimensionality of $\mathbf{x}$ can be very large and the features may have complex dependencies. An alternative solution to this problem is a *discriminative* model, which models the conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ directly and is the approach taken by CRFs. A CRF is a conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ with an associated graphical structure, which combines the ability of graphical models to compactly model multivariate outputs $\mathbf{y}$ with that of discriminative

classification to perform prediction using a large number of input features **x**. CRFs have been successfully used in many areas, including, for example, information retrieval [32], natural language processing [33], bioinformatics [34], and computer vision [35]. The formal definition of CRFs is as follows [9].

Definition 3.3. (Conditional Random Fields [9]) Let **X** = $\{X_1, ..., X_N\}$ and **Y** = $\{Y_1, ..., Y_N\}$ be two sets of random variables, **x** and **y** be realizations of **X** and **Y** respectively, G = (V, E) be an undirected graph over **Y** such that **Y** is indexed by the vertices of G, and C be the set of all cliques in G. Then (**X**, **Y**) is a *conditional random field* if for any value **x** of **X** (i.e., conditioned on **X**), the distribution $p(\mathbf{y} \mid \mathbf{x})$ factorizes according to G and is represented as:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{c \in C} \psi_c(\mathbf{y}_c, \mathbf{x}) \tag{1}$$

where $Z(\mathbf{x}) = \sum_{\mathbf{y} \in \Omega_\mathbf{x}} \prod_{c \in C} \psi_c(\mathbf{y}_c, \mathbf{x})$ is a normalization factor. Here $\Omega_\mathbf{x}$ denotes the set of possible assignments of **y** for **x**. The undirected graph encodes the qualitative aspects of the distribution and edges embody direct dependencies. In addition, note that the factorization in CRFs implicitly assumes a node **X** and edges (**X**, Y_i) in the undirected graph.

Each $\psi_c(\mathbf{y}_c, \mathbf{x})$ is a local function that defines the quantitative aspect of the distribution. $\psi_c(\mathbf{y}_c, \mathbf{x})$ depends on the whole **X** but only on a subset $\mathbf{Y}_c \subseteq \mathbf{Y}$ which belong to the clique c, and its non-negative scalar value can be deemed as a measure of how compatible the values $\mathbf{y}_c$ are with each other. Like Hidden Markov Models (HMMs), each local function has the special log-linear form over a prespecified set of feature functions $\{f_k\}_{k=1}^K$:

$$\psi_c(\mathbf{y}_c, \mathbf{x}) = \exp\Big\{ \sum_{k=1}^{K} \lambda_k f_k(\mathbf{y}_c, \mathbf{x}) \Big\} \tag{2}$$

Each feature function f_k: $\mathbf{Y}_c \times \mathbf{X} \rightarrow \mathbb{R}$ is used to score assignments of subset variables $\mathbf{Y}_c$, and λ_k is the model parameter to learn which represents the weight for feature function f_k. Typically, the same set of feature functions with the same parameters is used for every clique in the graph.

Example. Consider the Part-of-Speech (POS) Tagging problem where the goal is to label a sentence with tags like ADJECTIVE, NOUN, etc. For instance, given the sentence "Bob drank coffee at Starbucks", the labeling will be "NOUN VERB NOUN PREPOSITION NOUN". For this problem, the input random variables are the words in a sentence and the output variables are the corresponding tags. The underlying graph could be a liner-chain that connects each Y_i with Y_{i+1}, encoding the fact that the labels to different words are dependent on each other. For liner-chain CRF model, each feature function basically is a function that takes in as input: (1) a sentence **s**; (2) the position i of a certain word in **s**; (3) the tag $\mathbf{t}_i$ of the current word; (4) the tag $\mathbf{t}_{i-1}$ of the previous word. For instance, one possible feature function could be "f_k(**s**, i, $\mathbf{t}_i$, $\mathbf{t}_{i-1}$) = 1 if $\mathbf{t}_{i-1}$ = ADJECTIVE and $\mathbf{t}_i$ = NOUN, 0 otherwise". If the weight λ_k associated with this feature is large and positive, the underlying meaning is that adjectives tend to be followed by nouns. The weight for each feature function could be learned during training and then be employed during prediction. Given a sentence **s**, its tagging **t** can be scored by adding up the weighted features over all words in **s**:

$$score(\mathbf{t}|\mathbf{s}) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(\mathbf{s}, i, \mathbf{t}_i, \mathbf{t}_{i-1})$$

where the first sum iterates over each feature function j, and the inner sum iterates over each position i of the sentence. Considering all possible taggings, the scores can be transformed into probabilities $p(\mathbf{t} \mid \mathbf{s})$ by exponentiating and normalizing:

$$p(\mathbf{t}|\mathbf{s}) = \frac{\exp\Big\{ score(\mathbf{t}|\mathbf{s}) \Big\}}{\sum_{\mathbf{t}'} \exp\Big\{ score(\mathbf{t}'|\mathbf{s}) \Big\}}$$

4 Approach for Structured Prediction of Code Transform

In this section, we introduce our approach for structured prediction of AST-level code transform using conditional random field (CRFs). We first introduce a novel CRF specifically designed for code transform prediction on AST nodes, and then discuss how the approach can be achieved in a step-by-step manner.

4.1 CRF for Transform Prediction on AST Nodes

To effectively guide the code transform process, our aim is to predict the needed transforms at the AST node level. We thus first define the following two random fields.

Definition 4.1. (Random field of AST nodes and transforms). Let P = {1, 2, 3, ..., Q} be the set of AST nodes where the integer is a unique identifier to denote the nodes traversed in pre-order. We associate a random field **N** = $\{N_1, ..., N_Q\}$ over node symbol (a node symbol is a unique combination of node label and node value) in each position p in the AST and another random field **T** = $\{T_1, ..., T_Q\}$ over the code transform applied to each position p in the AST.

The realizations of **N** (denoted by **n**) will be the actual nodes for a specific input AST, and the realizations of **T** (denoted by **t**) will be the actual transforms applied to nodes of the specific input AST. Figure 2 (a) and Figure 2(b) give an example of the realizations of the two random fields where the transforms are applied to repair the bug. To repair the toy bug in Figure 2, we need to replace the binary operator '+' with the binary operator '*' (called "`Binop Replacement`" transform) and wrap expression *3*X* with a method call *Sin* (called "`Wrap-With-Method`" transform). According to definition 4.1, for realizations of random field **T**, these two repair transforms are attached to the AST nodes corresponding to binary operator '+' and expression *3*X* in the AST respectively.

Given the two random fields **N** and **T**, we now discuss the choice of the undirected graph over random variables in **T**, i.e., the choice of CRFs for transform prediction purpose. In CRFs, the more complex the graph, the more kinds of feature functions which relate all variables in a clique can be defined, which in turn would lead to a larger class of conditional probability distributions. However, note meanwhile a

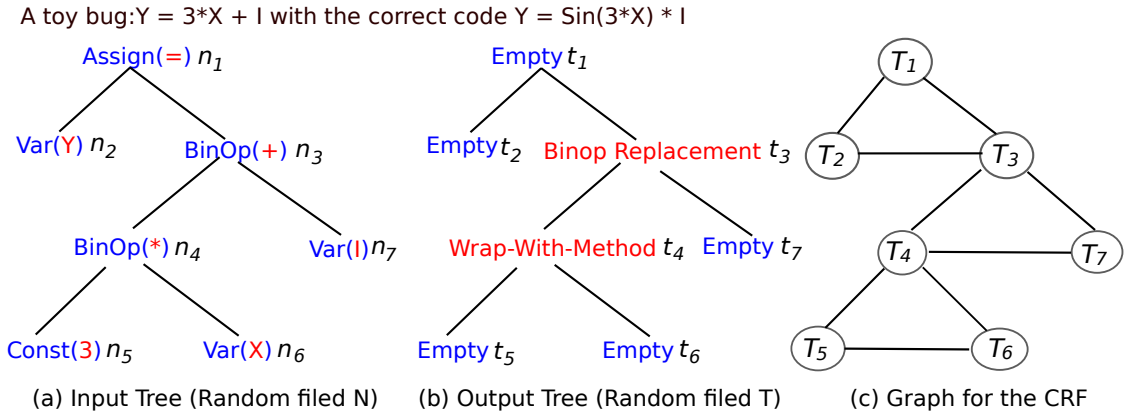

Fig. 2: An example of input tree, output tree, and CRF graph.

complex graph will make the exact inference algorithms become intractable [36]. For the transform prediction problem, we define the undirected graph in the following way:

Definition 4.2. (Graph for random field T) The undirected graph for random field $\mathbf{T} = \{T_1, ..., T_Q\}$ is $G = (V, E)$ such that (1) $V = \{T_1, ..., T_Q\}$ and (2) $E = \{(T_i, T_j) | PC(i, j) \vee IS(i, j)\}$ where $PC(i, j)$ and $IS(i, j)$ denote that there exist parent-child and immediate-sibling relationship between positions i and j in the input AST tree respectively.

The undirected graph is chosen for the following two reasons. On the one hand, the chosen graph structure enables to explore dependencies between transforms applied to different AST nodes in a recursive manner both vertically and horizontally, and the hierarchical nature of AST implies dependencies following these two recursions. On the other hand, the *maximal clique* in the graph is triangle and efficient exact inference algorithm is available for this kind of graph. If the maximal clique in the graph contains more than three nodes, approximate inference algorithms have to be used. Figure 2 (c) shows the undirected graph for the random field **T** in Figure 2 (b). For the input AST tree in Figure 2, there exist parent-child relationship for position pairs <1, 2>, <1, 3>, <3, 4>, <3, 7>, <4, 5>, and <4, 6> and immediate-sibling relationship for position pairs <2, 3>, <4, 7>, and <5, 6>. According to definition 4.2, the graph for random field **T** is thus as shown in Figure 2 (c).

The defined undirected graph contains three kinds of cliques: Node clique C_N, Edge clique C_E, and Triangle clique C_T which contain all vertices, connected edges, and connected triangles in the graph respectively. In the remaining of this paper, unless explicitly specified, we refer to clique of any kind when we mention clique.

According to the above definition of random fields **N** and **T** and the undirected graph for random field **T**, we define a CRF over the transforms **t** to nodes given the observable AST nodes **n** for the code snippet as:

$$p(\mathbf{t}|\mathbf{n}) = \frac{1}{Z(\mathbf{n})} \prod_{c \in C} \psi_c(\mathbf{t}_c, \mathbf{n}) = \frac{1}{Z(\mathbf{n})} \exp\left\{ \sum_{c \in C} \sum_{k=1}^{K} \lambda_k f_k(\mathbf{t}_c, \mathbf{n}) \right\} \tag{3}$$

Like our established CRF model, the observable input variables and the predicted output variables in general have the same structure for most applications of CRFs. Thus, each feature function $f_k(\mathbf{t}_c, \mathbf{n})$ for a certain clique c typically depends on the subset $\mathbf{n}_c$ of **n** in the same clique. Considering this, our established CRF model for transform prediction is as follows:

$$p(\mathbf{t}|\mathbf{n}) = \frac{1}{Z(\mathbf{n})} \exp\left\{ \sum_{c \in C} \sum_{k=1}^{K} \lambda_k f_k(\mathbf{t}_c, \mathbf{n}_c) \right\} \tag{4}$$

where $C = C_N \cup C_E \cup C_T$ and $Z(\mathbf{n}) = \sum_{\mathbf{t} \in \Omega_\mathbf{n}} \left\{ \sum_{c \in C} \sum_{k=1}^{K} \lambda_k f_k(\mathbf{t}_c, \mathbf{n}_c) \right\}$.

The feature functions are the key components of CRFs and the learned weight set $\{\lambda_k\}$ is critical for controlling the probability of a certain assignment **t** given the observable **n**. For instance, to favor the specific assignment $\mathbf{t}_c^s$ for a certain clique c, a feature function $f_j(\mathbf{t}_c^s, \mathbf{n}_c)$ can be defined with a large numerical value. With the weight $\lambda_j > 0$, the assignment with $\mathbf{t}_c^s$ for the clique c will receive a high conditional probability. We will discuss in detail about how feature functions can be defined for our established CRF model in the next section.

While the idea of structured prediction using CRF in principle can be employed to build a general model that is capable of predicting various kinds of code transforms (*e.g.*, feature improvement, refactoring, and bug-fixing) all at the same time, we in this paper focus on establishing a model for predicting a specific kind of code transform. In other words, the random field **T** will be only over a certain kind of code transform when we build the model. Building the general model will involve designing more complex feature functions that can effectively separate different kinds of code transforms.

4.2 Approach for Structured Prediction of AST-level Code Transform Using CRFs

After establishment of the CRF model for code transform prediction, we give how our approach can be achieved in this section.

Workflow. Our approach works in two phases. By using the history code transform information $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^m$ for a set of m examples where $\mathbf{n}^i$ corresponds to the AST for original code (i.e., before transform) of example i and $\mathbf{t}^i$ corresponds to the transforms applied to nodes of $\mathbf{n}^i$, we first use an offline training phase to learn a probabilistic model that captures the conditional probability $p(\mathbf{t} \mid \mathbf{n})$.

Once the model is learned, we then use it to structurally predict the most likely transforms needed for the AST nodes of a new, unseen code snippet.

We next give a step-by-step process for achieving the approach. We first give the framework for defining code transforms on AST nodes, then illustrate how to define the feature functions used in CRFs, and finally describe the training and prediction process.

4.2.1 Framework for Defining AST-level Code Transform

Fine-grained differences between two versions of a code snippet can be described using an edit script which is made up of the following 4 basic tree edit operations:

- **UPD**(*x*, *val*): Update the value of a node *x* with the value *val*.
- **ADD**(*x*, *y*, *i*): Add a new node *x*. If the parent *y* is specified, *x* is inserted as the i^{th} child of *y*, otherwise *x* is added as the new root node.
- **DEL**(*x*): Remove the leaf node *x* from the tree.
- **MOV**(*x*, *y*, *i*): Move the subtree having node *x* as root to make it the i^{th} child of a parent node *y*.

In other words, given two ASTs ast_b, ast_a before and after code changes, an edit script $\Delta = \{e_i | e_i \in \{\text{UPD}, \text{ADD}, \text{DEL}, \text{MOV}\}\}$ is a sequence of basic tree edit operations which can convert ast_b (AST before change) to ast_a (AST after change). We use $|\Delta|$ to denote the number of basic tree edit operations involved for an edit script Δ.

While code changes typically contain repetitive and predictable patterns [2], [16], predicting the edit script itself is typically hard for two reasons. First, a non-trivial code change typically involves multiple tree edit operations and such edit operations can involve complex interactions. Second, predicting a single tree edit operation can involve predicting the edit type, AST nodes involved, and edit position. Our insight is that these low-level tree edit operations embody more high-level code transform patterns, and the patterns can be extracted and attached to AST nodes. By lifting the low-level tree edit operations to high-level code transform patterns on AST nodes, powerful structured prediction can be effectively used to establish the relation between code features and AST-level code transform patterns. We below show how to achieve this lifting.

There in general are many possible edit scripts that can achieve the same code changes, an ideal edit script is one with shortest length.

Definition 4.3. (Shortest Edit Script). Let $\Theta = \{\Delta_i | ast_b \xrightarrow{\Delta_i} ast_a\}$ be the set of edit scripts that can convert ast_b to ast_a. An edit script $\Delta_{min} \in \Theta$ is the shortest edit script if $\forall \Delta^{'} \in (\Theta - \{\Delta_{min}\}), |\Delta^{'}| >= |\Delta_{min}|$.

The shortest edit script can reflect the essence of code changes, and advanced code differencing tools [37], [38] have been shown effective in producing it. For brevity, hereafter the shortest edit script is directly referred to as edit script.

The edit script can contain too fine-grained tree edit operations, in particular some tree edit operations are related to a high-level AST element but are scattered across the edit script. For example, for the code change that inserts a method call "*call(a)*", there are two add operations: $\text{ADD}(n_{call}, y, i)$ and $\text{ADD}(n_a, n_{call}, 1)$, and they are obviously related. Such related but scattered tree edit operations can incur difficulty in extracting high-level concise code transforms. To avoid this issue, we introduce the concept of root tree edit operation. Before the formal definition, we first introduce the concept of mapped node set N_m. AST differencing algorithms proceed by first establishing mappings between the similar nodes of ast_b and ast_a. The result of this process includes N_m which contains the mapped nodes, along with N_o and N_n that respectively contain nodes only in ast_b and ast_a.

Definition 4.4. (Root Tree Edit Operation). A basic tree edit operation $\text{O}(x, \_, \_) \in \{\text{UPD}, \text{ADD}, \text{DEL}, \text{MOV}\}$ is a root tree edit operation if (1) O is MOV operation; or (2) the parent node of *x* belongs to the mapped node set N_m.

Unlike UPD, ADD, and DEL operations that only affect one atomic node but not its descendant nodes, MOV operation moves the subtree rooted at one node. Thus, MOV operation can already reflect high-level concise code changes and is always viewed as root tree edit operation. For the mentioned two ADD operations $\text{ADD}(n_{call}, y, i)$ and $\text{ADD}(n_a, n_{call}, 1)$, only $\text{ADD}(n_{call}, y, i)$ is a root tree edit operation as the parent node n_{call} for the other ADD operation belongs to N_n.

We now give the definition of code transform on AST node.

Definition 4.5. (Code Transform on AST Node). Given the edit script Δ and a set *O* of root tree edit operations for two ASTs ast_b, ast_a before and after code change, the certain code transform *T* on a certain AST node $N \in ast_b$ is a predicate `Tran`(*T*, *N*):

$$\texttt{Tran}(T, N) \triangleq \texttt{TEO}(O, T, N, N^c) \wedge \texttt{CON}(O, T, N, N^c)$$

where $\texttt{TEO}(O, T, N, N^c)$ is a predicate about the root tree edit operations on node *N* and node set N^c that contains nodes that are context related with *N*, and $\texttt{CON}(O, T, N, N^c)$ is a predicate about constraints on node *N* and node set N^c. The constraint predicate can be anything related with the AST, like node label, node value, and parent-child relation. There exists code transform *T* on AST node *N* when `Tran`(*T*, *N*) evaluates to `true`.

If AST node $n \in N_m$, let $n^{\mapsto map}$ denote the mapped node of n in the other AST besides the AST that n resides. Context related node is defined as follows.

Definition 4.6. (Context Related AST node). For a certain AST node *N* in AST ast_b before code change, another AST node *M* is context related with it if one of the following dependence relations is satisfied:

1) (*Data Dependence*) *N* uses or defines a variable whose value is defined in *M*;
2) (*Relative Dependence*) *M* is a descendent, ancestor, or sibling of *N*;
3) (*Mapping Change Dependence*) If $N \in N_m$, *M* is $N^{\mapsto map}$ in AST ast_a after code change or *M* is a data or relative dependence node for $N^{\mapsto map}$ in ast_a; If $N \notin N_m$ but its parent *p(N)* $\in N_m$, *M* is $p(N)^{\mapsto map}$ in ast_a or *M* is a data or relative dependence node for $p(N)^{\mapsto map}$ in ast_a.

4.2.2 Feature Functions for CRFs

Feature functions are key to controlling the likelihood predictions in CRFs. Similar to the feature functions that have been proven useful in other application areas of CRFs [32], [35], we can consider two types of feature functions for AST-level code transform prediction problem.

Observation-based Feature Functions. The first type of feature function is called observation-based feature function. For a certain clique c, observation-based feature functions typically have the form:

$$f_k(\mathbf{t}_c, \mathbf{n}_c) = \mathbf{1}_{\mathbf{t}_c = \mathbf{t}'_c} q_k(\mathbf{n}_c)$$

The notation $\mathbf{1}_{\mathbf{t}_c = \mathbf{t}'_c}$ is an indicator function of $\mathbf{t}_c$ which takes the value 1 when $\mathbf{t}_c = \mathbf{t}'_c$ and 0 otherwise, $q(\mathbf{n}_c)$ is a function on the input $\mathbf{n}_c$ which we call *observation function*. In other words, the feature function is nonzero only for a single output configuration $\mathbf{t}'_c$. But as long as the constraint is met, then the feature value depends only on the input observation $\mathbf{n}_c$. Put it in another way, we can think of observation-based features as depending only on the input $\mathbf{n}_c$, but we have a separate set of weights (after learning) for each output configuration. In this case, for a certain clique c, we can establish the relation between $\mathbf{t}_c$ and $\mathbf{n}_c$ by analyzing the characteristics of input nodes $\mathbf{n}_c$.

For example, for a node clique $c' \in C_N$ which involves a node n' whose label is method call, we can establish a function $q(n')$ which analyzes whether the called method has overloaded method(s) and associates the function with different possible transforms that can be applied on this node. Suppose we are focusing on transforms to repair bugs, since overloaded methods are frequently mixed by developers, hopefully the feature function $\mathbf{1}_{\mathbf{t}_{c'} = \texttt{Meth-RW-Meth}} q(n')$ will have a relatively large weight after learning from large data. Same as mentioned in Section 2, here "`Meth-RW-Meth`" denotes a repair transform that replaces a method invocation by another one, including overloaded methods. For repair transform, we have designed a set of code element features that are likely to be correlated with repair transforms on code elements. After learning, the learned weights for the corresponding feature functions will reflect the strength of the correlation. Table 3, Table 4, and Table 5 respectively show type related, usage related, and syntax related code element features that we have established.

Indicator-based Feature Functions. The other type of feature function is called indicator-based feature function, which can be viewed as a pre-processing step before the launch of the learning phase and typically have the following form for a certain clique c:

$$f_j(\mathbf{t}_c, \mathbf{n}_c) = \mathbf{1}_{\mathbf{t}_c = \mathbf{t}^E_{c'} \wedge \mathbf{n}_c = \mathbf{n}^E_{c'}}$$

Here $\mathbf{n}^E_{c'}$ and $\mathbf{t}^E_{c'}$ denote the input and output for a clique c' observed in any of the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^m$. The idea behind indicator-based feature functions is for each kind of clique c^T (either node, edge or triangle clique), we get from training data all possible input-output tuples $\langle \mathbf{n}_{c^T}, \mathbf{t}_{c^T} \rangle$ for it and transform each input-output tuple into a feature function. By learning from a large representative dataset, a weight then can be associated with each input-output tuple which in turn can be used to do output predictions for new unseen inputs.

In summary, indicator-based feature functions are automatically generated from the training set using a domain-independent procedure. On the contrary, observation-based feature functions can supplement indicator-based feature functions by encoding our domain knowledge about the prediction problem. As previous applications [32], [35] of CRFs have shown, these two types of feature functions altogether can deliver good prediction performance.

4.2.3 Learning and Prediction

Given the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^m$ of m samples, we assume the samples are drawn independently from the underlying joint distribution $p(\mathbf{t}, \mathbf{n})$ and are identically distributed, i.e., the training data are IID. For our task, we just need to perform discriminative training rather than the more difficult generative training, i.e., we just need to estimate $p(\mathbf{t} \mid \mathbf{n})$ directly. The training goal is to automatically compute the optimal weights $\lambda = \{\lambda_k\}_{k=1}^K$ for feature functions in a way that achieves generalization. In other words, for the set of AST nodes $\mathbf{n}$ for a new code snippet drawn from the same distribution P (but not contained in the training dataset D), its needed code transforms $\mathbf{t}$ are predicted correctly by the learned model. There exist several approaches that can potentially be used to accomplish the training task, including for instance *maximum likelihood training* [36] and *max-margin training* [39].

Based on the above defined feature functions and the learned weight for each feature function, for the AST nodes $\mathbf{n}$ of a new code snippet, the conditional probability of each possible transform $\mathbf{t}$ can be calculated by substituting the defined feature functions and the learned weights into formula (4) in Section 4.1.

5 Structured Prediction of Repair Transform Using CRFs

In this section, using repair transform as example, we give a full realization of our approach for structured prediction of code transforms. We first give the definitions of repair transform on AST nodes based on the framework given in Section 4.2.1, then describe how feature functions are constructed for the specific repair transform prediction problem, and finally give the full CRF model learning and prediction algorithms which take the specific issues associated with repair transform prediction into consideration.

5.1 Repair Transform on AST Nodes

Using 16 repair transforms as an example, we show how code transforms are attached to AST nodes in this section. Repair transforms are transforms used to change the buggy code into correct code, and are at the heart of many program repair techniques [40]–[42]. The repair transforms used by these existing repair techniques are not at the level of AST node and are in general tried in a fixed order during repair. Our approach instead is able to account the specific characteristics of a certain bug and predict the needed repair transforms at the level of AST node. The 16 defined repair transforms cover the typical repair actions for the five most common repair patterns [43], [44], including change of `IF` condition expression, method call with different actual parameter values, method call with different number

of parameters or different types of parameters, change of assignment expression, and addition of IF precondition check. Note that our approach itself is not bound to the 16 defined repair transforms, and conceptually it works for any repair transform that would be automatically extracted and attached to AST nodes. Our definition targets object-oriented languages, but can be extended to other languages as well.

We first give some basic definitions, notations, and utility functions. Based on the definition of AST in Section 3, *n*(*lab*, *val*) is used to refer to an AST node with label *lab* and value *val*, $\delta(n)$ and *p(n)* are used to refer to the children nodes and parent node of node *n* respectively, and *l(n)* and *v(n)* are used to refer to the label and value of node *n* respectively. In addition, for a code snippet, ast^r is used to refer to the root node of its AST, and *subast(n)* is used to refer to the sub AST rooted at node *n*. Here, the leaf nodes of the sub AST are always leaf nodes of the original AST. Thus, the sub AST referred by *subast(n)* is unique for a given code snippet and a certain node *n*. Finally, if node *n* is a mapped node produced by the AST differencing algorithm (i.e., $n \in N_m$), we use $n^{\mapsto map}$ to denote the mapped node of n in the other AST as in Section 4.

Other basic definitions, notations, and utility functions are presented in Figure 3. We first give notations for basic tree edit operations (**AO**) and program elements in mainstream programming languages, including binary operator, logical operator, ternary operator, literal, logical expression, and code block (**CB**). In particular, the logical expression here denotes an expression made up by a set of atomic boolean expressions, and it cannot be extended with other atomic boolean expressions. We next give some utility mapping functions. For a code snippet, the function *root* achieves the mapping to its root AST node. When the label $l(n)$ of an AST node *n* is VariableAccess or MethodCall, the function *def* maps node *n* to the root node of definition code for $v(n)$. When the label $l(n)$ of an AST node *n* is Conditional − If, LogicalOperator or TernaryOperator, the function *LogExp* maps node *n* to the root node of the related logical expression. We then give some definitions concerning basic tree edit operations on AST nodes, and the definition is in the from of predicate. *TE*(*e*, *n*) is a predicate about whether there exists tree edit operation *e* on node *n*. Based on predicate *TE*(*e*, *n*), the predicates *NoE*(*n*), *ET*(*n*, *e*), and *NoDE*(*n*) are respectively about whether there exists any tree edit operation on node *n*, whether all nodes of the sub AST rooted at node *n* are subject to tree edit operation *e*, and whether there exists any tree edit operation on related definition code for node *n*. Here, *TE*, *NoE*, *ET*, and *NoDE* stand for the abbreviations for "tree edit", "no edit", "edit tree", and "no definition edit" respectively. We finally give some mapping functions and definitions related with code block. When the label $l(n)$ of an AST node *n* is Conditional − If or Try − Catch, the function *block* maps node *n* to the related code block $\mathbf{CB}^l$ with label *l*. The functions *move* and *min* map a code block to its subset which contains only moved statements and its enclosing statement with the smallest line index respectively. To facilitate the description, *comb*(*n*, *l*) is defined as the combination of functions *min*, *move*, and *block*.

Table 1 and Table 2 give the definitions for 16 repair transforms in detail, and Figure 4 uses examples to illustrate the repair transforms corresponding to the definitions in the two tables. For the definitions, the superscripts 'O' and 'N' are used to distinguish nodes in ASTs before and after code change respectively, the symbols 'Wrap', 'Unwrap', and 'RW' in transform name are used to denote that an existing program element is wrapped with other newly added constructs (e.g., method call, conditional check), an existing program element is unwrapped from other existing constructs, and an existing program element is replaced with other program elements respectively.

Table 1 gives the definitions for some repair transforms that target the inner nodes of a statement. Wrap − Meth moves an existing expression into an added method call and Unwrap − Meth instead moves an existing expression out of a removed method call. To avoid the case that the move operations arise because of changes in the signature of the involved method call, we add a constraint that there are no root tree edit operations on the root node of the method definition. Note that when the definition explicitly involves a certain variable access or method call, we in general have constraints about the tree edit operations on the definitions of the accessed variable or method. Var − RW − Var, Var − RW − Meth, Meth − RW − Meth, and Meth − RW − Var achieve the replacement of variable access and method call. In particular, there are two sub-cases for Meth − RW − Meth: the names of the replaced and original method calls are (1) different or (2) the same (i.e., method overload). BinOper − Rep and Constant − Rep represent the replacement of binary operator and constant literal respectively. LogExp − Exp and LogExp − Red expand and reduce an existing logical expression respectively. As logical expression can typically be expanded in different ways when it contains several atomic boolean expressions, we thus associate both these two repair transforms to the root node of the logical expression.

Other transforms target majorly the whole statement and are given in Table 2. Wrap − IF − N adds an 'If' conditional check (without 'Else' block) for an existing statement that is subject to MOV operation. Note that some other tree edit operations on the 'Then' block can be accompanied by the add of the 'If' conditional check. In such cases, we view the first statement in the 'Then' block (whose actual root node is subject to MOV operation) as the 'old' statement and the target of the conditional check. Wrap − IFELSE − N adds an 'If' conditional check with the 'Else' block. For this transform, we have 3 cases: the 'old' (moved) statement is wrapped by (1) 'Then' block, or (2) 'Else' block, or (3) the added check is in the form of ternary expression. For both Wrap − IF − N and Wrap − IFELSE − N, the logical condition of the added 'If' check has the node $n(Literal, NULL)$ (i.e., the added check is null check). Similarly, we consider additional transforms Wrap − IF − O and Wrap − IFLSE − O (not shown in Table 2 for space reason) for which the added 'If' check is not related to null check. Unwrap − IF removes the conditional check, and the check can be in the form of 'If' expression or ternary expression. Finally, Wrap − TRY warps an existing statement with "*Try-Catch*" exception handle.

For "*Wrap-If*" and "*Wrap-Try*" related transforms that target the whole statement s, we introduce the "virtual root" node (denoted as $s^{\mapsto vr}$) and attach transforms to it

AO = {ADD, DEL, MOV, UPD} **BinaryOperator** = {| |, &&, |, ;, &, ==, !=, <, >, <=, >=, «, », +, -, *, /,% }
LogicalOperator = {| |, &&} **TernaryOperator** = {?:} **Literal** = {Number, String, null} **CB** = $\{s_1, ..., s_n\}$
LogicalExpression = BoolExp | |(&&)... | |(&&)BoolExp $root : \text{code} \rightarrow ast^r$ $def : n \rightarrow root(v(n)^{\mapsto DEF})$
LogExp: $n \rightarrow root(n^{\mapsto \textbf{LogicalExpression}})$ $TE(e,n) \triangleq \exists\, e \in \textbf{AO},\ e(n,\_,\_)$ $NoE(n) \triangleq \forall e \in \textbf{AO} : \neg TE(ao, n)$
$ET(n,e) \triangleq \forall n' \in subast(n) : TE(e,n')$ $NoDE(n) \triangleq \forall e \in \textbf{AO} \forall n' \in \delta(def(n)) : \neg TE(e, def(n)) \wedge \neg TE(e, n')$
block:$n \times l \rightarrow \textbf{CB}^l$ $move : \textbf{CB} \rightarrow \{s \in \textbf{CB} | TE(\text{MOV}, root(s))\}$ $min : \textbf{CB} \rightarrow s^{min}$ $comb(n,l) \triangleq min \circ move \circ block(n,l)$

Fig. 3: Definitions, notations, and utility functions used for defining repair transforms on AST nodes.

TABLE 1: Definitions of the repair transforms that target the inner AST nodes of a statement.

Repair Transform `Tran`	Predicate About Root Tree Edit Operations `TEO`	Predicate About Constraints `CON`
`Tran` (`Wrap-Meth`, n_1^O)	$\text{MOV}(n_1^O, n_2^N, i) \wedge \text{ADD}(n_2^N, n_3^O, j) \wedge NoDE(n_2^N)$	$l(n_2^N) = \textit{MethodCall} \wedge n_3^O = p(n_1^O)$
`Tran` (`Unwrap-Meth`, n_1^O)	$\text{MOV}(n_1^O, n_2^O, i) \wedge \text{DEL}(n_3^O) \wedge NoDE(n_3^O)$	$l(n_3^O) = \textit{MethodCall} \wedge n_3^O = p(n_1^O) \wedge n_2^O = p(n_3^O)$
`Tran` (`Var-RW-Var`, n_1^O)	$\text{UPD}(n_1^O, val) \wedge NoDE(n_1^O) \wedge NoDE\big((n_1^O)^{\mapsto map}\big)$	$l(n_1^O) = \textit{VariableAccess}$
`Tran` (`Var-RW-Meth`, n_1^O)	$\text{DEL}(n_1^O) \wedge \text{ADD}(n_2^N, n_3^O, j) \wedge NoDE(n_1^O) \wedge NoDE(n_2^N)$	$l(n_1^O) = \textit{VariableAccess} \wedge l(n_2^N) = \textit{MethodCall} \wedge n_3^O = p(n_1^O)$
`Tran`^{1}(`Meth-RW-Meth`, n_1^O)	$\text{UPD}(n_1^O, val) \wedge NoDE(n_1^O) \wedge NoDE\big((n_1^O)^{\mapsto map}\big)$	$l(n_1^O) = \textit{MethodCall}$
`Tran`^{2}(`Meth-RW-Meth`, n_2^O)	$\text{DEL}(n_1^O) \wedge NoDE(n_2^O)$	$n_2^O = p(n_1^O) \wedge l(n_2^O) = \textit{MethodCall}$
`Tran` (`Meth-RW-Var`, n_1^O)	$\text{DEL}(n_1^O) \wedge \text{ADD}(n_2^N, n_3^O, j) \wedge NoDE(n_1^O) \wedge NoDE(n_2^N)$	$l(n_1^O) = \textit{MethodCall} \wedge l(n_2^N) = \textit{VariableAccess} \wedge n_3^O = p(n_1^O)$
`Tran`(`BinOper-Rep`, n_1^O)	$\text{UPD}(n_1^O, val) \wedge \forall n' \in \delta(n_1^O) : NoE(n')$	$l(n_1^O) = \textit{BinaryOperator}$
`Tran` (`Constant-Rep`, n_1^O)	$\text{UPD}(n_1^O, val) \wedge NoE(p(n_1^O))$	$l(n_1^O) = \textit{Literal}$
`Tran` (`LogExp-Exp`, $LogExp\,(n_2^O)$)	$\text{ADD}(n_1^N, n_2^O, i) \wedge \exists n' \in \delta(n_1^N) : NoE(n') \wedge \exists n'' \in \delta(n_1^N) : ET(n'', \text{ADD})$	$l(n_1^N) = \textit{LogicalOperator}$
`Tran` (`LogExp-Red`, $LogExp\,(n_1^O)$)	$\text{DEL}(n_1^O) \wedge \exists n' \in \delta(n_1^O) : NoE(n') \wedge \exists n'' \in \delta(n_1^O) : ET(n'', \text{DEL})$	$l(n_1^O) = \textit{LogicalOperator}$

TABLE 2: Definitions of the repair transforms that target majorly the virtual root node of a statement.

Repair Transform `Tran`	Predicate About Root Tree Edit Operations `TEO`	Predicate About Constraints `CON`
`Tran` (`Wrap-IF-N`, $comb\,(n_1^N, \texttt{THEN})^{\mapsto vr \mapsto map}$)	$\text{ADD}(n_1^N, n_2^O, i) \wedge move\big(block(n_1^N, THEN)\big)! = \emptyset$	$l(n_1^N) = \textit{Cond If} \wedge block(n_1^N, ELSE) = \emptyset \wedge n(Literal, NULL) \in subast\Big(root\big(LogExp(n_1^N)\big)\Big)$
`Tran`^{1} (`Wrap-IFELSE-N`, $comb\,(n_1^N, \texttt{THEN})^{\mapsto vr \mapsto map}$)	$\text{ADD}(n_1^N, n_2^O, i) \wedge move\big(block(n_1^N, THEN)\big)! = \emptyset \wedge move\big(block(n_1^N, ELSE)\big) = \emptyset$	$l(n_1^N) = \textit{Cond If} \wedge block(n_1^N, ELSE)! = \emptyset \wedge n(Literal, NULL) \in subast\Big(root\big(LogExp(n_1^N)\big)\Big)$
`Tran`^{2} (`Wrap-IFELSE-N`, $comb\,(n_1^N, \texttt{ELSE})^{\mapsto vr \mapsto map}$)	$\text{ADD}(n_1^N, n_2^O, i) \wedge move\big(block(n_1^N, THEN)\big) = \emptyset \wedge move\big(block(n_1^N, ELSE)\big)! = \emptyset$	$l(n_1^N) = \textit{Cond If} \wedge block(n_1^N, THEN)! = \emptyset \wedge n(Literal, NULL) \in subast\Big(root\big(LogExp(n_1^N)\big)\Big)$
`Tran`^{3} (`Wrap-IFELSE-N`, n_1^O)	$\text{MOV}(n_1^O, n_2^N, i) \wedge \text{ADD}(n_2^N, n_3^O, j)$	$l(n_2^N) = \textit{TernaryOperator} \wedge n_3^O = p(n_1^O) \wedge n(Literal, NULL) \in subast\Big(root\big(LogExp(n_2^N)\big)\Big)$
`Tran`^{1} (`Unwrap-IF`, n_1^O)	$\text{DEL}(n_1^O) \wedge move\big(block(n_1^O, THEN)\big) \cup move\big(block(n_1^O, ELSE)\big)! = \emptyset$	$l(n_1^O) = \textit{Cond If}$
`Tran`^{2}(`Unwrap-IF`, n_1^O)	$\text{DEL}(n_1^O) \wedge \text{MOV}(n_2^O, n_3^O, i)$	$l(n_1^O) = \textit{TernaryOperator} \wedge n_1^O = p(n_2^O) \wedge n_3^O = p(n_1^O)$
`Tran` (`Wrap-TRY`, $comb\,(n_1^N, \texttt{TRY})^{\mapsto vr \mapsto map}$)	$\text{ADD}(n_1^N, n_2^O, i) \wedge move\big(block(n_1^N, TRY)\big)! = \emptyset$	$l(n_1^N) = \textit{Try}$

rather than the actual root node $root(s)$. The aim is to separate other transforms that can possibly be attached to $root(s)$. For instance, for a method invocation statement whose actual root node is the called method, there can exist both `Wrap` − `IF` − `N` transform and `Meth` − `RW` − `Meth` transform that replaces the called method, and `Meth` − `RW` − `Meth` transform is attached to the actual root node. The "virtual root" node $s^{\mapsto vr}$ is inserted between the actual root node $root(s)$ and its parent node. Doing this also facilities the CRF model construction as it typically has a single-label per input assumption, i.e., single repair transform per AST node for our problem. For "virtual root" node $s^{\mapsto vr}$, we view the label and value of it as 'virtual' and 'null' respectively.

Example For the code diff in Figure 1, the edit script consists of two root tree edit operations **UPD**(⑪, z) and **UPD**(⑬, z) where ⑪ and ⑬ denote the 11th and 13th AST node respectively (as shown in Figure 1). Since the definitions for the old method *y* and the new method *z* have not changed, the predicates $NoDE(⑪^O)$ and $NoDE\big((⑪^O)^{\mapsto map}\big)$ evaluate to true (the part of predicate `TEO` about root tree edit operations). Meanwhile, the predicate "$l(⑪^O) = \textit{MethodCall}$" also evaluates to true (the part of predicate `CON` about constraints). Consequently, the predicate `Tran`^{1}(`Meth` − `RW` − `Meth`, $⑪^O$) evaluates to true according to the definition given in Table 1, implying that the repair transform "`Meth-RW-Meth`" is attached to node ⑪. Similarly, we can establish that the repair transform "`Meth-RW-Meth`" is also attached to node ⑬.

5.2 Feature Functions for Repair Transform Prediction

We in this section describe how observation-based and indicator-based feature functions can be constructed for the repair transform prediction problem.

Fig. 4: Examples to illustrate the definitions of repair transforms.

5.2.1 *Observation-based Feature Functions*

For a certain clique c, observation-based feature functions establish the relation between $\mathbf{t}_c$ and $\mathbf{n}_c$ by analyzing the characteristics of input nodes $\mathbf{n}_c$. With regard to repair transform, this basically means designing a set of node characteristics that are likely to be correlated with repair transforms on them. After learning, the learned weights for the corresponding feature functions then reflect the strength of the correlation.

We design observation-based feature functions related with different kinds of program elements reflected in AST nodes, including variable access, method call, logical expression, binary operator, and the whole statement. We first present the characteristics that we analyze and then present the observation-based feature functions on top of them.

Node Characteristics. Depending on the label of the AST node, we accordingly analyze different kinds of characteristics associated with it and the characteristics we statically analyze can be classified into 3 kinds based on their nature: type related, usage related, and syntax related.

Table 3, Table 4, and Table 5 respectively show type related, usage related, and syntax related node characteristics that we have established. For type related characteristics, we majorly explore whether the type is primitive or objective and whether certain type compatible relation holds. We investigate types of different kinds of program elements, including variables in different forms (local variable, method parameter, etc.) and returns of method definitions or method calls. In addition, we also examine whether types of certain program elements (regardless of same kind or different kind) are compatible with each other when the type comparison is applicable. For usage related characteristics, we consider how the variable or method call has been used elsewhere in the program and whether certain substitutions of variable or method call can result in other program elements in the program. We explore whether variable or method call has been used elsewhere with null check guard, normal check guard, and exception handle. When they are used as a method call parameter, we explore whether substitution between different variables (resp. different method calls) or substitution between variable and method call can result in another method call. For variable, we additionally investigate whether it has been referenced or assigned elsewhere and we distinguish local variable from field. For syn-

TABLE 3: Type related node characteristics.

Node Kind	Characteristic Description
Variable Access (*var*)	(V1): The type of *var* is primitive.
	(V2): The type of *var* is objective.
	(V3): *var* is an instance of the class that it resides.
	(V4): There exist variables in scope that are type compatible with *var*.
	(V5): There exist method definitions or calls for which at least one of their parameters is type compatible with *var*.
	(V6): There exist method definitions or calls whose return types are compatible with *var*.
Method Invocation (*m*)	(M1): The return type of *m* is primitive.
	(M2): The return type of *m* is objective.
	(M3): The types of some parameters of *m* are compatible with the return type of *m*.
	(M4): There exist variables in scope that are type compatible with the return type of *m*.
	(M5): There exist method definitions or calls whose return types are compatible with that of *m*.

TABLE 4: Usage related node characteristics.

Node Kind	Characteristic Description
Variable Access (*var*)	(V7): If *var* is a local variable, it has not been referenced before the statement that *var* resides.
	(V8): If *var* is a local variable, it has not been assigned before the statement that *var* resides.
	(V9): If *var* is a field, it has not been referenced in other statements of the class besides the statement that *var* resides.
	(V10): If *var* is a field, it has not been assigned in other statements of the class besides the statement that *var* resides.
	(V11): There exist other statements in the class that use some same type variables with *var*, but have null check guard.
	(V12): There exist other statements in the class that use some same type variables with *var*, but have normal check guard.
	(V13): When *var* is a parameter of a method call m_1, replace *var* with another variable var' can get another method call m_2 used in the class.
	(V14): When *var* is a parameter of a method call m_1, replace *var* with a method call m' can get another method call m_2 used in the class.
Method Invocation (*m*)	(M6): There exist other statements in the class that use a method call m' whose signature is same with *m*, but have null check guard.
	(M7): There exist other statements in the class that use a method call m' whose signature is same with *m*, but have normal check guard.
	(M8): There exist other statements in the class that use a method call m' whose signature is same with *m*, but have try-catch exception handle.
	(M9): When *m* is a parameter of a method call m_1, replace *m* with a variable *var* can get another method call m_2 used in the class.
	(M10): When *m* is a parameter of a method call m_1, replace *m* with a method call m' can get another method call m_2 used in the class.

TABLE 5: Syntax related node characteristics.

Node Kind	Characteristic Description
Variable Access (*var*)	(V15): There exist variables in scope that are similar in identifier name with *var*.
	(V16): There exist method definitions or calls that are similar in identifier name with *var*.
Method Invocation (*m*)	(M11): The identifier name of *m* starts with 'get'.
	(M12): *m* has overloaded method(s).
	(M13): There exist variables in scope that are similar in identifier name with *m*.
	(M14): There exist method definitions or calls that are similar in identifier name with *m*.
Binary Operator (*bo*)	(BO1): The operator kind of *bo*.
	(BO2): When *bo* is a logical operator, its operands contain the exclamation mark!.
	(BO3): When *bo* is a logical operator, its operands contain the literal 'null'.
	(BO4): When *bo* is a logical operator, its operands contain the number '0' or '1'.
Root Node	(LE1): There exists an atomic boolean expression that contains the exclamation mark !.
	(LE2): There exists an atomic boolean expression that is simply a boolean variable.
	(LE3): There exist two atomic boolean expressions ae_1 and ae_2 that are respectively null check and normal check.
Virtual Root Node	(S1): The statement kind of *s*.
	(S2): The statement kind of the previous statement in the same block with *s*.
	(S3): The statement kind of the next statement in the same block with *s*.
	(S4): The statement kind of the parent statement of *s*.
	(S5): The associated method of *s* throws exception or the associated class of *s* extends an exception type class.

tax related characteristics, we explore whether certain program elements have some specific syntax attributes. With regard to variable and method call, we majorly investigate whether their identifier names are special, including similar with that of others or start with the special character 'get'. For binary operator and root node for logical expression, we primarily study whether their children are unusual, including whether or not contain the exclamation mark !, the literal 'null', the number '0', or the number '1'. In terms of virtual root node for a certain statement s, we mainly explore the statement kinds of several statements related with s, including the statement s itself, previous statement of s, next statement of s, and parent statement of s.

When the characteristic involves measure of similarity (`V15`, `V16`, `M13`, and `M14`), we use Levenshtein distance metric to establish the difference between two string sequences. For the characteristics related with statement kind (`S1-S4`) or binary operator kind (`BO1`), we enumerate all possible kinds and establish a sub-characteristic *"The kind is X"* for each possible kind X. Typical binary operator kinds in mainstream languages include logical relation, bit operation, equality comparison, shift operation, and mathematical operation as shown in Figure 3. After doing this, each of the characteristics can be viewed as a boolean valued function, i.e, a predicate on the characteristics. We use $n.c$ to denote the boolean evaluation result of a certain characteristic c on node n.

Characteristic Propagation. As there exist structural dependencies between different AST nodes, the characteristic of a certain node can possibly imply repair transforms on other nodes. For instance, the characteristic `V7` can be related with *"Wrap-If"* related transform(s) on the virtual root node of the statement. To account this, we propagate some characteristics of certain child nodes to their parent nodes. First, we propagate the characteristics `V7`, `V8`, `V9`, `V10`, `V11`, and `V12` for a variable access node n_1 to the virtual root node of the statement and the method access node n_2 that is an ancestor of n_1. Second, we propagate the characteristics `M6`, `M7`, and `M8` for a method access node to the virtual root node of the statement. When propagating a certain characteristics c from a node n_1 to another node n_2, the corresponding characteristic c' for n_2 is "There exists at least one descendent node that has characteristic c", and the predicate value is calculated as follows:

$$n_2.c' = n_{c1}.c \vee n_{c2}.c \vee ... \vee n_{ck}.c$$

where n_{c1} to n_{ck} represent k descendent nodes of n_2 that have characteristic c, including n_1.

Observation-based Feature Functions. After analyzing the characteristics, the observation function $q(\mathbf{n}_c)$ can then be designed as indicator function of the characteristics. Let $C^L = \{c^i\}_{i=1}^n$ denote the set of characteristics we have established for node whose label is L. For a node n and a characteristic $c \in C^L$, we define the observation function as follows:

$$q(n, c) = \mathbf{1}_{l(n)=L \wedge n.c=true}$$

Then, we need to correlate the observation function with repair transforms. For different types of nodes, the set of possible repair transforms on them are different. To see what transforms are possible for a certain node, we make use of information from the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^m$ of m samples. For repair transform prediction problem, the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^m$ consists of m buggy programs where for each program, the repair transforms required to fix the bug are associated to appropriate AST nodes. For an AST node $n \in \mathbf{n}$, we use *t(n)* to denote the associated repair transform on it, and we define the viable transform set for a node whose label is L as follows:

$$T(L) = \{t | \exists \langle \mathbf{t}, \mathbf{n} \rangle \in D, \exists n \in \mathbf{n} : l(n) = L, t(n) = t\}$$

That is, we deem a repair transform possible for a certain type of node when we have observed the occurrence of this at least once in the training data. Let γ represent the set of possible labels for a node, we finally define observation-based feature functions on top of the observation functions and the viable transform set as follows:

$$f(t, n) = \mathbf{1}_{L \in \gamma \wedge l(n)=L \wedge t \in T(L) \wedge c \in C^L} q(n, c)$$

Intuitively speaking, for each possible repair transform t on a node whose label is L, we associate it with each of the characteristics we have designed for nodes with label L to form a feature function. Through training on big data, we can get weights for different transform-characteristic pairs.

Note that we in this paper define observation-based feature functions on node cliques, and it is possible to define more complex observation-based feature functions on edge cliques and triangle cliques by analyzing the characteristics involving all nodes in edge and triangle cliques.

5.2.2 Indicator-based Feature Functions

Given the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^m$ of m sample repair examples, we now formally describe how we establish indicator-based feature functions. Recall that *l(n)*, *v(n)*, and *t(n)* are used to denote node label, node value, and repair transform on node respectively. Here, we also use $n^{\mapsto i}$ to represent the ith child node of node n. For each tuple $\langle \mathbf{t}, \mathbf{n} \rangle \in D$, we define the observed set of transforms on nodes for different kinds of cliques as follows:

$$nodetran(\mathbf{t}, \mathbf{n}) = \{(T, L) | \exists n \in \mathbf{n} : l(n) = L, t(n) = T\}$$

$$edgetran(\mathbf{t}, \mathbf{n}) = \{(T_1, L_1, T_2, L_2) | \exists n \in \mathbf{n}, i \in \mathbb{N} : l(n) = L_1, t(n) = T_1, l(n^{\mapsto i}) = L_2, t(n^{\mapsto i}) = T_2\}$$

$$triangletran(\mathbf{t}, \mathbf{n}) = \{(T_1, L_1, T_2, L_2, T_3, L_3) | \exists n \in \mathbf{n}, i \in \mathbb{N} : l(n) = L_1, t(n) = T_1, l(n^{\mapsto i}) = L_2, t(n^{\mapsto i}) = T_2, l(n^{\mapsto (i+1)}) = L_3, t(n^{\mapsto (i+1)}) = T_3\}$$

In other words, we enumerate possible repair transforms on nodes for different cliques observed in the specific training example $\langle \mathbf{t}, \mathbf{n} \rangle$. For the entire training data D, we can then obtain all observed set of transforms on nodes as follows:

$$all\_nodetran(D) = \cup_{i=1}^m nodetran(\mathbf{t}^i, \mathbf{n}^i)$$
$$all\_edgetran(D) = \cup_{i=1}^m edgetran(\mathbf{t}^i, \mathbf{n}^i)$$
$$all\_triangletran(D) = \cup_{i=1}^m triangletran(\mathbf{t}^i, \mathbf{n}^i)$$

Based on the observed set of transforms on nodes, we finally define indicator-based feature functions for different kinds of cliques as follows:

$$f(t_1, n_1) = \mathbf{1}_{(t_1, l(n_1)) \in all\_nodetran(D)}$$
$$f(t_1, n_1, t_2, n_2) = \mathbf{1}_{(t_1, l(n_1), t_2, l(n_2)) \in all\_edgetran(D)}$$
$$f(t_1, n_1, t_2, n_2, t_3, n_3) =$$
$$\mathbf{1}_{(t_1, l(n_1), t_2, l(n_2), t_3, l(n_3)) \in all\_triangletran(D)}$$

where t_i corresponds to the repair transform associated with node n_i. For edge clique, n_1 and n_2 are parent and child node respectively. For triangle clique, n_1, n_2 and n_3 are parent, left child and right child node respectively. Note that in the remaining of this paper, we use the same notation as here.

The above defined indicator-based feature functions do not take the value of the nodes into account. To study the repair transforms associated with triangle clique when the value of the left child is the same with that of the right child (e.g., same variable access), we define another set of indicator-based feature functions for triangle clique as follows:

$$triangletran^{spe}(\mathbf{t}, \mathbf{n}) = \{(T_1, L_1, T_2, L_2, T_3, L_3) | \exists n \in \mathbf{n}, i \in \mathbb{N} : l(n) = L_1, t(n) = T_1, l(n^{\mapsto i}) = L_2, t(n^{\mapsto i}) = T_2, l(n^{\mapsto (i+1)}) = L_3, t(n^{\mapsto (i+1)}) = T_3, L_2 = L_3, v(n^{\mapsto i}) = v(n^{\mapsto (i+1)})\}$$
$$all\_triangletran^{spe}(D) = \cup_{i=1}^{m} triangletran^{spe}(\mathbf{t}^i, \mathbf{n}^i)$$
$$f^{spe}(t_1, n_1, t_2, n_2, t_3, n_3) = \mathbf{1}_{(t_1, l(n_1), t_2, l(n_2), t_3, l(n_3)) \in all\_triangletran^{spe}(D) \wedge v(n_2) = v(n_3)}$$

In the learning phase, we learn the corresponding weight for each of the indicator-based feature functions defined above. Note that the indicator-based feature functions and the learned weights for them can vary depending on the training data *D*, but they are independent of the buggy code snippet for which we are trying to predict repair transforms.

5.3 Learning and Prediction

We in this section describe how to train the CRF model for repair transform prediction and use the already trained CRF model to do prediction.

5.3.1 Learning

Recall that the learning problem is to determine the optimal weights $\lambda = \{\lambda_k\}_{k=1}^{K}$ for feature functions from the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^{m}$. The typical way to train CRF model is using classical *penalized maximum (log)-likelihood* [36], which optimizes the following log-likelihood objective function with respect to the model $p(\mathbf{t} \mid \mathbf{n}, \lambda)$:

$$l(\lambda) = \sum_{i=1}^{m} \log p(\mathbf{t} = \mathbf{t}^i | \mathbf{n} = \mathbf{n}^i; \lambda)$$

In other words, the weights for feature functions are chosen such that the training data has the highest probability under the model.

Transform Number Imbalance Issue. The above objective function treats each $p(\mathbf{t} = \mathbf{t}^i | \mathbf{n} = \mathbf{n}^i; \lambda)$ as equally important. However, one significant characteristic in repair transform prediction problem is that the buggy code snippet is nearly correct, and typically just a few actual repair transforms on certain AST nodes are needed. Note that we attach a virtual 'EMPTY' repair transform to those nodes which are not associated with any repair transforms. Thus, the training data $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^{m}$ is skewed in turns of the number of AST nodes that are associated with actual repair transforms. If the skew is too large, the learned weights $\lambda = \{\lambda_k\}_{k=1}^{K}$ will be dominated by those training examples with few repair transforms on few nodes. However, correctly predicting those instances which need relatively more repair transforms on nodes is important as those bugs are much harder to deal with. We call this issue "*transform number imbalance issue*".

To deal with the training data imbalance problem, there are typically three groups of solutions: sampling methods [45], cost-sensitive learning [46], and one-class learning [47]. For our repair transform prediction problem, we propose a method called "transform distribution aware learning". The method is similar to sampling methods, but does not have the disadvantage of removing important examples in under-sampling and adding redundant examples in over-sampling. The method analyzes the training data before the launch of training and gives more weight on those training examples which have relatively more nodes associated with actual repair transforms. Formally, for the training dataset $D = \{\langle \mathbf{t}^i, \mathbf{n}^i \rangle\}_{i=1}^{m}$, we define the set U which contains all the observed numbers of actual repair transforms used for repairing a bug as follows:

$$U = \{n | n \in \mathbb{Z}^+, \exists \langle \mathbf{t}^i, \mathbf{n}^i \rangle \in D : S(\mathbf{t}) = n\}$$

where S(**t**) denotes the number of actual repair transforms in **t**.

We then define a distribution-aware prior χ_i as:

$$\overline{N} = \frac{1}{|U|} \sum_{u \in U} N_u \,, \quad \chi_i = \Big(\frac{\overline{N}}{N_{S(\mathbf{t}^i)}} \Big)^q$$

where N_u is the number of training examples in D that have u actual repair transforms, $\overline{N}$ is the average number of training examples per each number in U, and q is a coefficient that controls the magnitude of the distribution-aware prior.

Finally, we multiply the distribution-aware prior with the log probability for each training example $\langle \mathbf{t}^i, \mathbf{n}^i \rangle$ in the objective function and get a new objective function:

$$l(\lambda) = \sum_{i=1}^{m} \chi_i \log Pr(\mathbf{t} = \mathbf{t}^i | \mathbf{n} = \mathbf{n}^i; \lambda)$$

Note that when the training dataset D has a uniform distribution (i.e., for each $u \in U$, N_u is equal) or when the coefficient q equates to 0, the new objective function is reduced to the typical objective function. Through the use of distribution-aware prior, more weights can be put on those training examples which have relatively more actual repair transforms attached to nodes and are scarce in D. The larger the coefficient q, the more weights we put on those types of training examples which are scarce in D. Overall, by using the distribution-aware prior, all the training examples in D can be adjusted to have a balanced impact in the learning process.

Regularization. As our CRF model contains a large number of feature functions, we use *regularization* to penalty

on weight vectors whose norm is too large (i.e., avoid overfitting). We use the typical penalty based on the Euclidean norm of λ and the strength of the penalty is determined by the parameter $1/2\delta^2$. The regularized objective function is then:

$$l(\lambda) = \sum_{i=1}^{m}\sum_{c\in C}\sum_{k=1}^{K}\chi_i\lambda_k f_k(\mathbf{t}_c^i, \mathbf{n}_c^i) - \sum_{i=1}^{m}\chi_i \log Z(\mathbf{n}^i) - \sum_{k=1}^{K}\frac{\lambda_k^2}{2\delta^2}$$

The above function is concave and every local optimum is also a global optimum. But $l(\lambda)$ in general cannot be maximized in closed form, so numerical optimization is used. We use the particularly successful method L-BFGS [48]. L-BFGS belongs to the family of quasi-Newton methods and can be used as a black-box optimization routine by feeding the value and the first derivative of the objective function.

The first derivative of the objective function $l(\lambda)$ for each parameter λ_k is:

$$\frac{\partial l(\lambda)}{\partial \lambda_k} = \sum_{i=1}^{m}\sum_{c\in C}\chi_i f_k(\mathbf{t}_c^i, \mathbf{n}_c^i) - \sum_{i=1}^{m}\sum_{c\in C}\sum_{\mathbf{t}_c^{i^+}\in\Phi_c}\chi_i p(\mathbf{t}_c^{i^+}|\mathbf{n}_c^i) \cdot f_k(\mathbf{t}_c^{i^+}, \mathbf{n}_c^i) - \frac{\lambda_k}{\delta^2}$$

where Φ_c ranges over all assignments to $\mathbf{t}$ in the clique c. The computation of the first term is straightforward (i.e., sum the feature function values over the training dataset), but calculating the second term requires to calculate the marginal probability $p(\mathbf{t}_c|\mathbf{n}_c)$, which is an inference task and we will discuss it below.

5.3.2 Prediction

Recall that prediction involves substituting the defined feature functions and the learned weights for them into formula (4) in Section 4.1 to get the conditional probability of each possible transform $\mathbf{t}$ for the AST nodes $\mathbf{n}$.

Typically, CRFs output the single most likely prediction by using the MAP (Maximum a Posteriori) [31] query: $\mathbf{t} = \text{argmax}_{\mathbf{t}'\in\Omega_\mathbf{n}} p(\mathbf{t}'|\mathbf{n})$. As said in the section about learning, the learning process needs to calculate the marginal probability $p(\mathbf{t}_c|\mathbf{n}_c)$ for a certain clique c. These are the two inference problems that arise in CRFs, and *can be seen as fundamentally the same operation on two different semirings* [36]. To change the marginalization problem to the maximization problem, we just need to substitute maximization calculation for addition calculation.

When the associated undirected graphs with CRF model have cycles, typically approximate inference algorithms have to be used. However, one advantage of our CRF model is that the maximal clique in the undirected graph is triangle, for which efficient exact inference algorithms are available. The process is first using *junction tree algorithm* to change the graph into a tree, and then *belief propagation algorithm* can be used to do the inference [36]. We refer readers to [49] and [50] for details about junction tree algorithm and belief propagation algorithm respectively.

The belief propagation algorithm is also called message-passing algorithm, and the marginal distributions are recursively computed using messages exchanged between all the nodes in the junction tree. When it comes to the original undirected graphs for CRFs, let c be a maximal clique and the set N^c be its neighbour maximal clique set, i.e., the set of maximal cliques that have common nodes with c. One can informally interpret message passing as that the marginal distribution of c is determined by summing over all the admissible label assignments for the nodes of each $c' \in N^c$, and for each c', its marginal distribution in turn relies on all the admissible label assignments for all of its neighbour maximal cliques.

Constraint on Valid Repair Transform. One important characteristic of the repair transform prediction problem is that the admissible repair transforms assigned to a certain node n are highly dependent of the label of itself and the labels of its neighbour nodes. For instance, for our defined 16 repair transforms, a transform t on the virtual root node of a statement is valid only when $t \in \{\texttt{Wrap}-\texttt{IF}-\texttt{N},\ \texttt{Wrap}-\texttt{IF}-\texttt{O},\ \texttt{Wrap}-\texttt{IFELSE}-\texttt{N},\ \texttt{Wrap}-\texttt{IFELSE}-\texttt{O}, \texttt{Wrap}-\texttt{TRY}\}$. To take this into account, we need to establish constraints to determine whether the joint assignment ($\mathbf{t}$, $\mathbf{n}$) for a certain clique is admissible.

Constraints restrict the sets of admissible label assignments for cliques and the message passing algorithm can be easily modified to account the constraint: only admissible labeling assignments are considered in the messages. Constraints can be established according to the domain knowledge about what kinds of repair transforms are possible for certain nodes in a clique. We in this paper define constraints based on the training dataset $D = \{\langle \mathbf{t}^i, \mathbf{n}^i\rangle\}_{i=1}^m$. The idea is that the training dataset D is representative enough, and for an assignment $\mathbf{t}^k$ to be admissible, we should have observed the occurrence of this in the training data. Using the notations in Section 5.2.2, for different types of cliques, we define the set of admissible repair transforms for different node labels as follows:

$$\textit{adm\_nodetran (D)} = \{(L, T^S)|\exists L \in \gamma, \forall T \in T^S, \mathbf{1}_{(T,L)} \in \textit{all\_nodetran}(D)\}$$

$$\textit{adm\_edgetran (D)} = \{(L_1, L_2, T^P)|\exists L_1, L_2 \in \gamma, \forall \langle T_1, T_2\rangle \in T^P, \mathbf{1}_{(T_1,L_1,T_2,L_2)\in \textit{all\_edgetran}(D)}\}$$

$$\textit{adm\_triangletran (D)} = \{(L_1, L_2, L_3, T^T)|\exists L_1, L_2, L_3 \in \gamma, \forall \langle T_1, T_2, T_3\rangle \in T^T, \mathbf{1}_{(T_1,L_1,T_2,L_2,T_3,L_3)\in \textit{all\_triangletran}(D)}\}$$

where γ represents the set of possible labels for a node, T^S, T^P, and T^T are the sets whose elements are 1-tuple, 2-tuple, and 3-tuple respectively.

Based on the established admissible repair transforms for different node labels, we can then determine whether the assignments for different cliques violate the constraint as follows:

$$V(t_1, n_1) = \mathbf{1}_{\exists(L,T^S)\in \textit{adm\_nodetran}(D): l(n_1)=L \wedge t_1 \notin T^S}$$

$$V(t_1, n_1, t_2, n_2) = \mathbf{1}_{\exists(L_1,L_2,T^P)\in \textit{adm\_edgetran}(D): l(n_1)=L_1 \wedge l(n_2)=L_2 \wedge (t_1,t_2)\notin T^P}$$

$$V(t_1, n_1, t_2, n_2, t_3, n_3) = \mathbf{1}_{\exists(L_1,L_2,L_3,T^T)\in \textit{adm\_triangletran}(D): l(n_1)=L_1 \wedge l(n_2)=L_2 \wedge l(n_3)=L_3 \wedge (t_1,t_2,t_3)\notin T^T}$$

where the value 1 means the constraint is violated and the assignment is not admissible.

The use of constraint arises for two reasons. First, it can significantly reduce the time complexity in inference as only admissible transform assignments are considered in the messages. Second, it allows to eliminate incorrect transform

assignments according to domain knowledge, resulting in accuracy improvement.

6 Experimental Evaluation

We present in this section the implementation details, the experimental methodology, and the evaluation results.

6.1 Implementation.

We implemented our repair transform prediction approach in a tool called `Seer`. The tool is written in Java and currently works for Java code. It consists of two parts: repair transform extraction and CRF model learning and prediction. For repair transform extraction, we use GumTree [37] to extract the AST tree edit script. GumTree is an off-the-shelf, state-of-the-art tree differencing tool that computes AST-level program modifications, and outputs them as the 4 basic tree edit operations: UPD, ADD, DEL, and MOV. We also use Spoon [51] to analyze the code that surrounds the AST nodes affected by tree edit operations. Besides, PPD [44] is employed to facilitate the detection of certain repair transforms. Our CRF model is implemented on top of the XCRF library [52], which is a framework for building CRFs to label XML data. We extend XCRF library to incorporate our specific feature functions, learning algorithm and prediction algorithm. In particular, a major modification both at the conceptual and implementation level is the support for computing the *top-k* predictions, i.e., the predictions with the *k* highest conditional probabilities.

6.2 Experimental Setup.

Dataset. We use the Boa dataset [20], [21] as the source to get the required dataset with repair transforms on AST nodes. Boa is a domain-specific language and infrastructure for analyzing ultra-large-scale software repositories and the Boa dataset includes 4,590,679 bug fixing commits. To compute the diffs of the bug-fixing commits with GumTree, we separately compute the diff between each file affected by a commit (i.e., a patched file) and its previous version (i.e., a buggy file). The output of GumTree is an *edit script* composed by *tree edit operations*. We find that when the diff is relatively large, the edit scripts are frequently not accurate enough to reflect real code changes. In addition, a bug-fixing commit with large diffs is much more likely to include irrelevant changes such as feature additions or refactorings [53]. Therefore, we limit our repair transform extraction to those bug-fixing commits with relatively small diffs. We experimented with different thresholds for root tree edit operations and find when the threshold is set to 10, the GumTree outputs are accurate enough to reflect real code changes in most cases and we can correctly attach repair transforms to AST nodes. After setting the threshold to be 10, we find that the tree edit operations in a certain file are typically targeting a single statement. Consequently, we use the AST of the targeted statement as the AST of the buggy code. In case occasionally the tree edit operations in a file involve multiple statements, we view each of these statements as an isolated bug. Finally, we have 267,555 pairs of $\langle \mathbf{t}, \mathbf{n} \rangle$ extracted from the original bug fixing commits, where $\mathbf{n}$ is the AST of a changed statement and $\mathbf{t}$ is the set of repair transforms associated with the nodes of $\mathbf{n}$. Note that our approach itself is applicable to ASTs of any code snippet.

Table 6 shows the number for different repair transforms we have in the dataset. The "Single" (resp. "Multiple") refers to the case where the number of actual repair transforms for a certain $\langle \mathbf{t}, \mathbf{n} \rangle$ is one (resp. larger than one). Note that a single repair transform may involve multiple root tree edit operations according to our definitions in Table 1 and Table 2, thus the "Single" case does not necessarily mean that the corresponding bug-fixing commit just contains one root tree edit operation. We can see from Table 6 that a majority of the data involves just one actual repair transform applied to a certain node. To check the quality of the dataset, we randomly sample 200 examples for each repair transform and also 200 examples for "multiple-transform" case, and manually check whether correct repair transforms are attached to correct AST nodes. More specifically, for the AST $\mathbf{n}$ of a changed statement, we carefully examine the bug-fixing commit and determine what kinds of repair transforms are associated with the nodes of $\mathbf{n}$, i.e., we manually establish the ground-truth repair transforms $\mathbf{t}^{real}$ associated with $\mathbf{n}$. We compare the automatically extracted $\langle \mathbf{t}, \mathbf{n} \rangle$ with the manually established $\langle \mathbf{t}^{real}, \mathbf{n} \rangle$ and if they are exactly the same (i.e., each AST node is associated with exactly the same repair transform, including EMPTY which means no repair transform), we view correct repair transforms have been attached to the correct AST nodes. The result of manual check is shown in the column *#Correct*. Overall, we can see that the accuracy is satisfactory, achieving 88% correctness rate for "multiple-transform" case and at least 91% correctness rate for "single-transform" case.

Cross-validation. We use cross-validation to select parameters of the CRF model and evaluate the performance of the trained CRF model. We split the whole dataset into 10 equal folds, with each fold having the same number of "single-transform" instances for each kind of the 16 considered repair transforms and same number of "multiple-transform" instances. We select one fold as test dataset to see the model performance.

The other 9 folds in the dataset are used as the training dataset and we use cross-validation to investigate the impact of the parameters and select them accordingly. There are three parameters involved in our CRF model: the regularization parameter δ^2, the parameter q that indicates the magnitude of the distribution-aware prior, and finally the L-BFGS iteration parameter G that specifies the number of gradient computations used by the optimization procedure. Higher values of δ^2 will incur larger penalty on large weights associated with feature functions and higher values of G will result in higher execution cost for inference. We evaluate the error rate on each fold by training a model on the other 8 folds. The error rate is established using the *top-3* evaluation metric described later. We repeat this process by using a set of different parameters and aim to identify parameters with the lowest error rate. The procedure determines that 500 and 200 are good values for δ^2 and G respectively, and the following results are based on these two values. The parameter q significantly impacts the prediction result for "single-transform" and "multiple-

TABLE 6: Descriptive statistics of the repair transforms in the dataset.

Repair Transform	#Number	#Correct	Repair Transform	#Number	#Correct
Wrap − IFELSE − N	4,528	90%	LogExp − Exp	21,331	98%
Wrap − IFELSE − O	11,366	92%	LogExp − Red	3,237	97%
Unwrap − Meth	1,658	92%	Wrap − IF − N	10,307	93%
Constant − Rep	84,248	96%	Wrap − IF − O	11,653	95%
Meth − RW − Var	4,084	91%	Wrap − Meth	6,403	95%
Meth − RW − Meth	43,314	92%	Var − RW − Var	29,322	96%
BinOper − Rep	5,928	97%	Unwrap − IF	6,936	96%
Var − RW − Meth	3,694	93%	Wrap − TRY	6,662	96%
Single	254,671	—	Multiple	12,884	88%

transform" instances, and we will discuss its impact in detail in the result section.

Evaluation Metric. For each tuple $\langle \mathbf{t}, \mathbf{n} \rangle$ in the test dataset, we view a prediction $\langle \mathbf{t}', \mathbf{n} \rangle$ by our trained CRF model as correct if $\mathbf{t} = \mathbf{t}'$. That is, for each node $n \in \mathbf{n}$, the predicted repair transform associated to it is exactly the same as the ground truth repair transform extracted from data. We use both metrics *top-1* and *top-3* to evaluate the model performance. For metric *top-1*, the prediction is considered as a success only if the top 1 prediction (i.e., the prediction with the highest probability) is correct. For metric *top-3*, we consider the prediction as a success if at least one of the top 3 predictions (i.e., the three predictions with the three highest probabilities) is correct. We respectively calculate the number of tuples in the test dataset that have been successfully predicted for these two metrics, and then respectively divide the two numbers by the total number of tuples in the test dataset to see model accuracy. These two metrics have been widely used to evaluate model performance, in particular for hard learning task such as ours [15], [22], [54].

Baseline. *History probability baseline:* We first establish a baseline which assigns repair transforms to nodes according to basic statistics on past commits. Using the same dataset employed for training the model, we establish a set of tuples <*L*, *T*, *P*> where *L* is a certain node label, *T* is a certain repair transform, and *P* represents the probability of assigning *T* to nodes with label *L*. Basically, let N^0 denote the number of nodes with label *L* in the training dataset and N^1 denote the number of nodes with label *L* and are associated with transform *T*, then $P = N^1/N^0$. The baseline first assumes there needs only one repair transform to correct the program and prioritizes the node *n* and repair transform *T* on it according to the established probability for <*l*(*n*), *T*>. When there are several nodes of the same label, it breaks ties according to their orders during pre-order traversal. After all possible single node transforms have been explored, the baseline gradually explores all possible two node transforms and so on.

NMT baseline: We also build another baseline based on neural machine translation (NMT) [13], in particular the sequence to sequence (Seq2Seq) translation model [15]. To establish this baseline, we first use pre-order to traverse the AST for a certain code snippet and get the linearized AST as a sequence. Then, we use one encoder for the AST node labels and one encoder for the AST node values. The two encoder outputs are combined and fed into the decoder which outputs a sequence of code transforms applied to each AST node accordingly. To train and evaluate the NMT baseline, we use the same training and test dataset as we used to build the CRF model. The NMT baseline is built on top of OpenNMT [55].

Partial Model and Partial Data Comparison. To see the impact of feature functions, we also compare our "Full Model" against a "Partial Model" that uses only observation-based or indicator-based feature functions. Besides, we also study the impact of training data size on model performance by keeping test data the same but only using 30% and 70% of the training data to build the model.

6.3 Experimental Results

Our training is run on a cluster node with 24 cores and 512 GB RAM, and the system is Ubuntu 16.04. For the full model, the training takes around 41 hours for each value of parameter q when 500 and 200 are set as the values for δ^2 and G respectively, and the average prediction time is 48ms.

Table 7 and Table 8 respectively show the performance of the model when *top-1* and *top-3* are used as the evaluation metric. The numbers in the cells of the tables represent the prediction accuracy for each kind of transform in "single-transform" category and also the accuracy for "multiple-transform" category (at the table bottom). The columns "History" and "NMT" represent the results obtained by history probability baseline and NMT baseline respectively. The "CRF Full Model" refers to the model that uses both observation-based and indicator-based feature functions, and regardless of the used evaluation metric, the overall optimal model is obtained when the distribution-aware prior coefficient q is set to be 0.5 (highlighted with gray). The "Partial Model" refers to the model that uses only observation-based feature functions (denoted as 'O') or only uses indicator-based feature functions (denoted as 'I'), and the "Partial Data" refers to the model performance when a subset of the original training data is used to build the model. For space reason, we only show the result obtained with the overall optimal value of q (that is 0.5) for partial models and models obtained with partial training data.

Comparison with Baselines. First, we can see that our optimal model overall performs much better than the history probability baseline. The history probability baseline provides some reasonable prediction accuracy only for a few repair transforms (including LogExp − Exp, Constant − Rep, Meth − RW − Meth, Unwrap − IF, and Wrap − IF − O) that are most widely used, and performs poorly for "multiple-transform" prediction as it needs to firstly explore all possible single node transforms. Our model, however, not only keeps high prediction accuracy for those few widely used transforms, but also provides good prediction accuracy for those comparably less used transforms and "multiple-transform" instances.

TABLE 7: The accuracy for AST-level repair transform prediction using Top-1 evaluation metric

Repair Transform	Transform Prediction Result								
	Baselines		CRF Full Model			Partial Model		Partial Data	
	History	NMT	q=0.0	q=0.5	q=1.0	O	I	30%	70%
Wrap − Meth	0.0	0.06	0.06	0.06	0.03	0.01	0.03	0.05	0.06
Unwrap − Meth	0.0	0.07	0.08	0.08	0.06	0.03	0.01	0.07	0.07
Var − RW − Var	0.09	0.29	0.31	0.30	0.27	0.13	0.27	0.28	0.29
Var − RW − Meth	0.0	0.10	0.13	0.15	0.11	0.00	0.16	0.15	0.14
Meth − RW − Var	0.0	0.06	0.06	0.06	0.03	0.02	0.04	0.04	0.05
Meth − RW − Meth	0.52	0.48	0.57	0.58	0.53	0.25	0.53	0.54	0.55
BinOper − Rep	0.01	0.22	0.24	0.23	0.15	0.01	0.09	0.22	0.22
Constant − Rep	0.77	0.68	0.76	0.76	0.67	0.51	0.75	0.75	0.75
LogExp − Exp	0.47	0.59	0.93	0.92	0.72	0.38	0.91	0.89	0.90
LogExp − Red	0.0	0.09	0.10	0.10	0.11	0.02	0.08	0.08	0.09
Wrap − IF − N	0.0	0.26	0.28	0.27	0.12	0.13	0.09	0.23	0.25
Wrap − IF − O	0.05	0.14	0.15	0.15	0.08	0.11	0.09	0.13	0.15
Wrap − IFELSE − N	0.0	0.05	0.05	0.05	0.03	0.02	0.00	0.04	0.05
Wrap − IFELSE − O	0.0	0.04	0.05	0.05	0.02	0.02	0.01	0.03	0.04
Unwrap − IF	0.01	0.18	0.21	0.20	0.13	0.83	0.18	0.16	0.19
Wrap − TRY	0.0	0.11	0.13	0.12	0.05	0.04	0.00	0.10	0.11
Average	0.12	0.21	0.26	0.26	0.19	0.15	0.20	0.23	0.24
Mul − Transform	0.0	0.15	0.11	0.19	0.28	0.04	0.15	0.16	0.17

TABLE 8: The accuracy for AST-level repair transform prediction using top-3 evaluation metric

Repair Transform	Transform Prediction Result								
	Baselines		CRF Full Model			Partial Model		Partial Data	
	History	NMT	q=0.0	q=0.5	q=1.0	O	I	30%	70%
Wrap − Meth	0.0	0.24	0.29	0.28	0.23	0.06	0.23	0.23	0.26
Unwrap − Meth	0.0	0.15	0.23	0.26	0.17	0.05	0.04	0.16	0.21
Var − RW − Var	0.32	0.52	0.58	0.60	0.57	0.31	0.50	0.53	0.56
Var − RW − Meth	0.0	0.30	0.41	0.44	0.38	0.15	0.34	0.36	0.41
Meth − RW − Var	0.0	0.26	0.30	0.30	0.23	0.04	0.20	0.24	0.27
Meth − RW − Meth	0.89	0.77	0.84	0.83	0.79	0.64	0.75	0.79	0.80
BinOper − Rep	0.14	0.55	0.65	0.65	0.59	0.05	0.60	0.61	0.63
Constant − Rep	0.93	0.88	0.90	0.89	0.87	0.78	0.80	0.86	0.88
LogExp − Exp	0.96	0.89	0.98	0.98	0.91	0.78	0.93	0.92	0.95
LogExp − Red	0.0	0.74	0.85	0.81	0.58	0.05	0.70	0.71	0.76
Wrap − IF − N	0.19	0.52	0.59	0.56	0.39	0.43	0.31	0.50	0.55
Wrap − IF − O	0.59	0.48	0.58	0.54	0.32	0.46	0.42	0.46	0.52
Wrap − IFELSE − N	0.0	0.09	0.14	0.14	0.08	0.07	0.05	0.09	0.12
Wrap − IFELSE − O	0.01	0.19	0.21	0.19	0.13	0.10	0.04	0.16	0.19
Unwrap − IF	0.93	0.75	0.85	0.81	0.54	0.93	0.68	0.79	0.82
Wrap − TRY	0.0	0.24	0.26	0.26	0.16	0.13	0.01	0.22	0.24
Average	0.31	0.47	0.54	0.53	0.43	0.32	0.41	0.47	0.51
Mul − Transform	0.0	0.36	0.21	0.41	0.48	0.16	0.33	0.37	0.39

Second, we can see that our optimal model performs consistently better than the NMT baseline for each of the 16 considered repair transforms. When *top-1* is used as the evaluation metric, the NMT baseline on average achieves 21% and 15% prediction accuracy for "single-transform" and "multiple-transform" instances respectively, our optimal model instead averagely achieves 26% and 19% prediction accuracy for "single-transform" and "multiple-transform" instances respectively. When *top-3* is used as the evaluation metric, the NMT baseline on average achieves 47% and 36% prediction accuracy for "single-transform" and "multiple-transform" instances respectively, while our optimal model achieves on average 53% and 41% prediction accuracy for "single-transform" and "multiple-transform" instances respectively. Recall that compared with NMT baseline which takes a flattened sequence of tokens as inputs, the main difference of Seer is to explicitly take the code structure into account. Since Seer performs better than the NMT baseline, this shows that taking the code structure into consideration is beneficial for predicting AST-level code transform.

Impact of Distribution-aware Prior. The distribution-aware prior is introduced to make "single-transform" and "multiple-transform" training examples have a balanced impact in the learning process. The results in both Table 7 and Table 8 show that the use of distribution-aware prior indeed improves the model performance on "multiple-transform" instances. For instance, by using the magnitude q = 1.0, the model performance for "multiple-transform" instances has gone from 11% (when q = 0.0) to 28% when *top-1* is used as the evaluation metric and has gone from 21% (when q = 0.0) to 48% when *top-3* is used as the evaluation metric. Note, however, that when the magnitude of q is too large, it brings an obvious performance degradation for "single-transform" instances. In this case, the "multiple-transform" instances instead have become the dominating class in the training data. Considering the accuracy for both "single-transform" and "multiple-transform" instances, *setting the magnitude q to 0.5 brings overall good performance on our dataset*.

Impact of Feature Function. Comparing the performance of the full model with that of the partial model, we can see that the performance has obviously degraded when only one kind of feature functions is used. Overall, the observation-based and indicator-based feature functions respectively perform well for certain repair transforms, and complement each other to form a strong model. In particular, the partial model which only uses observation-based feature functions performs better for repair transforms that target the whole statement (e.g. repair transforms `Wrap − TRY` and `Unwrap − IF`) than the partial model which only uses indicator-based feature functions, while the latter instead performs better for other repair transforms. To sum up, *considering both observation-based and indicator-based feature functions in conjunction is essential to get good performance.*

Impact of Training Data Size. Comparing the performance of the model trained using the whole data with that of the model trained using partial data, we can see that the prediction accuracy of the model has improved when more training data are used. When 100%, 70%, and 30% of the whole training dataset is used to train the model and *top-1* is used as the evaluation metric, the established CRF model on average achieves 26%, 24%, and 23% prediction accuracy for "single-transform" instances respectively, and averagely achieves 19%, 17%, and 16% prediction accuracy for "multiple-transform" instances respectively. When 100%, 70%, and 30% of the whole training dataset is used to train the model and *top-3* is used as the evaluation metric, the established CRF model on average achieves 53%, 51%, and 47% prediction accuracy for "single-transform" instances respectively, and averagely achieves 41%, 39%, and 37% prediction accuracy for "multiple-transform" instances respectively. Overall, *the amount of training data is critical to model performance.*

Summary. In summary, this set of large scale experiments has shown that 1) our model performs well for vastly different repair transforms, and and works well for both "single-transform" and "multiple-transform" code change scenarios; 2) our model performs consistently better than all baselines, including the strong NMT baseline, suggesting the importance of considering code structure in achieving good prediction accuracy for code transform; 3) the distribution-aware prior coefficient q is an important configuration parameter in the model, and our experiments show that $q = 0.5$ is a good value; 4) the use of both observation-based and indicator-based feature functions, i.e. the mix of automatically extracted structural features and carefully engineered code analysis features, is key to obtain good performance; 5) the size of training data significantly impacts model performance and more training data will be beneficial.

6.4 Case Studies

We now build on seven examples to illustrate the performance of our established CRF model. Besides the Boa dataset, we also use examples from Defects4J [53] and Bugs.jar [41], two widely used benchmarks in the program repair community. For all these examples, the most likely prediction by our overall best performance CRF model (q = 0.5) contains exactly the ground-truth repair transform(s) needed for correcting each bug. In other words, our best performance CRF model makes perfect prediction for all these seven examples. With regard to the NMT baseline, the top 3 predictions by it for each of these seven examples do not contain the ground-truth repair transform(s) needed for correcting the bug. For history probability baseline, it places the ground-truth repair transform needed for correcting the bug example in Figure 5 (a) or Figure 5 (c) as the second most likely prediction, and the top 3 predictions by it for each of the other five examples do not contain the ground-truth repair transform(s) needed for correcting the bug.

Where to Apply Transform and What Transform to Apply? The three examples in Figure 5 (a), Figure 5 (b), and Figure 5 (c) respectively require a single repair transform to a constant, variable access, and method call. There are a wide variety of program elements in the input code snippets for all these three examples that can possibly be subject to our defined repair transforms (i.e., the defined repair transforms can possibly apply to multiple program elements), and in particular there are respectively more than one constant, variable access, and method call in the input code snippets for Figure 5 (a), Figure 5 (b), and Figure 5 (c), but our established CRF model can accurately predict which specific program element should be subject to repair transforms. In addition, the program elements constant, variable access, and method call all can be subject to various repair transforms defined in this paper, our established CRF model can precisely predict the specific repair transforms needed. Overall, our established CRF model is capable of exactly predicting both "where to apply code transform" and "what code transform to apply".

Co-Transform The four examples in Figure 6 all require multiple repair transforms to be simultaneously applied. For the two examples in Figure 6 (a) and Figure 6 (b), there is a need for multiple repair transforms to instances of the same program element and our established CRF model can accurately predict this kind of co-transform. In particular, note that there is a need for 8 `Var − RW − Var` transforms to the variable access "nativeres" in Figure 6 (b), our established CRF model is able to establish this fact. For the example in Figure 6 (c), the input code snippet contains multiple instances of the same program element (including variable access "paint" and variable access "stroke") and it requires repair transforms to some specific but not all instances of a certain program element: our established CRF model is capable of precisely predicting this kind of co-transform. For the example in Figure 6 (d), multiple radically different types of repair transforms are needed to be applied to different kinds of program elements: our established CRF model can also accurately predict this kind of co-transform. In short, our established CRF model can recognize the necessity of applying multiple repair transforms to repair a certain bug and can accurately predict different categories of co-transforms.

6.5 Usefulness of the Predicted Repair Transforms

To achieve reasonable accuracy and scalability in automatically guiding code changes, we in this paper abstract low-level AST edits into high-level code transforms and separate code change generation into high-level code transform prediction and high-level code transform concretization phases.

```
-  colq.set(currentDocID.getBytes(), zeroIndex + 1, currentDocID.getLength() - zeroIndex -
     2);
+  colq.set(currentDocID.getBytes(), zeroIndex + 1, currentDocID.getLength() - zeroIndex -
     1);
```

(a) cloudera/accumulo (2f0643a9)---from Bugs.jar

```
-  final double gammaAbs = SQRT_TWO_PI / x *FastMath.pow(y, absX + 0.5) *FastMath.exp(-y)
     * lanczos(absX);
+  final double gammaAbs = SQRT_TWO_PI / absX *FastMath.pow(y, absX + 0.5) *FastMath.exp(-
     y) * lanczos(absX);
```

(b) apache/commons-math (9e0c5ad4)---from Bugs.jar

```
-  tTableRow.add(getDataRatekBStr(tCountAnnounceCoord + tReports + tShares, tPeriod));
+  tTableRow.add(getDataRateStr(tCountAnnounceCoord + tReports + tShares, tPeriod));
```

(c) CS-TU-Ilmenau /fog (0a9452a5)---from Boa dataset

Fig. 5: Examples to illustrate the ability of the CRF model in predicting "single-transform" instances.

```
-  int[] values = chrono.get(zeroInstance, chrono.set(start, 0L), chrono.set(end, 0L));
+  int[] values = chrono.get(zeroInstance, chrono.set(start, START_1972), chrono.set(end,
     START_1972));
```

(a) JodaOrg/joda-time (d122b034)---from Defects4J (Time-10)

```
-  c.setRGBA(accum_red / (nativeres*nativeres), accum_green / (nativeres*nativeres),
     accum_blue / (nativeres*nativeres), accum_alpha / (nativeres*nativeres));
+  c.setRGBA(accum_red / (res*res), accum_green / (res*res), accum_blue / (res*res),
     accum_alpha / (res*res));
```

(b) webbukkit/dynmap (06b7a5dd)---from Boa dataset

```
-  super(paint, stroke, paint, stroke, alpha);
+  super(paint, stroke, outlinePaint, outlineStroke, alpha);
```

(c) jfree/jfreechart (c2efee16)---from Defects4J (Chart-20)

```
+ try {
-   while (scanner.hasNextLine())
+   while (scanner.hasNextLine() && !hasExited()) {
      execCommand(scanner.nextLine(), true, isVerbose());
+   }
+ } finally {
+       scanner.close();
+ }
```

(d) cloudera/accumulo (ef0f6ddc)---from Bugs.jar

Fig. 6: Examples to illustrate the ability of the CRF model in predicting "multiple-transform" instances.

The accurate prediction of high-level code transform is the essential prerequisite for the concretization phase, and is the major focus of our work in this paper. We propose a structured prediction approach based on CRF in order to accurately predict the high-level code transforms, and the approach specifically takes two types of code features into account. Our systematic evaluation has shown that the established CRF model consistently performs better than several strong baselines for the repair transform case. For illustrating the high-level code transform concretization phase and demonstrating the usefulness of the predicted high-level repair transforms, an additional evaluation on the benchmark *Defects4J v1.2* [53] is performed.

This version of Defects4J consists of 395 bugs from 6 projects (*i.e.*, Chart, Closure, Lang, Math, Time, and Mockito), and has been extensively used for evaluating the effectiveness of various automatic program repair (APR) techniques [5], [7], [23]–[25], [27], [41], [56]. The specific number of bugs for each of the 6 projects is shown in column "Bugs (ori)" of Table 9. As our model currently targets the 16 repair transforms described in Section 5.1 and is trained using bug-fixing changes that target a single statement, we manually check the 395 Defects4J bugs to identify ones whose repair changes center around one statement and fall into the targeted 16 repair transforms. Finally, we find that 89 Defects4J bugs satisfy the requirement, and the column "Bugs (sel)" of Table 9 gives the qualified number of bugs for each project.

For these selected 89 Defects4J bugs, we use the repair transform extraction part of our tool `Seer` to attach repair transforms to AST nodes of the targeted statement, and we also manually verify that correct repair transforms indeed have been attached to correct AST nodes. We then use our optimal model (i.e., full model with $q = 0.5$) to predict repair transforms on the AST nodes, and compare the predicted repair transforms on AST nodes with the extracted repair transforms on AST nodes. The two columns "#(top-1)" and "#(top-3)" of Table 9 respectively show the number of bugs that can be successfully predicted by our model for each project when *top-1* and *top-3* are used as the evaluation metrics. Overall, the prediction accuracy for the 89 Defects4J bugs is 42.69% and 53.93% respectively for the evaluation

TABLE 9: The prediction performance of our optimal model for the selected 89 Defects4J bugs

Subjects	#Bugs (ori)	#Bugs (sel)	#(top-1)	#(top-3)
Chart	26	10	4	6
Closure	133	34	20	23
Lang	65	12	3	4
Math	106	21	7	10
Time	27	4	1	2
Mockito	38	8	3	3
Total	395	89	38	48

metrics *top-1* and *top-3*. The result again shows that our model achieves good accuracy in predicting AST-level code transform by taking code structure into account.

We then implement a proof-of-concept synthesizer to illustrate the usefulness of the predicted AST-level repair transforms in producing the final patches. Among the 16 targeted repair transforms, we concretize 8 repair transforms to generate the final patch, including `Unwrap − Meth`, `Var − RW − Var`, `Var − RW − Meth`, `Meth − RW − Meth`, `Meth − RW − Var`, `BinOper − Rep`, `Constant − Rep`, and `LogExp − Red`. For the other 8 repair transforms, more complex code snippet synthesis algorithms are needed to concretize them into patches and we leave this to future work. In particular, the considered 8 repair transforms are concretized as follows:

- For repair transform `Unwrap − Meth`, we delete the outer method call and keep the expression(s) wrapped inside the method call. If the deleted method call has several different expressions as its arguments, we separately keep each expression in ascending order of the argument index before a plausible patch is found.
- For repair transform `Var − RW − Var`, we replace the variable v (attached with `Var − RW − Var`) with a certain searched variable. The candidate variables are the set of accessible variables at the buggy statement whose types are same as that of v, and they will be tried in descending order of the similarity with v. The similarity is established for the identifier names of variables based on the Levenshtein distance.
- For repair transform `Var − RW − Meth`, we replace the variable v (attached with `Var − RW − Meth`) with a certain searched method call whose return type is same as the type of v. Note that whenever we refer to method search, the details about the process will be explained later.
- For repair transform `Meth − RW − Meth`, we first replace the method call m (attached with `Meth − RW − Meth`) with another searched method call which accepts exactly the same arguments as m. The same here specifically refers to same in number and type of arguments. All of the candidate method calls are tried in descending order of the similarity with m, and the similarity again is established for the identifier names of method calls based on the Levenshtein distance. If this first procedure does not result in a plausible patch, we then search the file (where the buggy statement resides in) for overloaded methods of m and delete or add certain arguments for m accordingly. These two procedures correspond to the two definitions for repair transform `Meth − RW − Meth` in Table 1.
- For repair transform `Meth − RW − Var`, we replace the method call m (attached with `Meth − RW − Var`) with a certain variable whose type is same as the return type of m. The candidate variables are those that are accessible at the faulty statement, and these variables are tried in ascending order of their occurrence in the file where the faulty statement resides in.
- For repair transform `BinOper − Rep`, we replace the binary operator *bo* (attached with `BinOper − Rep`) with another binary operator. We divide binary operators into 5 kinds: logical (||, &&), bit (|, ^, &), comparison (==, !=, <, >, <=, >=), shift («, »), and arithmetic (+, -, *, /, %). We only replace *bo* with another binary operator whose kind is the same as that of *bo*.
- For repair transform `Constant − Rep`, we replace the value of the literal (attached with `Constant − Rep`) with another literal value of the same kind. We currently consider three kinds of literal value: numerical, boolean, and string. We only replace literal value of these three kinds, and the specific literal values used for replacing the old literal value are searched in the file where the faulty statement resides in.
- For repair transform `LogExp − Red`, we remove the condition(s) of the logical expression until a plausible patch is found. We first try to see whether it is possible to achieve this by removing one condition, then two conditions, and so on up to all conditions.

A number of points deserve comment. First, when we search for methods to be called, the candidate methods include those methods that have been called and those defined but not called methods in the file where the faulty statement resides in. All of the searched method calls are tried in ascending order of their occurrence in the file unless otherwise specified, and if a certain method call involves additional argument(s), we use accessible variables at the faulty statement whose type is compatible with the desired type of each argument to fill in. Second, if there are multiple predicted repair transforms attached to different AST nodes of the buggy statement and the predicted repair transforms are among the 8 repair transforms for which we give the implementation to concretize them into patches, we combinatorially consider the concretization procedure for each involved repair transform. Third, our predicted repair transforms are attached to specific AST nodes, and this can significantly facilitate the synthesis process as the search space will be dramatically reduced. Finally, our synthesizer is by no means complete for the considered 8 repair transforms and more advanced synthesis procedure can generate more diverse and complex patches. Instead, our synthesizer serves to illustrate that our model can precisely predict high-level code transforms attached to AST nodes and these high-level code transforms can effectively guide the concrete code transform process.

In line with existing work on automatic program repair [28], [56]–[58], our proof-of-concept synthesizer considers a patch as correct if it passes the test suite associated with each project. Once the synthesizer gets a plausible patch, it stops the patch synthesis process. We apply the synthesizer solely to concretize the top 1 prediction (of needed AST-

TABLE 10: The comparison of our proof-of-concept synthesizer with five DL-based APR systems

Compared APR System	Bugs Repaired Only by Our Proof-of-concept Synthesizer
CODIT [23]	Chart-1, Chart-20, Closure-10, Closure-14, Closure-31, , Closure-70, Closure-123, Math-5, Math-58, Math-75, Math-85, Mockito-26, Time-19
CoCoNut [24]	Chart-8, Chart-20, Closure-10, Closure-14, Closure-123, Math-5, Math-70, Math-75, Math-85, Mockito-26
DLFix [25]	Chart-8, Closure-31, Closure-123, Mockito-26
CURE [26]	Closure-14, Closure-31, Closure-123, Math-5, Math-85, Mockito-26
Recoder [27]	Closure-123, Mockito-26

level repair transforms) by our model into patches. If the repair transform(s) involved in the top 1 prediction fall(s) into the 8 repair transforms that the synthesizer has considered, the synthesizer proceeds and tries to come up with a patch, otherwise it directly returns "patch synthesis failure" result. As the synthesizer currently does not concretize all the 16 considered repair transforms into patches, this is a reasonable evaluation setting. Note that if we consider concretizing top n (n>1) predictions and directly ignore the prediction that involves repair transforms for which we currently do not have an implementation to concretize them, over-optimistic results maybe obtained as an actual implementation for these repair transforms may return a plausible patch and prevent the synthesis process for other predictions with lower probability. Among the 38 Defects4J bugs for which our model gives the perfect prediction (i.e., the 38 bugs shown in column "#(top-1)" of Table 9), the needed repair transforms for 21 bugs fall into the 8 repair transforms that the synthesizer has considered, and the remaining 17 bugs involve the other 8 repair transforms.

We apply our synthesizer to the 21 bugs, and the synthesizer successfully returns plausible patches for 16 bugs. For the other 5 bugs, more advanced synthesis procedure will be needed. As the test suite is in general incomplete, the generated patch by the synthesizer can just over-fit to the test suite and be plausibly correct [28], [59], [60]. We compare the synthesized plausible patch with the human patch, and find that the synthesized patches for the 16 bugs are same as the human patches. As overfitting is a fundamental issue for program repair techniques, this reflects that by attaching repair transforms to specific AST nodes, our approach has the potential to significantly alleviate the over-fitting issue.

We compare the results achieved by the proof-of-concept synthesizer with existing deep learning based automatic program repair approaches. To enable fair comparison, we need DL-based APR approaches that have been evaluated under the perfect fault localization, i.e., the buggy statement is known. Note that this setting is the preferred way to evaluate APR approaches according to recent work [61], as it enables fair assessment of APR techniques independently of the fault localization approach used. Also, we require that the APR approaches are evaluated using *Defects4J v1.2*. Finally, we select 5 state-of-the-art DL-based APR approaches that satisfy the requirement, including CODIT [23], CoCoNut [24], DLFix [25], CURE [26], and Recoder [27]. According to the original papers, CODIT, CoCoNut, DLFix, and Recoder respectively repair 16, 44, 40, 57, and 67 bugs under the perfect fault localization setting.

```
- primitiveValues.put(double.class, 0);
+ primitiveValues.put(double.class, 0D);
```

Fig. 7: The Mockito-26 bug repaired uniquely by our approach.

Results. We compare the 16 bugs repaired by our proof-of-concept synthesizer with the bugs repaired by these 5 DL-based APR approaches, and find that our synthesizer repairs a significant number of unique bugs as shown in Table 10). Compared with CODIT, CoCoNut, DLFix, CURE, and Recoder, the proof-of-concept synthesizer repairs 13, 10, 4, 6, and 2 unique bugs respectively. The result implies that on top of the AST-level repair transforms predicted by our model, even a proof-of-concept synthesizer is highly complementary to these best-performing DL-based APR approaches. By using a more advanced and complete synthesis procedure for all the 16 considered repair transforms by our model, we could hypothetically concretize other predicted repair transforms into patches and the patch generation process accordingly can take top n (n>1) predictions into account before a plausible patch is found. In this way, many more correct patches hypothetically will be generated.

While existing DL-based APR approaches typically use the encoder-decoder architecture and view the patch as a sequence of tokens [25] or edits [27], our approach for generating patches instead explicitly separates the patch generation process into high-level repair transform prediction and high-level repair transform concretization phases. This is in line with the widely used paradigm adopted by program synthesis techniques [62], [63] which separates code synthesis into sketch generation and sketch completion phases. Given that both tokens and edits are low-level changes to some extent, they thus are not quite learnable across projects. For tokens, it is all-too-common that projects can use project-specific identifiers and literals. While edits are higher-level changes than token changes, predicting the exact order of a sequence of edits and the parameters of each edit is also a daunting task. Instead, high-level repair transforms abstract away the project specific details and capture the essential patterns, and are thus much more learn-able across projects. Paired with an effective synthesis strategy, the high-level repair transforms can be concretized into the final patch. Our proof-of-concept synthesizer shows that using our paradigm, even a quite simple synthesis strategy can repair a significant number of unique bugs. As an example, Figure 7 shows the Mockito-26 bug which is repaired by our synthesizer but not by the 5 considered DL-based APR approaches. To repair the bug, we need to update the constant literal "0" with constant literal "0D". From the perspective of token-level or edit change, this literal update is quite rare in the training data and the DL-based APR approaches do not perform well. Note edit change needs the specific literal "0D" as the parameter. However, from the perspective of high-level repair transform, this kind of constant replacement change (i.e., `Constant` − `Rep`) is much more common and our structured code transform

prediction approach does the perfect prediction. Given the predicted `Constant − Rep` transform, our proof-of-concept synthesizer successfully generates the correct patch.

Summary. Our paradigm for generating patches explicitly splits the patch generation process into high-level repair transform prediction and high-level repair transform concretization phases, and shows clear advantages over existing DL-based APR approaches in case the ground-truth patch involves project-specific low-level (at the level of token or edit) changes that are not learnable across projects. In addition, note that our predicted repair transforms are attached to specific AST nodes, which can guide the patch synthesis in a highly focused manner and the synthesized patches thus suffer less from the overfitting issue.

6.6 Discussion

Evaluation Subjects: Large Diffs over Small Diffs. Our evaluation limits the number of root tree edit operations and targets the repair transforms associated with small diffs. Note, however, that our approach itself is applicable to ASTs of any code snippet, and we limit the evaluation to small diffs as evaluation on large diffs would pose several problems. First, for large diffs, tree differencing tools like GumTree have accuracy issues, i.e., the produced edit scripts are not accurate enough to reflect real code changes. We refer the readers to [64] for detailed analysis and concrete examples for the accuracy issues. When the edit script is inaccurate, the repair transforms cannot be correctly attached to AST nodes. Second, a large bug-fixing commit is likely to include changes such as feature additions or refactorings that are irrelevant to bug fixing [53], and it is notoriously difficult to detect these irrelevant changes [65]. By focusing on small diffs, we significantly increase the chance that the code changes are actually bug-fixing ones. Third, while our defined 16 repair transforms cover the typical repair actions for the five most common repair patterns, some additional repair transforms (e.g., type replacement) may need to be defined in order to fully consider the large bug-fixing diffs, and we consider this as future work.

Performance Diversity over Different Transforms. We can see from the recorded results that despite our approach achieves overall good performance, its performance varies over different repair transforms. When the overall best performance model is considered (q = 0.5) and *top-3* is used as the evaluation metric, while our established CRF model achieves 0.98 accuracy for the `LogExp − Exp` repair transform, the accuracy 0.14 for repair transform `Wrap − IFELSE − N` is relatively low. We discover that the performance of our established CRF model on different repair transforms is typically in line with that of the *history probability baseline*. In other words, the performance of the established CRF model is relatively good (resp. bad) if that of the *history probability baseline* is not too bad (resp. bad). The implication is that our established CRF model is inclined to assign certain "node-transform" pairs with high probabilities if such "node-transform" pairs are comparably frequent in the training data used to build the model. The exception is repair transform `LogExp − Red` for which the history baseline achieves accuracy of 0.0 when *top-3* is used as the evaluation metric, but our overall best performance CRF model still has the accuracy of 0.81 using the same evaluation metric. We believe that the underlying reason is that our established feature functions can precisely capture the actual context of the AST nodes that implies this transform. To improve the performance of our model over all repair transforms, one essential way is to further enrich the current feature functions so that they can better capture the relation between AST node context and repair transform.

Learning Time. The CRF model learning process requires calculating the marginal probability many times and our approach uses the exact inference algorithm. The exact inference algorithm is relatively slow and can require exponential time in the worst case for complex graphs, and that is why the training process takes around 41 hours for the full model. To speed up the training process, in particular when we consider large changes for which the input AST will be much larger, we believe that approximate inference algorithms can be used. For CRF model learning, there are two popular classes of approximate inference algorithms [31]: Monte Carlo algorithms and variational algorithms. Monte Carlo algorithms attempt to approximately produce a sample from the distribution of interest using a stochastic process. By trying to find a simple approximation that most closely matches the intractable marginals of interest, Variational algorithms instead convert the inference problem into an optimization problem. Studying the applicability and performance of alternative inference algorithms is an interesting area of future work.

Suitable Application Scenario. We envision that the usefulness of the predicted code transforms lies more in guiding the automated code evolution in a try-and-error manner. The iterative loop compensates for the fact that the predicted code transforms by our model can not always be perfect. For the instantiated particular kind of code transform—repair transform, in line with most program repair techniques, the predicted repair transforms will be embodied into final patches (through synthesis) using exactly the try-and-error manner, called generate-and-validate in the program repair community. That is, the faulty program is iteratively modified with different changes synthesized according to the predicted repair transforms and then validated with a fitness function (typically test suite evaluation) until a plausible patch is generated. Due to the absence of a perfect fitness function [66], the full correctness of the plausible patch will have to be established through manual check. Thus, despite the potentially compromised prediction accuracy (i.e., imperfect prediction), the try-and-error workflow can reconcile with the imperfection and effectively generate patches as debugging aids.

6.7 Threats to Validity

We leverage the Boa dataset in this study and one potential threat to external validity is whether the results presented in this paper will generalize to other datasets. However, to the best of our knowledge, Boa dataset is one of the largest well established datasets of Java bugs and has been widely used as the experimental subject [21], [67], [68]. Another threat to external validity is that we use 16 repair transforms to evaluate the approach and doubts may arise whether the approach will still be effective for other code

transforms. Note that our approach itself is not limited to specific types of code transform, it only requires code transforms are defined on AST nodes and we give the framework for defining AST-level code transforms. The 16 considered repair transforms serve as an instantiation of the code transform and they cover the typical repair actions for the five most common repair patterns. A final threat to external validity is that our experimental evaluation uses repair transforms extracted from bug-fixing commits with relatively small diffs, and concerns may be raised about whether the dataset extracted this way is representative enough and whether our approach still has advantages over the two baselines when considering large diffs. Note, however, that our approach itself is applicable to ASTs of any code snippet, and we confine the evaluation to small diffs only for the reasons given in Section 6.6. For the identified small diffs, we do not intentionally select specific ones from which to extract the defined repair transforms. In addition, note that our approach takes code structure into account compared to the two baselines, which is essential for achieving good prediction accuracy regardless of the size of the input AST and the number of repair transforms associated with it.

A potential threat to internal validity is that our evaluation needs to attach repair transforms to AST nodes and concerns may be raised about the accuracy of this process. To reduce this threat as much as possible, our evaluation focuses on repair transforms associated with bug-fixing commits with relatively small diffs and the manual check suggests that the achieved accuracy is satisfactory, achieving 88% correctness rate for "multiple-transform" case and at least 91% correctness rate for "single-transform" case. With regard to possible errors in our implementation and experiments, the whole artifact related with this paper is made available online for scrutiny and extension by other researchers. Besides, we also use 10-fold cross validation to objectively assess the performance of our established model.

6.8 Future Directions

To begin with, our evaluation in this paper focuses on repair transforms extracted from bug-fixing commits with relatively small diffs for the reasons given in Section 6.6. One important future direction is investigating the performance of the approach on repair transforms associated with large bug-fixing diffs. To achieve this, we need to improve state-of-the-art AST differencing tools and design accurate and efficient refactoring detection algorithms so that we can accurately attach repair transforms to AST nodes according to our definition. In particular, the "*transform number imbalance issue*" is likely to be more serious for large code changes, and we would like to study whether the proposed distribution aware learning can still effectively alleviate this issue and design possible improvements if needed.

Then, our approach lifts AST edits to high-level code transforms and this abstraction brings both reasonable accuracy and scalability for the arguably hard learning task of predicting code changes. Some high-level code transforms will involve selecting certain code elements to operationalize them and another future direction is designing customized effective synthesis algorithm to finish this process. Our proof-of-concept synthesizer concretizes 8 repair transforms to get the final patches, it is worthwhile to explore more advanced and complete synthesis procedures that account for all the 16 considered repair transforms by our model. There are abundant effective code snippet synthesis algorithms [29], [30], [69]–[71] that we can potentially build upon.

Besides, our evaluation in this paper assumes that we know the faulty code snippet in advance. Thus, it would be interesting to investigate the performance of our approach when it is integrated with fault localization techniques [72]–[74]. In this way, we can more objectively evaluate its effectiveness in industry and overcome potential limitations.

In addition, the realization of our approach currently uses repair transform for fixing bugs as instantiation of code transform, it would be worthwhile to investigate the effectiveness of our approach for other kinds of transforms such as code refactoring and feature improvement.

Furthermore, another important future direction is recognizing additional factors other than program syntax and semantics that can influence code evolution, and integrate them into the probabilistic model. For instance, modern software development typically features the use of integrated development environments (IDEs) and there are in general a broad range of tools within IDEs for helping developers with the code transform task [75]–[77]. It would be valuable to explore the integration of these programming environment factors into our established model. To explore such a direction, one important obstacle we need to overcome is acquiring a great deal of code evolution data associated with programming environment information.

Finally, our evaluation in Section 6.3 shows that the use of both observation-based and indicator-based feature functions are indispensable for obtaining good prediction performance. For observation-based feature functions, we respectively consider various type related, usage related, and syntax related program element characteristics. For indicator-based feature functions, we respectively consider structural features for AST vertices, edges, and triangles. It would be worthwhile to detailly investigate the specific role each of these code features plays in obtaining the good prediction performance, and a feasible way to achieve this is employing the Recursive Feature Elimination (RFE) method [78], [79].

7 Related Work

In this section, we review some work closely related with this paper.

7.1 Big Code

"Big code" approaches [10] for programming refer to data-driven approaches that typically learn form the large amount of freely accessible code, in particular in on-line repository hosting services such as GitHub and BitBucket. The research interest on "Big code" approaches has been rising rapidly in recent years, and we can already see a great many successful applications. Some example applications include defect prediction [80], [81], code completion [70], [82], code synthesis [29], name-based bug detection [83],

deobfuscation [84], inferring cryptographic API rules [85], effort estimation [86], generation of meaningful assert statements for unit tests [87], detection of argument selection bug [88], infer of loop invariants [89], type inference for dynamically typed languages [90], discovery of aliasing specifications of APIs [91], acquisition of strategy for adapting program analysis [92], fixing MOOC programs [14], sketch implementations [93] or learn the relation between code features and sketch implementations [62], and fuzzing smart contracts [94], etc. We refer the readers to [95] for a recent overview of machine learning on code.

In this paper, we target a problem that has not been studied before: the prediction of code transforms at the level of AST nodes. In the next two sections, we respectively review the closely related areas about "code transform learning" and "program representation learning".

7.2 Code Transform Learning

During the life-cycle of a software, its source code will go through a lot of transformations, for reasons such as bug fixing, code refactoring, and feature improvement. To aid in the code transform process, a lot of learning-based techniques have been developed.

Meng et al. [16] propose an approach which learns from a similar but not identical edit instance to create a context-aware edit script. They later further extend the approach to learn from multiple edit instances, enabling automatic identification of the target edit location and generation of more accurate edit script [17]. Rolim et al. [18] use a programming-by-Example (PBE) technique to search for code transform that is consistent with all example transform instances. Long et al. [42] infer code transforms from bug-fixing commits that have fixed the same type of exception bugs. For automating API migration, by learning from client differences in API usage from multiple examples, Andersen et al. [96] create an edit script for the correct API usage and then update the incorrect API use accordingly. Koyuncu et al. [19] make use of an iterative clustering strategy to mine fix patterns from atomic changes within patches. To make programming by demonstration/examples (PBD/PBE) systems more widely adopted in practice, Miltner et al. [97] present a modeless system that can synthesize context-specific repetitive edits on the fly. Bader et al. [98] propose a hierarchical clustering algorithm that summarizes past fix patterns into a general-to-specific hierarchy and then a context-aware ranking technique is used to select the most appropriate fix for a specific bug. The approach targets six bug categories, such as null dereferences and incorrect API calls. When using multiple edit instances, these approaches require either that all the example edits are similar enough or that the example edits are clustered (manually or automatically) using a certain similarity criterion first. Essentially, these approaches try to extract useful edit information from a few highly similar (according to certain definition of similarity) edit instances. However, it can be difficult for few highly similar examples to contain all the information needed for making prediction. In contrast, our presented approach is able to extract and assembly information from diverse (i.e., not that similar) example edits and collectively builds a probabilistic model that can make accurate prediction for general categories of code transform.

There are few recent works that aim to establish probabilistic models of code transforms. Given a partially finished edit, Brody et al. [99] aim to predict the remaining parts of the edit. The technique tries to represent the four basic AST edit operations (UPD, ADD, DEL, MOV) using two nodes from the original AST, and then uses the AST path between the two nodes to represent an edit operation. Unlike our approach which in principle is amenable to all kinds of edit operations, an issue with their technique is that a significant portion of real-life edit operations can not be represented using their representation. Yin et al. [100] propose to use a neural architecture to embed source code edits in a high-dimensional embedding space. One key difference with our work is that their embedding is not actionable. Conversely, our approach abstracts basic tree edit operations into code transforms on AST nodes that are fully actionable and explainable. Besides, unlike our work, the work of Yin et al. [100] typically requires that the edit to be predicted is exactly included in the context. Tufano et al. [13] mine bug fixes from GitHub projects and use the associated AST edit actions to train an Encoder-Decoder model capable of translating buggy code into correct code. However, one limitation is that the translation model can only learn fixes that involve re-arranging keywords, identifiers, and literals already available in the context of the buggy code. Mesbah et al. [15] train a neural network to learn simplified edit script. Unlike our work which targets the wide variety of code errors and the predicted code transforms are associated to specific AST nodes, the work of Mesbah et al. [15] focuses on certain compilation errors and the learned edit script does not contain edit position information. More importantly, compared to all of these works, only our approach abstracts the AST edits into high-level code transforms and this abstraction brings both accuracy and scalability for this hard learning task.

7.3 Program Representation Learning

Within the big code literature, some works specifically target the representation problem. In particular, inspired by the word2vec [101] milestone, which embeds words into a numerical space where similar meaning words are in close proximity, many research efforts have been devoted to neural program representations in recent years.

Alon et al. [102] use a path-based attention model to learn continuous distributed vector representations for code snippet (i.e., code embeddings), which then can further be used to predict semantic properties of the snippet. It is yet not clear how the learned representation can be used for predicting the newly defined AST-level code transforms in this paper. DeFreez et al. [103] use paths in the control flow graph to embed functions, aiming to identify function synonyms. Allamanis et al. [104] explore a graph-based representation of both the syntactic and semantic structure of code, which then is given as input to a graph neural-network. Henkel et al. [105] represent programs with abstractions of traces obtained from symbolic execution. Hellendoorn et al. [106] argue that source code has special properties and improve existing language modeling techniques by incorporating handle of particular aspects of source code, including for example frequent changes and

deeply nested scopes. Ideally, part of the representation is automatically inferred for these works, which is called "learned semantic feature" by Wang et al. [107]. Compared with neural embedding of code snippet, our approach abstracts code edits into code transforms on AST nodes that are fully actionable and explainable. Our representation of programs is a combination of abstract syntax trees (AST) and rich carefully engineered and powerful features, which not only facilitates the learning process but also enables the definition of explainable AST-level code transform.

7.4 Data-driven Program Repair

The past two decades have witnessed an ever-growing research interest on program repair, which aims to automatically correct bugs. Two most well-known categories of program repair techniques are generate-and-validate techniques [108] and synthesis-based techniques [109]. We refer the readers to [110] for an overview of program repair.

In particular, some works in the program repair literature [110] explore data-driven program repair. To investigate the effectiveness and promise of program repair in fixing real bugs, Zhong et al. [111] empirically study more than 9,000 real-world bug fixes and come up with four insights on fault localization [72], [73] and faulty code fix. Le et al. [5] use the commit history to select the most likely mutation-based patch. Long et al. [2] propose an approach to select the patch that resembles the most to past human patches, aiming to get patches that suffer less from overfitting [28]. Wang et al. [6] use data-driven program repair for automated feedback generation. Jiang et al. [112] combine the search space from similar code with search space obtained by abundant existing patches to better guide program repair. Wen et al. [113] use three AST mutation operators (Replacement, Insertion, and Deletion) to make the search space include potentially more correct patches, and use context information of fixing ingredients obtained from real bug fixes in open source projects to explore the search space. Chakraborty et al. [23] propose CODIT, which learns code change patterns by employing a tree-based neural machine translation model to encode source code changes. Lutellier et al. [24] make use of ensemble learning on the combination of convolutional neural networks (CNNs) and neural machine translation (NMT) model to generate patches token by token. Li et al. [25] introduce a two-tier DL model named DLFix, whose first layer aims to learn the contexts of bug fixes and second layer tries to learn the bug-fixing patch by using the result from first layer as an additional weighting input. Jiang et al. [26] propose CURE, which makes use of three strategies to to improve neural machine translation based APR techniques, including pre-trained programming language model, code-aware search, and sub-word tokenization. Zhu et al. [27] propose Recoder, which uses a syntax-guided manner to generate edits and features provider/decider architecture and placeholder generation. In our work, we represent code with full ASTs and carefully engineered code features, and combine the representation with structured prediction to predict AST-level repair transforms. The predicted high-level repair transforms are then concretized into the final patches on top of effective code snippet synthesis algorithms. To our knowledge, this radically departs from the existing related work in data-driven program repair.

7.5 Conditional Random Fields for Programming

Conditional random fields (CRFs) belong to the powerful probabilistic graphical models (PGM) and have been successfully used in a wide variety of applications, including information retrieval [32], natural language processing [33], bioinformatics [34], and computer vision [35]. Recent years have also witnessed a growing interest of using CRFs for programming tasks, and we finally review representative works in this direction.

By modelling programs with CRFs, Raychev et al. [10] achieve good results in predictions of identifier names and variable types for JavaScript code. To establish relationships between program elements, they use certain grammar rules peculiar to JavaScript. He et al. [84] build CRFs for assembly code and then use the established CRF model to predict properties of meaningful elements in unseen stripped binaries, including symbol names, types and locations. Similarly, Bichsel et al . [114] target Android applications and employ CRFs to achieve prediction of identifier names renamed by layout obfuscation. Given the increased availability of software engineering social content (e.g, Stack Overflow Q&A discussions), Ye et al. [115] adapt CRFs for analyzing software engineering texts such that a broad category of software entities for a wide range of popular programming languages, platforms, and libraries can be recognized. Differently, our way of establishing the CRF model for code is novel and our use of CRFs for predicting code transforms is fundamentally original.

8 Conclusion

A computer program evolves under a sequence of code transforms, for reasons such as bug fixing and code refactoring. These code transform activities represent a significant amount of the cost of a system, calling for tools to aid them. We present in this paper the first approach for structurally predicting code transforms at the level of AST nodes. Our approach leverages conditional random fields (CRFs), and first learns offline a probabilistic model that captures how certain code transforms are applied to certain AST nodes from existing data, and then uses the learned model to predict transforms for arbitrary new, unseen code snippet. Our approach is generic for different kinds of code transforms and languages, and involves novel representation of both programs and code transforms. In particular, we introduce the formal framework for defining AST-level code transforms and we demonstrate how the CRF model can be accordingly designed, learned, and used for prediction. We present in this paper an instantiation in the context of repair transform prediction for Java programs. Our instantiation contains a set of carefully designed code features, and deals with the training data imbalance and transform constraint issues that arise for repair transform prediction problems. A large-scale experimental evaluation has shown the effectiveness of our approach. The result also shows that our model outperforms two baselines based on history probability and neural machine translation (NMT), suggesting

the importance of considering code structure in achieving good prediction accuracy for AST-level code transform. In addition, a proof-of-concept synthesizer is implemented to concretize some repair transforms to get the final patches. The evaluation of the synthesizer on the Defects4j benchmark confirms the usefulness of the predicted AST-level repair transforms in producing high-quality patches.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (Grant No. 62102233), Shandong Province Overseas Outstanding Youth Fund (Grant No. 2022HWYQ-043), Qilu Young Scholar Program of Shandong University, and Wallenberg Artificial Intelligence, the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation. Some experiments were performed on resources provided by the Swedish National Infrastructure for Computing.

REFERENCES

[1] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[2] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[3] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," *SIGPLAN Not.*, vol. 37, no. 5, pp. 304–314, May 2002.

[4] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[5] X. B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 213–224.

[6] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *PLDI*, 2018, pp. 481–495.

[7] 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, Tech. Rep. 1901.01808, 2018.

[8] G. BakIr, T. Hofmann, B. Schölkopf, A. J. Smola, and B. Taskar, *Predicting structured data*. MIT press, 2007.

[9] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *ICML*, 2001, pp. 282–289.

[10] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *POPL*, 2015, pp. 111–124.

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *PLDI*, 2018, pp. 404–419.

[12] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," 2018. [Online]. Available: https://arxiv.org/abs/1812.08434

[13] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[14] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 2016, pp. 39–40.

[15] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: Learning to repair compilation errors," in *ESEC/FSE 2019*. New York, NY, USA: ACM, 2019, pp. 925–936.

[16] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *PLDI '11*. New York, NY, USA: ACM, 2011, pp. 329–342.

[17] ——, "Lase: locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 502–511.

[18] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 404–415.

[19] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

[20] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-large-scale software repository and source-code mining," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 7:1–7:34, Dec. 2015.

[21] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: Patterns, replacements, deletions, and additions," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 512–515.

[22] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 761–774.

[23] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[24] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114.

[25] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 602–614.

[26] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[27] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 341–353.

[28] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, 05 2018.

[29] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Acm Sigplan Notices*, vol. 50, no. 10. ACM, 2015, pp. 416–432.

[30] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 275–286.

[31] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[32] D. Pinto, A. McCallum, X. Wei, and W. B. Croft, "Table extraction using conditional random fields," in *SIGIR*, 2003, pp. 235–242.

[33] F. Sha and F. Pereira, "Shallow parsing with conditional random fields," in *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2003, pp. 213–220.

[34] B. Settles, "Biomedical named entity recognition using conditional random fields and rich feature sets," in *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications (NLPBA/BioNLP)*. Geneva, Switzerland: COLING, Aug. 28th and 29th 2004, pp. 107–110.

[35] X. He, R. S. Zemel, and M. A. Carreira-Perpinan, "Multiscale conditional random fields for image labeling," in *CVPR*, 2004.

[36] C. Sutton and A. McCallum, "An introduction to conditional random fields," *Found. Trends Mach. Learn.*, Apr. 2012.

[37] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014, pp. 313–324.

[38] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[39] I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun, "Large margin methods for structured and interdependent output variables," *Journal of machine learning research*, vol. 6, no. Sep, pp. 1453–1484, 2005.

[40] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[41] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *ASE 2017*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 648–659.

[42] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017, pp. 727–739.

[43] E. C. Campos and M. d. A. Maia, "Discovering common bug-fix patterns: A large-scale observational study," *Journal of Software: Evolution and Process*, vol. 31, no. 7, p. e2173, 2019.

[44] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, "Towards an automated approach for bug fix pattern detection," *arXiv preprint arXiv:1807.11286*, 2018.

[45] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[46] C. Elkan, "The foundations of cost-sensitive learning," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 973–978.

[47] D. M. Tax and R. P. Duin, "Support vector data description," *Machine Learning*, vol. 54, no. 1, pp. 45–66, Jan 2004.

[48] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical Programming*, pp. 503–528, Aug 1989.

[49] F. V. Jensen and F. Jensen, "Optimal junction trees," in *Proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence*, ser. UAI'94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 360–366.

[50] J. Pearl, "Reverend bayes on inference engines: A distributed hierarchical approach," in *Proceedings of the Second AAAI Conference on Artificial Intelligence*, ser. AAAI'82. AAAI Press, 1982, pp. 133–136.

[51] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[52] F. Jousse, R. Gilleron, I. Tellier, and M. Tommasi, "Conditional Random Fields for XML Trees," in *Workshop on Mining and Learning in Graphs*, Berlin, Germany, Sep. 2006.

[53] 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*, 2014, pp. 437–440.

[54] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning (ICML)*, 2016.

[55] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," *CoRR*, vol. abs/1701.02810, 2017.

[56] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018, pp. 1–11.

[57] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *ASE*, 2017, pp. 648–659.

[58] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *ASE*, 2017, pp. 660–670.

[59] ——, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 226–236.

[60] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 831–841. [Online]. Available: https://doi.org/10.1145/3106237.3106274

[61] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *ICSE 2020*. ACM, 2020, p. 615–627.

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

[63] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: Query synthesis from natural language," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017.

[64] G. De la Torre, R. Robbes, and A. Bergel, "Imprecisions diagnostic in source code deltas," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 492–502.

[65] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 483–494.

[66] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. Association for Computing Machinery, 2014, p. 254–265.

[67] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 779–790.

[68] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in java?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 516–519.

[69] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 653–663.

[70] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*, 2014, pp. 419–428.

[71] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 357–367.

[72] Z. Yu, C. Bai, and K.-Y. Cai, "Does the failing test execute a single or multiple faults? an approach to classifying failing tests," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 924–935.

[73] ——, "Mutation-oriented test data augmentation for gui software fault localization," *Information and Software Technology*, vol. 55, no. 12, pp. 2076–2098, 2013.

[74] Z. Yu, H. Hu, C. Bai, K. Cai, and W. E. Wong, "Gui software fault localization using n-gram analysis," in *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, 2011, pp. 325–332.

[75] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[76] W. Ni, J. Sunshine, V. Le, S. Gulwani, and T. Barik, "recode: A lightweight find-and-replace interaction in the ide for transforming code by example," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 258–269.

[77] S. Proksch, S. Nadi, S. Amann, and M. Mezini, "Enriching in-ide process information with fine-grained source code history," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 250–260.

[78] X. Huang, L. Zhang, B. Wang, F. Li, and Z. Zhang, "Feature clustering based support vector machine recursive feature elimination for gene selection," *Applied Intelligence*, vol. 48, pp. 594–607, 2018.

[79] B. F. Darst, K. C. Malecki, and C. D. Engelman, "Using recursive feature elimination in random forest to account for correlated variables in high dimensional data," *BMC genetics*, vol. 19, no. 1, pp. 1–6, 2018.

[80] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[81] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[82] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, 2009.

[83] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018.

[84] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *CCS*, 2018, pp. 1667–1680.

[85] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," in *ACM SIGPLAN Notices*, vol. 53, no. 4. ACM, 2018, pp. 450–464.

[86] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A deep learning model for estimating story points," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 637–656, 2019.

[87] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *ICSE*, May 2020.

[88] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[89] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 7762–7773.

[90] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2020, p. 91–105.

[91] J. Eberhardt, S. Steffen, V. Raychev, and M. Vechev, "Unsupervised learning of api aliasing specifications," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019.

[92] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 572–588.

[93] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian sketch learning for program synthesis," in *ICLR*, 2018.

[94] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA, 2019, p. 531–548.

[95] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[96] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated software engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[97] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, "On the fly synthesis of edit suggestions," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[98] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[99] S. Brody, U. Alon, and E. Yahav, "A structural model for contextual code changes," 2020.

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

[101] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[102] 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, pp. 1–29, 2019.

[103] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," in *Proceedings of ESEC/FSE*, 2018.

[104] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*, 2018.

[105] J. Henkel, S. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proceedings of ESEC/FSE*, 2018.

[106] 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*, 2017, pp. 763–773.

[107] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *ICSE*, 2016, pp. 297–308.

[108] 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, pp. 54–72, 2012.

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

[110] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys*, vol. 51, pp. 1–24, 2017.

[111] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[112] 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*, 2018, pp. 298–309.

[113] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[114] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA, 2016, p. 343–355.

[115] D. Ye, Z. Xing, C. Y. Foo, Z. Q. Ang, J. Li, and N. Kapre, "Software-specific named entity recognition in software engineering social content," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 90–101.