Embedding Context as Code Dependencies for Neural Program Repair

Noor Nashid
University of British Columbia
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
University of British Columbia
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

***Abstract*—Deep learning-based program repair has received significant attention from the research community lately. Most existing techniques treat source code as a sequence of tokens or abstract syntax trees. Consequently, they cannot incorporate semantic contextual information pertaining to a buggy line of code and its fix. In this work, we propose a program repair technique called GLANCE that combines static program analysis with graph-to-sequence learning for capturing contextual information. To represent contextual information, we introduce a graph representation that can encode information about the buggy code and its repair ingredients by embedding control and data flow information. We employ a fine-grained graphical code representation, which explicitly describes code change context and embeds semantic relationships between code elements. GLANCE leverages a graph neural network and a sequence-based decoder to learn from this semantic code representation. We evaluated our work against six state-of-the-art techniques, and our results show that GLANCE fixes 52% more bugs than the best performing technique.**

***Index Terms*—deep learning, program repair, program slicing, control flow, data flow, graph neural networks**

I. Introduction

Automated program repair aims to reduce the cost of software debugging and maintenance [1], [2]. With the recent advancement of deep learning and the availability of a massive open-source code corpus, neural program repair is gaining significant traction [3]–[16]. Neural program repair can learn repetitive fix patterns from numerous samples of developer-written patch automatically.

Existing studies [17], [18] on developer behaviour during code comprehension and program repair tasks show that usage and inference of contextual information is an important part of the development process. Consequently, it is essential to leverage contextual information according to the need of a source code processing task. Current neural program repair techniques employ contextual information from the buggy statement and its surrounding context in various ways. This contextual information can be limited to the buggy statement [19], [20], the surrounding lines of code, enclosing function [8], [9], enclosing class [3], enclosing file [7], [21], or encapsulated AST subtree [5].

Despite promising results achieved by existing neural program repair techniques, they have several key limitations that concern the selection and representation of contextual relation, namely, they (1) focus on the buggy statement and choose contextual information in an ad-hoc fashion, e.g., with a pre-defined code token limit [3], [8], [9] or number of nodes in the Abstract Syntax Tree (AST) [7], (2) treat code as a sequence of tokens [3], [8], [9], [20] or AST [22] and do not embed semantic relationships between code elements, and (3) ignore control and data flow dependencies between program entities such as variables, statements, or function calls pertaining to the bug and its fix. For a program repair task to be effective, context needs to encode possibly related information pertaining to the buggy line that is meaningful to fix that bug. We call this the *repair ingredient*. Our hypothesis is that the contextual information based on program dependencies such as control and data flow could be leveraged to extract repair ingredients instead of treating context in an ad-hoc way.

In this paper, we present a framework GLANCE (**G**raph-to-sequence **L**e**A**r**N**ing with **C**ontext **E**mbedding) that leverages contextual information using program analysis for repair learning. We apply backward slicing [23] to extract related repair ingredients pertaining to the buggy statement. Once we isolate the fix ingredient, we extract control and data flow dependencies through static program analysis and embed this context as a graph. We call this graph with control and data dependencies the *Flow-Augmented Graph* (FAG). This graph has explicit control and data flow edges that makes the code amenable to machine learning. We employ a graph-to-sequence architecture for learning patterns from this graphical representation of the code. This paper makes the following contributions:

- A graph-based technique called GLANCE for neural program repair tasks that leverages contextual information using static program analysis.
- A novel *flow-augmented graph* for representing code as a graph augmented with contextual information which combines (a) backward slicing for isolating repair ingredients, (b) buggy statement and context demarcation, and (c) explicit modelling of control and data flow edges to capture relevant semantic relationships between program elements.
- An empirical evaluation of GLANCE against six state-of-the-art techniques to assess the effectiveness through quantitative and ablation analyses. GLANCE achieves 43.02% accuracy in fixing bugs, outperforming the

two top-performing approaches, CoCoNuT [9] and Katana [24], by 74.3% and 52%, respectively.
- A replication package which is publicly available [25].

Our empirical results show that employing our novel *flow-augmented graph* with abstraction and framing the program repair objective as a graph-to-sequence learning task is effective in improving accuracy of GLANCE. Backward slicing extracts contextual information while reducing noise for the learning model. Furthermore, our results indicate explicit modelling of control and data-flow edges helps the neural model to learn from the underlying code structure. Given that most current techniques rely on ad-hoc context representation, our study highlights that the selection of context needs to be more systematic, and program analysis techniques are helpful for selecting and encoding relevant contextual information for learning-based repair.

II. Motivation

Debugging software is an unavoidable part of software development and maintenance [26]. During the software debugging step, at first, developers need to locate the buggy piece of code. Once the buggy piece of code is localized, developers examine the code context systematically. Developers essentially try to narrow down the search space so that they can gather sufficient information that is relevant to fix that bug. To do so, developers started to examine statements surrounding the buggy code, enclosing method, and class to gather contextual information for that bug. As a complementary step to better understand the repair context, it is common for developers to also analyze the flow of execution between code blocks. To achieve this, developers track how variables are initialized and values are modified at different parts of a program. Intuitively, there are two consecutive steps (a) *isolate:* what is the region of interest to fix this buggy piece of code without getting overwhelmed with the codebase, and (b) *understand:* how different code elements are related in this chosen program segment. Current learning-based program repair techniques attempt to leverage context in varied ways. Sequence-based approaches [3], [8], [9] treat source code from a lexical perspective and use the enclosing method, enclosing class, or enclosing file as the context. Furthermore, these sequence-based approaches cannot embed richer semantic information in the streams of text. On the other hand, graph-based approaches [7] use the whole file as AST with a predefined limit to select context in an ad-hoc way. A recent work [24] applied a slicing-based approach to select code statements related to the buggy line. These techniques are limited in how additional information, such as control and data flow, could be embedded.

As a running example, we will use Listing 1, which shows a bug and its fix for a JavaScript project. This type of bug for dereferencing non-values is frequent in JavaScript [27]. Once the developer localizes the buggy line 55, they investigate the buggy file to locate the function `getMarkerAtDocumentPos`. Following that, developers attempt to understand related code segments that are relevant to the buggy line. At this point, they scan the highlighted code region that are depicted in yellow in Listing 1. Now they need to understand how the variable `match` is manipulated in the selected code region. To do so, they analyze the flow of control statements and data flow of variables to understand the program path. In our example, the conditional statement (line-39) branches off to two possible paths to modify the variable `match`. By tracking where the variable `match` is defined and modified, they observe that the variable is set to `null` in the else block (line-42). At this point, they realized how this bug is introduced while dereferencing a null value. To fix this bug, they use the optional chaining operator (`?.`) that is available in ES6.

```
1  use strict;
2
3  var DocumentManager = require("docs/DocumentManager"),
4  ...
25 function getMarkerAtDocumentPos(editor, pos, markCache){
26     var marks, match;
30     markCache = markCache || {};
35     marks = getSortedMarks();
36     if (!marks.length) {
37         return null;
38     }
39     if (marks.length > 1) {
40         match = { mark: marks[marks.length - 2] };
41     } else {
42         match = null;
43     }
45     markTags(editor, pos)
46     ....
55 -   return match.mark;
55 +   return match?.mark;
56 }
60 function markTags(cm, node) {
61     ...
```

Listing 1: Sliced program with incorrect null dereference

Current learning-based repair techniques [3], [7]–[9] either arbitrarily choose context, which adds noise to learning or cannot embed relationships between code elements. While a recent technique [24] uses slicing-based program analysis, the relationship that could be derived from control flow and data flow is still not leveraged. However, source code has richer semantics that could be leveraged from the program's control, data, and call dependencies. Using the program data and control flow as a graphical code structure would make the representation more amenable to learn from. We hypothesize that the code region relevant to the buggy line and the relationship within that region are important for a repair task. To that end, we build a framework that enables us to embed semantic information using control and data flow.

III. Approach

In this section, we describe the design of GLANCE, a novel framework that combines program analysis with graph-to-sequence learning for source code. At a high level, given a code segment, GLANCE first derives a flow-augmented graph, as illustrated in Figure 1, by employing backward slicing, context delineation, and embedding control, data-flow dependencies. The resulting flow-augmented graph is then fed to a graph-to-sequence (Graph2Seq) based model. This approach initially learns word embedding, then derives final node embedding via message passing. GLANCE then employs an RNN-based decoder to generate the final output sequence. Our overall approach, including the model architecture, is shown in Figure 2. Next, we elaborate on each step.

A. *Flow-Augmented Context Representation*

Given a JavaScript file with a buggy line of code, GLANCE employs a static analysis technique, namely backward slicing [28], to determine the relevant ingredients for program repair, such as the variables, objects, function calls, and where these ingredients have been defined, used and modified. In addition, for each of these ingredients, GLANCE captures control and data-flow information where applicable. We employ both intra-procedural (within function) and inter-procedural (within program) analysis. As JavaScript code allows stepwise execution in the global code without requiring a main method (e.g., in Java), we opted to go beyond intra-procedural analysis, but limit the analysis scope to the enclosing file as a significant portion of the fixing ingredients can be found in the file containing the bug [29].

We perform a pre-processing step where whitespaces, empty lines and comments are removed from the sliced code. We then construct a JSON object which contains the buggy line number, backward sliced code as context with respect to the buggy line, control flow and data flow analysis of the sliced code, and the target fixed line, as shown in Figure 2. Next, we represent this contextual information by constructing a Flow-Augmented Graph (FAG).

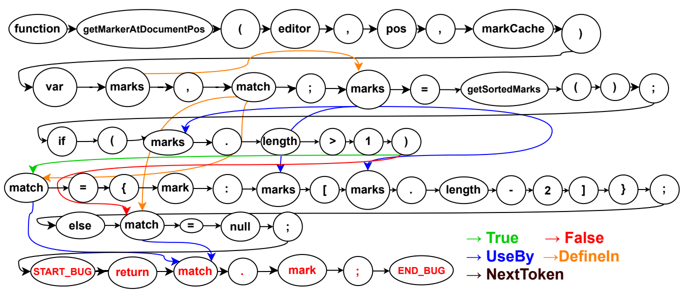


Fig. 1: Flow-augmented graph.

We tokenize the sliced code snippet to form a stream of tokens using the word-level tokenization – a widely used technique in NLP. GLANCE then performs code abstraction wherein strings and literals in the code snippet are replaced with tokens `$STRING$` and `$LITERAL$`, similar to a previous work [9].

With the extracted stream of tokens, GLANCE creates a graph where each token is a node and has a directed edge connecting it to the next token in the stream. We then enclose the tokens from buggy line with two additional tokens – `START_BUG` and `END_BUG` for delineation of the buggy line. We refer to this resulting graph as a Sequence Graph (SEQGRAPH).

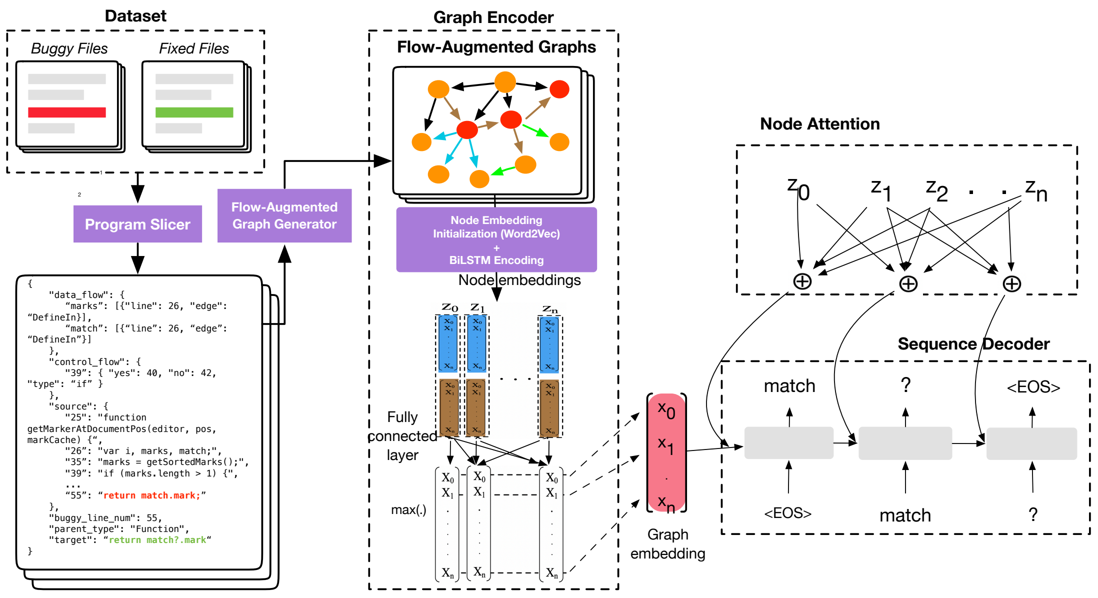
Next, we add control and data-flow edges to this SEQGRAPH using the information extracted during program analysis phase. The control flow edges are added to indicate the flow of program execution for conditionals, loops and try-catch blocks. For example, in Figure 1, the green and red edges indicate the control flow in a conditional statement, where the edges point to the first token in the particular code path. The data-flow edges represent the result of the def-use analysis. For example, the orange and blue edges indicate the data flow ($DefineIn, UseBy$) from the nodes of variables `marks` and `match` to the nodes in lines where they have been defined and used. Similarly, the output of call-graph analysis is indicated through $CallBy$ edges. The buggy line is color encoded in red and enclosed within two bug delineation tokens, `START_BUG` and `END_BUG`, as illustrated in Figure 1. We refer to this resulting graph where SEQGRAPH is enhanced with control and data-flow edges to be a flow-augmented graph (FAG). To the best of our knowledge, we are the first to utilize program analysis output in a graphical representation for a learning based repair task. A flow-augmented graph (FAG) is different from a program-dependence graph (PDG) [30] in two ways: *(i)* the smallest possible unit in a FAG are tokens instead of statements and *(ii)* the control and data-flow edges in a FAG are applied on individual tokens which provides more granularity in determining the exact locations of identifier definition and use. We employ the extracted FAG as input for our learning model. Next, we describe how FAGs are employed for training and inference of a learning model built for program repair task.

B. *Model Architecture*

Overall, we designed our model, shown in Figure 2, to follow an encoder-decoder architecture pattern where we employ a GNN as encoder and an RNN as decoder. We choose GNN as our encoder because of their ability to process graph-structured data [31]–[33]. We employ RNN as decoder because of their popularity in generation tasks. In the rest of this subsection, we describe details of the model components.

Embedding construction module. To apply learning-based techniques, source code needs to be vectorized. As shown in Figure 2, at first we feed FAG into the embedding construction module. The goal of the embedding construction module is to learn the initial node embedding from the input FAG before feeding it to the subsequent GNN encoder. The initialization of node embedding is critical for GNN-based models [34]. To capture the name-based semantics of source code [35], we first use word2vec [36] embeddings to initialize the tokens. Following that, we apply a BiLSTM [37] encoder to update the initial embedding further. The intuition behind using the bidirectional sequential information is it would capture the relationship between preceding tokens and subsequent code tokens in the source code corpus.

Graph Encoder. The embedding vectors are now fed into the graph convolutional network (GCN) layer, as presented in Figure 2. GCN is a variant of the graph neural network



```
{
    "data_flow": {
        "marks": [{"line": 26, "edge":
"DefineIn}],
        "match": [{"line": 26, "edge":
"DefineIn"}]
    },
    "control_flow": {
        "39": { "yes": 40, "no": 42,
"type": "if" }
    },
    "source": {
        "25": "function
getMarkerAtDocumentPos(editor, pos,
markCache) {",
        "26": "var i, marks, match;",
        "35": "marks = getSortedMarks();",
        "39": "if (marks.length > 1) {",
        ...
        "55": "return match.mark;"
    },
    "buggy_line_num": 55,
    "parent_type": "Function",
    "target": "return match?.mark"
}
```

Fig. 2: Overview of our approach.

that uses spectral-based graph filters [38]. At this stage, we aggregate the feature information among all neighbour nodes in FAG. GCN can capture information only about immediate neighbours with one layer of convolution. So we adopt multiple-layer GCN to learn from the larger neighbourhood and perform a layer-wise propagation rule following spectral graph theory [38].

Sequence Decoder. For sequence generation, we employ a Recurrent Neural Network (RNN) as the decoder. This sequence decoder is responsible for decoding the target sequence from vectors produced by the graph encoder. The graph-based decoder, like most sequence-to-sequence decoders, uses an attention mechanism to learn the soft alignment between the input FAG and the bug fixing code sequence [39]. However, there are differences in how the attention mechanism is leveraged. Unlike other sequence-to-sequence decoders, the decoder attends to both the sequential output and the tokens in the graph encoder. Then it fuses two vectors using concatenation to obtain the attention result into a single context vector to generate the next token. The decoder predicts the next token until `<EOS>` tag is read and produces the correct patch sequence as the final output during inference, as shown in Figure 2. We apply the copy [40] and coverage [41] mechanism in the decoder. For the copy mechanism, the model copies words directly from the source sequence using a pointer network [42].

Patch Generation. After training, we use the test dataset to evaluate the model during inference. Given a buggy program, we assume a fault localization step already identifies the potential buggy statement, which is consistent with other learning-based program repair techniques [3], [5], [9]. For the evaluation of SEQGRAPH representation, input to the inference engine is the SEQGRAPH, constructed from backward sliced buggy file and bug delineation tokens as defined in Section III-A. On the other hand, to evaluate FAG, input to the inference engine is a FAG, which is constructed from the backward sliced buggy file and augmented with control and data flow edges along with bug delineation tokens, as shown in Figure 1. The output of the model is a sequence of tokens which is the predicted fix for the bug. Therefore, given a buggy FAG, g_{bug} fed into the graph encoder, the sequence decoder tries to predict the next token y_i, following the previously predicted tokens $y_{<i} = y_1, ...y_{i-1}$, RNN hidden state s_i and the context vector c_i [43]. Instead of greedy search, we use beam search to select the top k candidates as the candidate output sequence, where k is the beam size.

IV. EXPERIMENTAL SETUP

A. Dataset

Our technique is evaluated on JavaScript bugs collected from open source GitHub projects. JavaScript, which was initially used primarily for client-side web development, has gained popularity for backend development with the emergence of the Node.js [44] framework. Due to its weak and dynamic typing, coding with JavaScript poses unique challenges which can lead to a myriad of language-specific bugs and code smells e.g. `let` and `const` interchangeability, null exception, incorrect comparison [27], scoping errors caused by `var` [45] etc. Therefore, we opted to use JavaScript as our target language for repairing bugs.

Data Deduplication. To train a model and evaluate it faithfully, we need to eliminate duplicate data points [46]. We remove all duplicates from the test dataset as well as any recurrent data points between the test and training datasets.

Data Collection. The dataset is collected from open-source GitHub repositories. We further filter out commits based on the number of AST node differences. This heuristic, employed by an existing work [7], considers a difference of one node between buggy and fixed files because it is more likely to be considered a bug than refactoring or ad-hoc code change. Our

TABLE I: Hyperparameters and the optimal configuration

Hyperparameter	Possible Values	Tuned Value
GNN layers	[1, 2, 3]	2
Learning rate	[0.1, 0.01, 0.001]	0.001
GNN Dropout	[0.1, 0.2, 0.3]	0.2

dataset consists of top 1,100 open-source projects (based on GitHub stars) containing buggy commits from both frontend and backend JavaScript code. We collected a total of 113,975 pairs of buggy and fixed JavaScript files having one AST node difference between the pairs. We folded the dataset into 80% (91,181) training, 10% (11,397) validation and 10% (11,397) testing sets.

B. Hyperparameter Optimization and Training

GNNs need to be tuned for optimal performance with a hyperparameter optimization step. As shown in Table I, we constructed a search space of various configurations following previous studies [47], [48]. We tuned the model with unsliced code, i.e., without slicing, as it is the closest to the actual code. We trained 27 different models, and the tuned value is shown in Table I. We use Adam optimizer [49] to update network weights. After 5 epochs, we observed negligible improvement in the accuracy score with this optimal configuration. As a result, the stopping criterion for training 5 epochs. For the sliced code, the batch size was fixed at 16 for all training. However, since unsliced code has a larger graph size, it consumes more GPU memory. As a consequence, we kept the batch size for unsliced code fixed at 4 since that was the largest batch size our infrastructure could handle. We describe details of our infrastructure in Section IV-D. Throughout the training, the embedding size was set at 300. For the decoder, we set dropout to 0.3, following best practices [50]. We evaluate all techniques with a beam size of 5 and use top-1 inference to calculate accuracy. Our final input vocabulary size for the dataset is 110,470 tokens.

C. Evaluation Metrics

We use commonly used evaluation metrics such as accuracy [11], [51], [52] and BLEU-n score [53]. For all the metrics, a higher value is considered a better score and the metrics are reported as an average for the entire test set.

Accuracy. Accuracy is measured as the percentage of unseen test dataset for which the model can suggest an expected output. Recent work [51] has highlighted the need to employ *top-1* accuracy as the primary criteria for assessment, which has been also adopted in other studies [10], [11], [52]. Similarly, we choose top-1 exact match accuracy as the key evaluation metric since it is the most stringent assessment criterion for evaluation. There are also approaches that propose suggesting top-k patch candidates and filtering the actual patch based on available test cases. However, test cases may not always be available for learning-based program repair. Furthermore, showing a large number of patch candidates to the developer would limit the usability of a technique. Hence, we adopt *top-1* accuracy throughout this study.

BLEU-n score. BLEU [54] is a number between 0 and 1, which measures the precision of generated sequences by calculating the average of the n-gram precision (i.e., 1-gram, 2-grams, 3-grams and 4-grams for BLEU-4). It is a widely used metric in the SE literature [53], [55]–[59]. The n-gram precision is defined as the ratio of the number of matched n-grams to the number of all the n-grams in the generated sequence. This metric is used to determine the similarity between the expected patch and the actual patch.

D. Implementation

Dataset collection. For collecting our dataset, we employ PyDriller [60] library to crawl commits from the selected GitHub projects based on the keywords ‘bug’, ‘fix’ and ‘resolve’.

Curation of FAG. We use the tool Understand by SciTools [61] for static program analysis of the buggy JavaScript files. As we were unable to find any existing slicing tool that supports the latest JavaScript ES10 features, we implemented a slicer for extracting the control and data flow information with respect to the slicing criterion, following previous work [24], [62]. The slicer takes as input a buggy JavaScript file, correct line and the slicing criterion, which is the buggy line and the entities (variables, objects or functions) used in that line. It produces as output a JSON object containing the sliced code, control flow object, data flow object, buggy line number and the correct line, which is parsed for creating the FAG.

GNN model. We implemented our model architecture using a recently released graph neural network library, GRAPH4NLP [34], [50]. We customized the graph encoder and sequencer decoder components from this library to build the learning framework for GLANCE.

Infrastructure. Hyperparameter tuning, training, and inference were performed on a single Amazon EC2 G4DN.12XLARGE instance running Ubuntu 18.04.2 LTS, which is optimized for running deep-learning workload. This server was equipped with 48 Intel Cascade Lake vCPUs, 4 NVIDIA T4 GPUs, 192 GB memory, and 500 GB SSD.

V. EVALUATION

We address the following research questions in order to assess the effectiveness of GLANCE:

RQ1 What are the contributing factors on the overall accuracy of GLANCE?

RQ2 What is the effect of embedding contextual semantics information?

RQ3 How well does our technique perform in comparison with state-of-the-art learning-based program repair approaches?

A. Role of bug delineation and abstraction (RQ1)

In this section, we discuss how the different components of GLANCE affect the overall performance in accurately generating patches. We represent source code as a flow-augmented graph (FAG) having control and data flow edges. Additionally, we incorporate bug delineation to differentiate the buggy line from context within the graph, as well as abstraction to limit the vocabulary size. In order to assess the

TABLE II: Results - role of abstraction and bug delineation

Abstraction	Bug delineation	Accuracy (%)	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Duration(hh:mm:ss)
No	No	2.69	0.1905	0.1072	0.0728	0.0542	10:01:21
No	Yes	7.32	0.3175	0.2017	0.1442	0.1113	10:05:56
Yes	No	25.69	0.3532	0.2727	0.2171	0.1780	7:32:05
Yes	Yes	40.43	0.6114	0.5160	0.4364	0.3752	8:22:05

TABLE III: Results - role of control and data flow edges

Program Analysis	Representation	Accuracy (%)	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Duration (hh:mm:ss)
Unsliced	SEQGRAPH	37.40	0.5306	0.4335	0.3555	0.2968	41:07:00
	FAG	39.73	0.5705	0.4783	0.4018	0.3428	38:32:55
Sliced	SEQGRAPH	40.43	0.6114	0.5160	0.4364	0.3752	8:22:05
	FAG	43.02	0.6165	0.5253	0.4480	0.3876	8:21:54

importance of these two components individually, we conduct a sensitivity analysis on the sequence graph (SEQGRAPH) by excluding the control and data flow edges. Table II shows the ablation study of how the two components contribute to the accuracy of GLANCE. We used the metrics described in Section IV-C for evaluation. The quantitative metrics in each column is color coded, where the darker hue entails better result for the specific metric. Without abstraction and bug delineation, the accuracy of GLANCE is only 2.69% which is very low. The n-gram BLEU scores are below 0.2 which denotes that the tokens in generated patches are very different from the ground truth. Furthermore, the model takes around 10 hours to train due to the absence of these two components. By adding only bug delineation, we observe an increase in accuracy to 7.32% and BLEU score ranging from 0.1 to 0.3. With the addition of abstraction only, the accuracy significantly increases to 25.69%, however the BLEU score increases slightly with more improvement in case of 3-grams. We also observed a 3.5 hours decrease in training time with the inclusion of only abstraction. Finally, with the addition of both abstraction and bug delineation, the accuracy increases to 40.43% and the training time increases by 50 minutes. The BLEU score improves in all the 4-grams – with 0.6114 as the highest in BLEU-1 and 0.3752 being the lowest in BLEU-4.

B. *Encoding Semantic Information (RQ2)*

In GLANCE, we encode the sliced buggy source code and its context as a FAG that leverages control and data flow information both implicitly (as sliced code) and explicitly (as graph edges). The most basic representation of our code graph is a SEQGRAPH that connects all tokens using the $NextToken$ edge. We consider this representation as the *baseline*. We then add control and data flow edges on top of SEQGRAPH to form the FAG. In order to understand how these factors contribute to the accuracy of our framework, we conducted an ablation study with the SEQGRAPH and FAG on both the sliced and unsliced buggy code. Here, we refer to the dataset before slicing code as unsliced. We represented the unsliced code as a FAG and SEQGRAPH, but kept the bug delineation and abstraction aspect on. Table III shows the results of our ablation study. We used the same two metrics as the previous section for evaluation. In the unsliced code, we observed that the accuracy of bug fixes for SEQGRAPH was 37.40% with highest BLEU score 0.5306 for 1-gram and lowest BLEU score of 0.2968 for 4-gram. We then build on top of the SEQGRAPH and perform program analysis to add control and data flow edges. This increases the accuracy to 39.73% with an increased BLEU score ranging from 0.3428 to 0.5705 for 1 to 4 grams. However, for the unsliced dataset, training time is significantly high, with 41 hours for SEQGRAPH as compared to 38 hours for FAG.

Next, we train the model on sliced code and represent it as a SEQGRAPH and FAG. To construct FAG, on average, the program analysis step takes around 330 milliseconds for a single data point, and this pre-processing step does not incur significant overhead. As shown in Table III, the accuracy of program repair slightly increases further after using the sliced version of buggy code to 40.43%. The BLEU score increases to 0.6114 in BLEU-1 and 0.3752 in BLEU-4. After adding control and data flow edges using a FAG, the accuracy of bug fixes increases to 43.02%. The BLEU score also increased to 0.6165, 0.5253, 0.4480 and 0.3876 for 1 to 4 grams, respectively. For the sliced dataset, the training time reduced considerably, taking around 8 hours 22 minutes for both of the graph representations.

C. *Comparative study with neural-repair approaches (RQ3)*

Baseline Selection Criteria. There is a wide array of neural program repair techniques in the literature [3], [5], [7]–[9], [11], [12], [14]–[16], [20], [63]–[65]. Our selection criteria for our comparative evaluation included (1) the availability of the technique's source code, and (b) support for JavaScript or the ability to retrain the tool's model on a JavaScript dataset. We considered a number of techniques that were excluded based on these selection criteria. For instance, CURE [63] and Recoder [14] are designed for Java and rely on language-specific implementations. As a result, extending these tools to a dynamic language such as JavaScript would not be trivial; RewardRepair [16] relies on Java test cases during training, which is not available in our dataset; CIRCLE [12] is a multilingual repair technique, but the tool/model is currently not available in their latest official repository [66].

We included SEQUENCER [3] and Tufano et al. [8] despite being designed for Java. The reason is that neither of these techniques relies on language-specific features, and they employ a sequence-based model that can be extended to other

TABLE IV: Design comparison of GLANCE with other learning-based program repair techniques.

	Approaches	**Tufano et al. [8]**	**Hoppity**	**SequenceR**	**PLUR**	**CoCoNuT**	**Katana**	**GLANCE**
Model Architecture	**Encoder**	BiLSTM	GIN	BiLSTM	GGNN	FConv-Context	GIN	GCN
	Decoder	BiLSTM	LSTM	BiLSTM	ToCoPo	FConv	LSTM	RNN
	Input	Buggy Sequence	Buggy AST	Buggy Sequence	Buggy AST	Buggy Sequence	Dual Sliced Buggy AST	FAG
	Output	Correct Sequence	AST Edits	Correct Sequence	AST Edits	Correct Sequence	AST Edits	Correct Sequence
Context Extraction	**Scope**	Enclosing method	Enclosing file	Enclosing file	Enclosing class	Enclosing method	Enclosing file	Enclosing file
	Program Slicing	✗	✗	✗	✗	✗	✓	✓
	Control & data-flow delineation	✗	✗	✗	✗	✗	✗	✓

languages. We also included CoCoNuT [9], Hoppity [7], PLUR [67], and Katana [24] as they all support JavaScript. TFix [11] was excluded despite being a JavaScript repair tool because it is specifically designed for fixing linter errors, and it requires ESLint error messages. While we considered including DLFix [5] and DEAR [15], both were unavailable at the time of writing this paper. Our findings are consistent with a recent comparative study [68] on learning-based program repair tools, which observed that not all tools have their artifacts available for comparison and many of these tools cannot be trained on a new dataset from scratch.

Thus, the set of tools to compare against consists of two learning-based repair tools built for Java (Tufano et al. [8], SequenceR), and four repair tools (Hoppity, PLUR, CoCoNuT, Katana) that already support JavaScript.

Design Comparison. We present the design choices based on model architecture and context extraction of different repair tools in Table IV.

- **Encoder**: Depending on the graph or token sequence input to the model, variants of LSTM or GNN could be employed to learn latent vector representation.
- **Decoder**: To produce the output sequence, variants of RNN could be leveraged.
- **Model Input**: Code could be fed as text sequence, AST or graph to the model.
- **Model Output**: Output from the model could be a correct sequence pertaining to the buggy line or AST edit(s) applied to the buggy AST.
- **Context Scope**: Input to the model could be derived from a varied level of contextual information: enclosing method, enclosing class, whole file.
- **Program Slicing**: Context pertaining to the buggy line could be leveraged using program slicing.
- **Context Delineation**: A fine-grained context delineation could be embedded using control and data-flow edges.

Evaluation Setup. Now we describe the steps we used to train models from the selected learning-based program repair techniques with our JavaScript dataset using their artifact.

Tufano et al. [8] investigate the potential of NMT to generate candidate patches by using a recurrent neural network (RNN) encoder-decoder architecture, as shown in Table IV. It uses a sequence-to-sequence approach to represent buggy methods as input and fixed methods as output. They limit the overall number of tokens of each methods to 100 tokens as a means to reduce the scope of context and apply abstraction to limit the vocabulary size. Since the bugs in our dataset are not always scoped within a function, we captured the surrounding tokens of a buggy line as context such that the total number of tokens is within the 100 token limit. After tokenizing the buggy and fixed JavaScript files, we generate a dictionary of frequent tokens and replaced the infrequent identifiers, methods and variables with sequential abstracted tokens (e.g. `METHOD_1`, `VARIABLE_1`). The final vocabulary size was 815, containing abstracted tokens, JavaScript keywords and frequent tokens.

Hoppity [7] uses a graph neural network for learning graphs to code edits, using a pre-defined node limit. Hoppity is a graph-based program repair technique which represents the source code as AST to feed as input to the model and the output is AST edits applied to the buggy graph. The context scope is limited to 500 nodes within the enclosing file of the buggy line. The final vocabulary size from our dataset for this framework was 5,003 tokens.

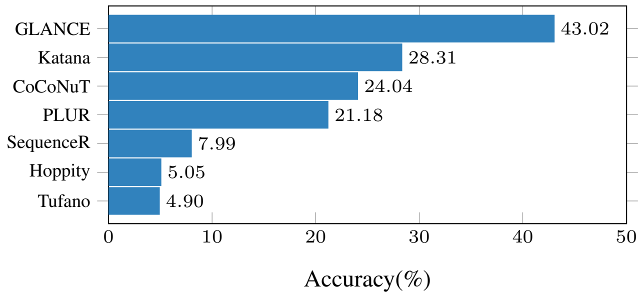
SequenceR [3] employs a sequence-to-sequence network along with a copy mechanism for repair learning. In SequenceR, we represented the buggy JavaScript file as a sequence of tokens and the output is a fixed line (see Table IV). We start by encapsulating the buggy line within the tokens `<START_BUG>` and `<END_BUG>` and capture context by extracting 2/3rd more tokens from the preceding statements of the buggy line following their approach. SequenceR limits the context by keeping the max token size to 1,000 tokens, which we incorporated in our dataset as well. The final vocabulary size for this framework was limited to 1,000 tokens.

PLUR [67] introduced a graph learning framework for different source code processing tasks such as program repair. In their work, PLUR demonstrated superior accuracy in comparison to Hoppity using a GGNN to encode the graph structure and a ToCoPo decoder to output edit operation in the AST. We employ the same model architecture using reported hyperparameters from their artifact.

CoCoNuT [9] also employs a sequence-to-sequence learning using CNN and ensemble learning for learning repair. It uses the surrounding tokens of a buggy line as context by limiting the scope to 1,022 tokens. Their framework incorporates abstraction for tokens that are strings and literals. The final vocabulary size generated from our dataset for this framework was 63,499 tokens.

Katana [24] employs the code representation and learning model from Hoppity. However, during training, Katana uses program slicing to derive context from both the buggy and fixed files, namely dual-slicing, instead of ad-hoc quantitative limitation through nodes or tokens. For inference, Katana extracts backward sliced context from buggy line as input. This is unlike other deep learning approaches as input to the model is not the same during training and inference. The vocabulary size for KATANA was the same as HOPPITY.

Fig. 3: Accuracy comparison of GLANCE with other tools.



Results. In Figure 3, we show accuracy for all the six deep learning based program repair techniques. The lowest accuracy was obtained by Tufano et al. [8] with only 4.9% accuracy in fixing bugs. Even though the context scope is limited to 100 tokens with abstracted literals and identifiers, the model was not able to infer enough information to generate patches. HOPPITY was able to fix bugs with only 5.05% despite being a sophisticated graph-based technique that generates patches using a series of AST edits. SEQUENCER performed slightly better than HOPPITY with 7.99% accuracy. PLUR outperformed SEQUENCER with an accuracy of 21.18%. COCONUT demonstrated slightly better results than the previous techniques with 24.67% in fixing bugs, followed by KATANA which showed a more improved accuracy of 28.31% due to its dual slicing technique. GLANCE outperformed all six techniques with a 43.02% accuracy in fixing bugs. Using a novel representation of source code, bug delineation and abstraction implemented in a graph-to-sequence model, GLANCE is able to surpass the existing techniques by 777.8% (Tufano et al.), 751.7% (HOPPITY), 438.3% (SEQUENCER), 103.12% (PLUR), 74.3% (COCONUT), and 52% (KATANA) percentage increases in accuracy.

VI. DISCUSSION

Bug delineation plays a role. For a learning-based model, it is essential to understand what encompasses the buggy line and where it appears in the context to learn from a source code. Without this explicit delineation of the buggy line, it is (a) harder for the model to extract the relationship between the buggy statement and its context and (b) difficult to learn the alignment between the buggy statement and correct statement. As a result, highlighting the buggy code statement in the graph representation helps the model to learn. As shown in Table II, bug delineation has a significant impact on the overall accuracy.

While we delineate the buggy statement in this work, we do not highlight what part from within the buggy statement is changed to go to the correct statement. For example, we could embed the addition and deletion that needs to happen to go from the buggy statement to the fixed statement. As a future work, this fine-grained edit representation is promising as it could further assist the model in understanding the alignment between the buggy statement and the correct statement.

Program analysis for learning repairs. Current learning-based repair approaches overlook the need to apply program analysis for extracting contextual information. However, program analysis, such as slicing isolates the repair ingredients pertaining to the buggy line. Additionally, the control and data dependencies that exist in source code entities, e.g., variables, statements, functions, could be derived using program analysis techniques that infer control and data dependencies. As reported in Figure 3 our technique GLANCE with FAG outperforms existing approaches. Unsliced code contains redundant context that may not always be relevant to the bug or fix. This type of context adds noise to the learning process and thus, the model requires significant effort in reaching convergence due to larger graphs. We observe this in the training duration where sliced graphs require five times less training time than unsliced. Furthermore, sliced FAGs are 3.3% more accurate than unsliced FAGs. In previous graph-based approaches [7], [24], the buggy source code had been represented as AST with additional edges (e.g. $ValueLink$, $SuccLink$). $SuccLink$ is similar to our $NextToken$ edge which is used for encoding the sequence order by connecting leaf nodes, whereas $ValueLink$ edges connect the value nodes containing actual content of the source code to internal nodes of an AST. However, the resulting augmented AST representation neither contains the control flow nor the data flow semantics in their graphs. We also observe the efficacy of control and data flow edges and, slicing as GLANCE outperformed a superior graph based model – PLUR. As shown in Table II, a SEQGRAPH representation for learning to fix bugs still outperforms the baseline techniques, with 32.1% increase in accuracy from the best performing baseline, KATANA. We hypothesized that, the extra nodes in an AST-based graph representation (e.g. $FunctionDeclaration$, $VariableDeclaration$ etc.) could potentially add noise to the learning process. A similar observation was noted in a recent work [69], where they demonstrated that AST could be noisy and there is a need to simplify AST structure by removing unnecessary tree nodes before feeding it to the learning model. Our results confirm this hypothesis with the introduction of a SEQGRAPH representation in Table II, which improves the accuracy in both unsliced and sliced code by 37.4% and 40.43%, respectively. Furthermore, we observe from our study that embedding the control and data flow dependencies via a FAG, assuming that the buggy code contains these aspects, can improve the accuracy even further.

GLANCE is effective for learning code-change patterns. Table V shows a qualitative analysis of the bug patterns found among the correct inferences in GLANCE. As shown, GLANCE is able to fix a wide variety of bug patterns. The authors manually assessed all the correct patches generated by GLANCE, categorized the bug patterns individually and came to a consensus if they agreed on the classification. The assessment and categorization of the patches took approxi-

TABLE V: Qualitative analysis of bugs fixed by GLANCE.

Bug Pattern	Description	Buggy Code	Fixed Code
Wrong Function Name	Wrong function with similar name and signature is called	`let node = document.getElementsById('grand-node'))`	`let node = document.getElementById('grand-node')`
		`assert.deepEqual(x, expected.shift())`	`assert.deepStrictEqual(x, expected.shift())`
Change Unary Operator	Unary expression with wrong unary operator used	`n--;`	`n++;`
		`this.count --;`	`this.count ++;`
Change Binary Operator	Binary expression with wrong binary operator used	`while (num <= i);`	`while (num < i);`
		`for (i = 0; i <= cart.length; i++){`	`for (i = 0; i < cart.length; i++){`
Same Function More Args	Function with missing argument in the buggy line	`table.increments();`	`table.increments('id');`
		`currentUser: service();`	`currentUser: service(user);`
Missing Return Statement	Expression with missing `return` keyword	`React.createElement("div", {`	`return React.createElement("div", {`
		`changeAnimal();`	`return changeAnimal();`
Incorrect Comparison	Expression with incorrect equality comparison	`if (method != 'get'){`	`if (method !== 'get'){`
		`if (Object.keys(req.files).length == 0){`	`if (Object.keys(req.files).length === 0){`
Change Boolean Literal	Expression with incorrect boolean literal used	`db.sequelize.sync({ force: false }).then(()=> {;`	`db.sequelize.sync({ force: true }).then(()=> {;`
		`required: true,`	`required: false,`
Change Identifier Used	Expression with incorrect identifier used	`mongoose.connect(process.env.MONGOLAB_TEAL_URI, {useNewUrlParser: true })`	`mongoose.connect(process.env.MONGODB_URI, {useNewUrlParser: true })`
		`loaders: [`	`rules: [`

mately 10 person-hours in total. For labelling the bug types, we used the bug categories from ManySStuBs4J [70], and Hanam et al. [27] to identify the category of the bug fixes. We present eight bug patterns, with two examples each from the correct inferences by GLANCE in Table V. Among the eight bug patterns, six of them are common one-off bugs pervasive across all languages, and the remaining two patterns are specific to JavaScript [27], namely *Missing Return Statement* and *Incorrect Comparison*. The remaining six bug patterns are examples of simple stupid bugs that commonly occur in almost all programming languages [70], [71]. These bugs are usually caused after refactorings or while copy-pasting lines of code. The bug pattern *Missing Return Statement* occurs in JavaScript because of its dynamic typing, which allows errors of this variant to propagate silently. *Incorrect Comparison* is another JavaScript specific bug type which occurs when strict equality === is not used. The strict equality operator compares both the value and type of the operands whereas non-strict equality == operator tries to coerce the data type of operands during comparison, which may lead to bugs.

Role of control and data flow edges. In GLANCE, we represent the flow-augmented graph as a homogeneous graph and encode it using a Graph Convolutional Network (GCN) for learning to generate bug fixes as described in Section III. Homogeneous graphs contain the same type of nodes, and all edges represent relationships of the same kind. As a result, we learned connectivity information from the control and data flow edges during training. Table III demonstrates that as we embed the program analysis edges over the SEQGRAPH for the unsliced code, training time is reduced from 41 hours 07 minutes to 38 hours 32 minutes. We conjecture that the connectivity information from the control and data flow edges helps the model to converge faster.

Heterogeneous graphs could also be potentially adopted as they are capable of representing relational information with additional type information for the nodes and edges in the graph. In a recent work [72], heterogeneous graphs have proven to improve the performance of tasks such as method name prediction and code classification. As the underlying deep-learning library [50] for developing our GNN model does not support heterogeneous graphs, we have not implemented it in our framework to verify its effectiveness in fixing bugs. However, besides establishing a neighbourhood relationship between code statements and variables with control and data-flow edges, our program analysis edge types can capture information flow, e.g., where a variable is defined, and then the variable is used in a statement with a $UseBy$ edge relationship. It is important to learn edge embedding from control and data flow edge types to use FAG to the fullest extent. In the future, we plan to learn edge embeddings from the types of edges using Relational Graph Convolutional Network (R-GCN) [73], [74].

Limitations. Modern front-end frameworks, such as, ReactJS [75] and VueJS [76] use templated code (e.g. JSX, HTML) to combine markup and UI logic. Our analysis tool does not always extract accurate slices for bugs within front-end templated code. Our tool is currently limited to file-level interprocedural analysis. In the future, we plan to expand GLANCE to handle inter-procedural analysis across the project.

VII. Threats to Validity

Choice of programming languages. We chose JavaScript since it is a popular programming language that has been investigated in previous learning-based program repair papers [7], [11]. The key components of our approach are generic across programming languages: control flow analysis, data flow analysis, and graph neural network. Since we do not rely on language-specific features, our findings should apply to other programming languages, although more experiments are needed in other languages to empirically validate the hypothesis. We chose to evaluate our framework against the state-of-the-art techniques by using a single programming language, as building a slicing tool for each language requires substantial development effort.

Dataset selection. The main external validity threat relates to the quality and generalizability of our dataset. Our model is trained and evaluated on a corpus of the open source GitHub repositories of JavaScript projects. An external validity is that not all datapoints are actual bug fixes, as commits can contain different types of code changes, such as feature additions, refactorings and bug fixes. We used search heuristics to filter buggy commits following prior work [70] to mitigate this threat.

We employ a dataset consisting of a single AST node difference, as in previous work [7]. As a result, we cannot apply benchmarks such as BugsJS [77] for evaluation, as it is not limited to a single AST node difference. However, our dataset selection is consistent with prior work [7], [24]. In the future, we plan to collect even more commits from open source repositories and extend our evaluation.

Evaluation metrics. In this study, we employed top-1 accuracy and BLEU score. However, a program repair tool may generate fix suggestions that do not exactly match the developer-written code but could be functionally equivalent, which these metrics cannot capture. Nevertheless, top-1 accuracy has been used in previous studies [10], [11], [51], [52]. Similarly, BLEU score has also been employed as an evaluation metric in prior studies [4], [56], [57].

Hyperparameter tuning. One approach for hyperparameter optimization could be to tune each code representation. However, due to the large search space of all available hyperparameters, finding an optimal setting can be computationally expensive. The goal of this work is not to find the best setting but rather to compare performance on identical settings. Hence, we opted for tuning the sequence graph context representation that is closest to the actual code and kept the same hyperparameter for other contextual representations.

Reproducibility. We have made available our model and framework [25] in order for the findings to be reproducible.

VIII. Related Work

A wide array of techniques have been proposed for automated program repair, including search-based [78]–[81], constraint-based [82], [83], heuristics-based [84], and template-based [85]–[89]. These techniques have limitations as they require domain-specific knowledge about bug patterns and fixes in a given programming language. Our work pertains to neural program repair and the role of context in source code.

Neural program repair. Learning-based repair techniques that use sequence-based models cannot embed the semantic relationship between program elements as they present source code as a series of tokens [3], [8], [9], [20], [65], [90]. There are attempts [5], [7], [22], [64], [67] to incorporate code structure using AST. For example, Hoppity [7] uses buggy code as AST where context is limited by a pre-defined node limit. Katana [24] applied program slicing and adopted the GNN model from Hoppity. However, recent work [69] has shown that using AST without pre-processing would yield a large input to the model. In contrast, we employ program analysis techniques to explicitly encode the relationship between code elements using control and data-flow information for program repair using a novel FAG representation.

Context in neural source code processing tasks. Different neural code-related tasks attempted to leverage context in varied ways, from ignoring context [19], [20] to using enclosing file [21], class [3], enclosing function [9], [90], [91], surrounding statements [9], or encapsulating AST subtrees [92], [93]. Researchers attempted to learn code embedding from syntactical information (AST) [94]–[96]. Recently, efforts have been made [97] to learn code embedding using data and control flow information for predictive tasks such as method name prediction. There are attempts to encode syntactic and semantic information with an augmented AST [98] for variable naming and misuse. Compared to these works, we investigate how including context using program analysis can aid learning-based repair task.

IX. Conclusion

Contextual information and the relationship that exists in the code play a vital role in the process of understanding and fixing bugs. Existing neural program repair techniques do not consider semantic information pertaining to the buggy line, which can include the relevant statements and the control and data dependencies. In this work, we proposed GLANCE, a framework that leverages contextual information via program analysis and uses a graph-to-sequence learning to generate correct patches. Along with the incorporation of components such as abstraction and bug delineation, and slicing, our technique is able to outperform six existing state-of-the-art program repair techniques with an accuracy of 43.02%. In future work, we plan to apply heterogeneous graphs for the representation learning from our flow-augmented graph. We also plan to expand our framework to support program-wide inter-procedural analysis and other programming languages such as Java. We hope this work inspires the community to apply program analysis techniques to neural models for program repair and other source code processing tasks.

References

[1] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, 2019.

[2] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, 2018.

[3] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "SEQUENCER: Sequence-to-sequence learning for end-to-end program repair," *Transactions on Software Engineering*, pp. 1–1, 2019.

[4] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "DeepDelta: Learning to repair compilation errors," in *Foundations of Software Engineering*. ACM, 2019, p. 925–936.

[5] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-based code transformation learning for automated program repair," in *International Conference on Software Engineering*. ACM, 2020, p. 602–614.

[6] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common c language errors by deep learning," in *AAAI Conference on Artificial Intelligence*. AAAI Press, 2017, p. 1345–1351.

[7] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020.

[8] 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," *Trans. Softw. Eng. Methodol.*, 2019.

[9] 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 *International Symposium on Software Testing and Analysis*. ACM, 2020, p. 101–114.

[10] T. Ahmed, N. R. Ledesma, and P. Devanbu, "SYNSHINE: improved fixing of syntax errors," *Transactions on Software Engineering*, 2022.

[11] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[12] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "CIRCLE: Continual repair across programming languages," in *International Symposium on Software Testing and Analysis*. ACM, 2022, p. 678–690.

[13] E. Mashhadi and H. Hemmati, "Applying CodeBERT for automated program repair of java simple bugs," in *Mining Software Repositories*. IEEE, 2021, pp. 505–509.

[14] 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 *Foundations of Software Engineering*. ACM, 2021, p. 341–353.

[15] Y. Li, S. Wang, and T. N. Nguyen, "DEAR: A novel deep learning-based approach for automated program repair," in *International Conference on Software Engineering*. ACM, 2022, p. 511–523.

[16] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *International Conference on Software Engineering*. ACM, 2022, p. 1506–1518.

[17] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Conference on Program Comprehension*. ACM, 2018, p. 286–296.

[18] S. Aljehane, B. Sharif, and J. Maletic, "Determining differences in reading behavior between experts and novices by investigating eye movement on source code constructs during a bug fixing task," in *ACM Symposium on Eye Tracking Research and Applications*. ACM, 2021.

[19] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

[20] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *arXiv:1812.07170*, 2019.

[21] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *Mining Software Repositories*. ACM, 2020, p. 300–310.

[22] B. Tang, B. Li, L. Bo, X. Wu, S. Cao, and X. Sun, "GrasP: Graph-to-sequence learning for automated program repair," in *International Conference on Software Quality, Reliability and Security*. IEEE, 2021, pp. 819–828.

[23] M. Weiser, "Program slicing," *Transactions on Software Engineering*, vol. 10, no. 4, p. 352–357, 1984.

[24] M. Sintaha, N. Nashid, and A. Mesbah, "Katana: Dual slicing-based context for learning bug fixes," *arXiv preprint arXiv:2205.00180*, 2022.

[25] "GLANCE," https://github.com/neural-repair/glance, 2023.

[26] D. Spinellis, *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley Professional, 2016.

[27] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Foundations of Software Engineering*. ACM, 2016, p. 144–156.

[28] N. Sasirekha, A. E. Robert, and D. M. Hemalatha, "Program slicing techniques and its applications," *arXiv:1108.1352*, 2011.

[29] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Foundations of Software Engineering*. ACM, 2014, p. 306–317.

[30] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *International Conference on Software Engineering*. ACM, 1992, p. 392–411.

[31] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[32] 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," *AI Open*, pp. 57–81, 2020.

[33] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[34] L. Wu, Y. Chen, K. Shen, X. Guo, H. Gao, S. Li, J. Pei, and B. Long, "Graph neural networks for natural language processing: A survey," *arXiv:2106.06090*, 2021.

[35] Y. Wainakh, M. Rauf, and M. Pradel, "IdBench: Evaluating semantic representations of identifier names in source code," in *International Conference on Software Engineering*. IEEE, 2021, pp. 562–573.

[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations, Workshop Track Proceedings*, 2013.

[37] S. Zhang, D. Zheng, X. Hu, and M. Yang, "Bidirectional long short-term memory networks for relation classification," in *Pacific Asia conference on language, information and computation*, 2015, pp. 73–78.

[38] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[39] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference on Learning Representations*, 2015.

[40] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *arXiv:1704.04368*, 2017.

[41] Z. Tu, Z. Lu, Y. Liu, X. Liu, and H. Li, "Modeling coverage for neural machine translation," in *Association for Computational Linguistics*. ACL, 2016, pp. 76–85.

[42] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[43] K. Xu, L. Wu, Z. Wang, Y. Feng, and V. Sheinin, "Graph2Seq: Graph to sequence learning with attention-based neural networks," 2018.

[44] "Node.js," https://nodejs.org/en/, 2022, accessed: 2021-10-17.

[45] L. Sotomayor, "Avoiding javascript scoping pitfalls," https://nearsoft.com/blog/avoiding-javascript-scoping-pitfalls, 2022, accessed: 2022-01-06.

[46] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2019, p. 143–153.

[47] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *International Conference on Software Engineering*. IEEE, 2021, pp. 163–174.

[48] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*, ser. LNCS. Springer, 2012, vol. 7700, pp. 437–478.

[49] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015.

[50] L. Wu, Y. Chen, H. Ji, and B. Liu, *Deep Learning on Graphs for Natural Language Processing*. ACM, 2021, p. 2651–2653.

[51] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *International Conference on Automated Software Engineering*. IEEE, 2021, pp. 443–455.

[52] Y. Xiao, S. Ahmed, W. Song, X. Ge, B. Viswanath, and D. Yao, "Embedding code contexts for cryptographic api suggestion: New methodologies and comparisons," *arXiv:2103.08747*, 2021.

[53] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, “An empirical study on the usage of BERT models for code completion,” in *Mining Software Repositories*. IEEE, 2021, pp. 108–119.

[54] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A method for automatic evaluation of machine translation,” in *Association for Computational Linguistics*. ACL, 2002, p. 311–318.

[55] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, “An empirical study on the usage of transformer models for code completion,” *Transactions on Software Engineering*, pp. 1–1, 2021.

[56] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *International Conference on Software Engineering*. IEEE, 2021, pp. 336–347.

[57] A. Mastropaolo, N. Cooper, D. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Using transfer learning for code-related tasks,” *Transactions on Software Engineering*, no. 01, pp. 1–20, 2022.

[58] Z. Gao, X. Xia, J. Grundy, D. Lo, and Y.-F. Li, “Generating question titles for stack overflow from mined code snippets,” *Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–37, 2020.

[59] R. Shahbazi, R. Sharma, and F. H. Fard, “API2Com: On the improvement of automatically generated code comments using api documentations,” in *International Conference on Program Comprehension*. IEEE, 2021, pp. 411–421.

[60] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *Foundations of Software Engineering*. ACM, 2018, p. 908–911.

[61] “Understand by scitools,” https://www.scitools.com/, 2021, accessed: 2021-12-30.

[62] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “Context-aware patch generation for better automated program repair,” in *International Conference on Software Engineering*. ACM, 2018, p. 1–11.

[63] N. Jiang, T. Lutellier, and L. Tan, “CURE: Code-aware neural machine translation for automatic program repair,” in *International Conference on Software Engineering*. IEEE, 2021, pp. 1161–1173.

[64] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, “CODIT: Code editing with tree-based neural models,” *Transactions on Software Engineering*, pp. 1–1, 2020.

[65] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, “Patching as translation: The data and the metaphor,” in *International Conference on Automated Software Engineering*. ACM, 2020, p. 275–286.

[66] “Circle github repository,” https://github.com/2022CIRCLE/CIRCLE, 2022, accessed: 2022-10-28.

[67] Z. Chen, V. J. Hellendoorn, P. Lamblin, P. Maniatis, P.-A. Manzagol, D. Tarlow, and S. Moitra, “PLUR: a unifying, graph-based view of program learning, understanding, and repair,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 089–23 101, 2021.

[68] W. Zhong, H. Ge, H. Ai, C. Li, K. Liu, J. Ge, and L. Bin, “StandUp4NPR: Standardizing setup for empirically comparing neural program repair systems,” in *International Conference on Automated Software Engineering*. IEEE, 2022.

[69] J. Gu, Z. Chen, and M. Monperrus, “Multimodal representation for neural code search,” in *International Conference on Software Maintenance and Evolution*. IEEE, 2021, pp. 483–494.

[70] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the ManySStuBs4J dataset,” in *Mining Software Repositories*. ACM, 2020, p. 573–577.

[71] A. V. Kamienski, L. Palechor, C.-P. Bezemer, and A. Hindle, “PySStuBs: Characterizing single-statement bugs in popular open-source python projects,” in *Mining Software Repositories*. IEEE, 2021, pp. 520–524.

[72] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, “Learning to represent programs with heterogeneous graphs,” in *International Conference on Program Comprehension*, 2022, pp. 378–389.

[73] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” in *European semantic web conference*. Springer, 2018, pp. 593–607.

[74] A. Tian, C. Zhang, M. Rang, X. Yang, and Z. Zhan, “RA-GCN: Relational aggregation graph convolutional network for knowledge graph completion,” in *International Conference on Machine Learning and Computing*. ACM, 2020, p. 580–586.

[75] “Reactjs,” https://reactjs.org/, 2022, accessed: 2021-10-17.

[76] “Vue.js,” https://vuejs.org, 2022, accessed: 2021-10-17.

[77] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszédes, R. Ferenc, and A. Mesbah, “BugsJS: a benchmark of javascript bugs,” in *IEEE Conference on Software Testing, Validation and Verification*. IEEE, 2019, pp. 90–101.

[78] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A generic method for automatic software repair,” *Transactions on Software Engineering*, vol. 38, pp. 54–72, 2012.

[79] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, “LSRepair: Live search of fix ingredients for automated program repair,” *Asia-Pacific Software Engineering Conference*, pp. 658–662, 2018.

[80] B. Mehne, H. Yoshida, M. Prasad, K. Sen, D. Gopinath, and S. Khurshid, “Accelerating search-based program repair,” *International Conference on Software Testing, Verification and Validation*, pp. 227–238, 2018.

[81] L. Chen, Y. Pei, M. Pan, T. Zhang, Q. Wang, and C. A. Furia, “Program repair with repeated learning,” *Transactions on Software Engineering*, 2022.

[82] F. Wotawa, M. Nica, and I. Nica, “Automated debugging based on a constraint model of the program and a test case,” *The Journal of Logic and Algebraic Programming*, vol. 81, p. 390–407, 2012.

[83] M. Z. Malik, J. H. Siddiqui, and S. Khurshid, “Constraint-based program debugging using data structure repair,” in *International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 190–199.

[84] L. Schramm, “Improving performance of automatic program repair using learned heuristics,” in *Foundations of Software Engineering*. ACM, 2017, p. 1071–1073.

[85] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *International Conference on Software Engineering*. IEEE, 2013, p. 802–811.

[86] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, “ELIXIR: Effective object oriented program repair,” in *International Conference on Automated Software Engineering*. IEEE, 2017, p. 648–659.

[87] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, “Dynamic patch generation for null pointer exceptions using metaprogramming,” in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017, pp. 349–358.

[88] X. Liu and H. Zhong, “Mining stackoverflow for program repair,” in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 118–129.

[89] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “TBar: Revisiting template-based automated program repair,” in *International Symposium on Software Testing and Analysis*. ACM, 2019, p. 31–42.

[90] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, “On learning meaningful code changes via neural machine translation,” in *ICSE*. IEEE, 2019, p. 25–36.

[91] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, “On learning meaningful assert statements for unit test cases,” in *International Conference on Software Engineering*. ACM, 2020, p. 1398–1409.

[92] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *International Conference on Software Engineering*. IEEE, 2019, p. 783–794.

[93] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Deep learning similarities from different representations of source code,” in *Mining Software Repositories*. ACM, 2018, p. 542–553.

[94] Z. Ding, H. Li, W. Shang, and T.-H. P. Chen, “Towards learning generalizable code embeddings using task-agnostic graph convolutional networks,” *Trans. Softw. Eng. Methodol.*, 2022.

[95] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019.

[96] N. D. Bui, Y. Yu, and L. Jiang, “Infercode: Self-supervised learning of code representations by predicting subtrees,” in *International Conference on Software Engineering*. IEEE, 2021, pp. 1186–1197.

[97] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. Le Traon, “Graphcode2vec: Generic code embedding via lexical and program dependence analyses,” in *Mining Software Repositories*. IEEE, 2022, pp. 524–536.

[98] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *International Conference on Learning Representations*, 2018, pp. 520–524.