SEQUENCER: SEQUENCE-TO-SEQUENCE LEARNING FOR END-TO-END PROGRAM REPAIR

Zimin Chen[1]* Steve Kommrusch[2]* Michele Tufano[3]
Louis-Noël Pouchet[2] Denys Poshyvanyk[3] Martin Monperrus[1]

[1]KTH Royal Institute of Technology
Email: {zimin, monp}@kth.se
[2]Colorado State University
Email: {steveko, pouchet}@cs.colostate.edu
[3]The College of William and Mary
Email: {mtufano, denys}@cs.wm.edu

Abstract: This paper presents a novel end-to-end approach to program repair based on sequence-to-sequence learning. We devise, implement, and evaluate a system, called SEQUENCER, for fixing bugs based on sequence-to-sequence learning on source code. This approach uses the copy mechanism to overcome the unlimited vocabulary problem that occurs with big code. Our system is data-driven; we train it on 35,578 samples, carefully curated from commits to open-source repositories. We evaluate it on 4,711 independent real bug fixes, as well on the Defects4J benchmark used in program repair research. SEQUENCER is able to perfectly predict the fixed line for 950/4711 testing samples, and find correct patches for 14 bugs in Defects4J. It captures a wide range of repair operators without any domain-specific top-down design.

1 INTRODUCTION

People have long dreamed of machines capable of writing computer programs by themselves. Having machines writing a full software system is science-fiction but teaching machines modifying an existing program to fix a bug is within the reach of current software technology; this is called automated program repair [21].

Program repair research is very active and dominated by techniques based on static analysis (*e.g.,* Angelix [20]) and dynamic analysis (*e.g.,* CapGen [35]). While great progress has been achieved, the current state of automated program repair is limited to simple small fixes, mostly one line patches. The best techniques are heavily top-down, based on intelligent design and domain-specific knowledge about bug fixing in a given language or a specific application domain. In this paper, we aim at doing program repair in a fairly generic manner, where machine learning only is responsible for capturing the operators useful to repair bugs.

Our key insight is to use sequence-to-sequence learning with copy mechanism for program repair. Sequence-to-sequence learning is a branch of statistical machine learning, mostly used for machine translation: the algorithm learns to translate text from one language (say French) to another language (say Swedish) by generalizing over large amounts of sentence pairs from French to Swedish. The training data comes from the huge amount of text already translated by humans, starting with the Rosetta stone written in 196 BC. The name of the technique is explicit: it is about learning to translate from one sequence of words to another sequence of words.

Now let us come back to the problem of programming: we want to learn to 'translate' from one sequence of program tokens (a buggy program) to a different sequence of program tokens (a fixed program). The training data is readily available: we have millions of commits in open-source code repositories. Yet, we still have major challenges to overcome when it comes to using sequence-to-sequence learning on code: 1. the raw (unfiltered) data is rather noisy; one must deploy significant effort to identify and curate commits that focus on a clear task; 2. contrary to the vocabulary of natural language, software systems' vocabulary is virtually infinite [11]; it encompasses all the programming language keywords (`if`, `for`, `class`), all the numbers ($1, 2, \ldots, \infty$), all the identifier strings, *etc.* 3. in natural language, the dependencies are often in the same sentence ("it" refers to

*Zimin Chen and Steve Kommrusch have equally contributed to the paper as first authors.

"dog" just before) , or within a couple of sentences, while in programming, the dependencies have a longer range: one may use a variable that has been declared dozens of lines before.

We are now at a tipping point to address those challenges. First, sequence-to-sequence learning has reached a maturity level, both conceptually and from an implementation point of view, that it can be fed with sequences whose characteristics significantly differ from natural language. Second, there has been great recent progress on using various types of language models on source code [2]. Based on this great body of work, we present our approach to using sequence-to-learning for program repair, which we created to repair real bugs from large open-source projects written in the Java programming language.

Our end-to-end program repair approach is called SEQUENCER and it works as follows. First, we focus on one-line fixes: we predict the fixed version of a buggy programming line. For this, we create a carefully curated training and testing dataset of one-line commits. Second, we devise a sequence-to-sequence network architecture that is specifically designed to address the two main aforementioned challenges. To address the unlimited vocabulary problem, we use the copy mechanism [28], this allows SEQUENCER to predict the fixed line, even if the fix contains a token that was too rare (*i.e.,* an API call that appears only in few cases, or a rare identifier used only in one class) to be considered in the vocabulary. This copy mechanism works even if the fixed line should contain tokens which were not in the training set. To address the dependency problem, we construct *abstract buggy context* from the buggy class, which captures the most important context around the buggy source code and reduces the complexity of the input sequence. This enables us to capture long range dependencies that are required for the fix.

We evaluate SEQUENCER in two ways. First, we compute a reference accuracy over 4,711 real one-line commits, curated from 3 open-source projects. The accuracy is measured by the ability of the system to predict the fixed line exactly as originally crafted by the developer, given as input the buggy file and the buggy line number. Our golden configuration is able to perfectly predict the fix for 950/4711 (20%) of the testing samples. This sets up a baseline for future research in the field. Second, we apply SEQUENCER to the mainstream evaluation benchmark for program repair, Defects4J. For the 75 one-line bugs from Defects4J, SEQUENCER generates patches which pass the test suite for 19 bugs and patches which are semantically equivalent to the human-generated patch for 14 bugs. To our knowledge, this is the first report ever on using sequence-to-sequence learning for end-to-end program repair, including validation with test cases.

Overall, the novelty of this work is as follows. First, we create and share a unique dataset for evaluating learning techniques on one-line program repair. Second, we report on using the copy mechanism on seq-to-seq learning on source code. Third, on the same buggy input dataset, SEQUENCER is able to produce the correct patch for 119% more samples than the closest related work [33].

To sum up, our contributions are:

- We devise an approach for fixing bugs based on sequence-to-sequence learning on token sequences. This approach uses the copy mechanism to overcome the unlimited vocabulary problem in source code. The input program token sequence are the level of full classes in order to capture long-range dependencies in the fix to be written.
- We implement our approach in an end-to-end program repair tool called SEQUENCER on top of the neural machine translation library OpenNMT based on a state-of-art of encoder-decoder architecture.
- We evaluate our approach on 4,711 real bug fixing tasks. Contrary to the closest related work [32], we do not assume bugs to be in small methods only. Our golden trained model is able to perfectly fix 950/4711 testing samples, which is to the best-of-our knowledge, the best results ever reported on such a task as the time of writing.
- We evaluate our approach on the one-line bugs of Defects4J, which is the most widely used benchmark for evaluating programming repair contributions. SEQUENCER is able to find 2,321 patches, 761 compile successfully, 61 are plausible (they pass the full test suite) and 18 are semantically equivalent to the patch written by the human developer.
- We provide a qualitative analysis of 8 interesting repair operators captured by sequence-to-sequence learning on the considered training dataset.

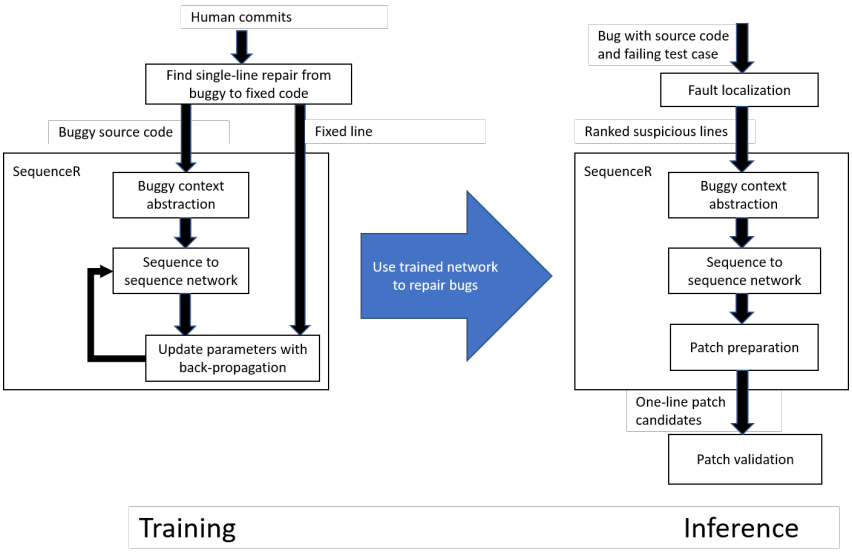


Figure 1: Overview of our approach using sequence-to-sequence learning for program repair.

2 Approach to Using Seq-to-Seq Learning for Repair

SequenceR is a sequence-to-sequence deep learning model that aims at automatically fixing bugs by generating one-line patches. Given a Software System with a faulty behavior (*i.e.,* failing test case), state-of-the-art fault localization techniques are used to identify the buggy method and the suspicious buggy lines. SequenceR then performs a novel ***Buggy Context Abstraction*** (Section 2.2) process which intelligently organizes the fault localization data (*i.e.,* buggy class, method, and line) into a representation that is concise and suitable for the deep learning model yet able to preserve valuable information regarding the context of the bug, which will be used to predict the fix. The representation is then fed to a trained sequence-to-sequence model (Section 2.3.4) which performs ***Patch Inference*** (Section 2.4) and is capable of generating multiple single-lines of code that represent the potential one-line patches for the bug. Finally, SequenceR in the ***Patch Preparation*** (Section 2.5) step generates the concrete patches by formatting the code and replacing the suspicious line with the proposed lines. Figure 1 shows the aforementioned steps both for the training phase (left) and inference phase (right). In the remainder of this section we will discuss the common steps as well as those specific for training and inference.

2.1 Problem Definition

Given a buggy system b^s, and test suite t, we assume a fault localization technique, FL, which identifies a ordered set of bug locations $l = \{l_1, l_2, ...\}$, where each location l_i consists of the buggy class b_i^c, buggy method b_i^m, and the buggy line b_i^l:

$$l = \{loc \mid loc \in FL(b^s, t)\}$$
$$\forall l_i \in l, l_i = \{b_i^c, b_i^m, b_i^l\} \quad \text{and} \quad b_i^l \subset b_i^m \subset b_i^c$$

The problem is to predict (*i.e.,* generate) a fixed line f_i^l, where l_i is the true bug location, such that by replacing b_i^l with f_i^l in b_i^m, the resulting system f^s passes the test suite and the bug is considered fixed. SequenceR tackles this problem by taking as input the fault localization data (*i.e.,* $l = \{l_1, l_2, ...\}$) of a buggy system and attempts to generate fixed line f_i^l for each l_i in order. The b^s, t, l, l_i, b_i^c, b_i^m, b_i^l, f_i^l and f^s notations are used throughout this work.

2.2 Buggy Context Abstraction

The *context* of a bug plays a fundamental role in understanding the faulty behavior and reasoning about the possible fix. During bug-fixing activities, developers usually identify the buggy lines, then

```
1  class Foo {
2    int i;
3    int bar;
4    Foo (int bar){
5      this.bar = bar;
6    }
7    int decrement(){
8      return bar-1;
9    }
10   int increment(){
11     return bar-1;
12   }
13 }
```

Listing 1: Original code

```
1  class Foo {
2    int i;
3    int bar;
4    Foo (int bar){
5    }
6    int decrement(){
7    }
8    int increment(){
9      <START_BUG>
10     return bar-1;
11     <END_BUG>
12   }
13 }
```

Listing 2: *abstract buggy context*

```
1  class <unk> {
2    int i;
3    int <unk>;
4    <unk> (int <unk>){
5    }
6    int <unk>(){
7    }
8    int increment(){
9      <START_BUG>
10     return <unk>-1;
11     <END_BUG>
12   }
13 }
```

Listing 3: Context with <unk>

Figure 2: Illustration of the *abstract buggy context* step in SEQUENCER. b^c is highlighted in yellow, b^m is highlighted in orange and b^l is highlighted in red.

analyze how they interact with the rest of the method's execution, and observe the context (*e.g.,* variables and other methods) in order to reason about the possible fix and possibly select several tokens in the context to build the fixed line.

SEQUENCER mimics this process by constructing the *abstract buggy context* and organizing the fault localization data into a representation that is concise yet retains the necessary context that allows the model to predict the possible fix. During this process SEQUENCER needs to balance two contrasting goals: (i) reduce the buggy context into a reasonably concise sequence of tokens (since sequence-to-sequence models suffer from long sentences [7]), (ii) while at the same time retaining as much information as possible to allow the model to have enough context to predict a possible fix.

Given the bug locations $l = \{l_1, l_2, ...\}$, for each $l_i \in l, l_i = \{b_i^c, b_i^m, b_i^l\}$, SEQUENCER performs the following steps:

Buggy Line The buggy line b_i^l is marked using two special tokens to indicate the start and the end of the line (*i.e.,* <START_BUG> and <END_BUG>). The rationale is that we would like to propagate the information extracted by the fault localization technique and indicate to the model what is a buggy line. In doing so, we mimic developers who focus on the buggy lines during their bug-fixing activities.

Buggy Method The remainder of the buggy method b_i^m is kept in the representation. The rationale is that the method provides crucial information on where the buggy line is placed and its interaction with the rest of the method.

Buggy Class From the buggy class b_i^c we keep all the instance variables and only the signature of the constructor and non-buggy methods (stripping out the body). The rationale for this choice is that the model could use variables and method signatures as potential sources when building the fixed line f_i^l.

After these steps, SEQUENCER performs tokenization and truncation to create the *abstract buggy context*. Truncation will limit the *abstract buggy context* to a predetermined size to improve model training and inference performance.

The *abstract buggy context* represents the input to the sequence-to-sequence network which will be used to predict the fixed line. Internally, *abstract buggy context* is represented as a sequence of tokens belonging to a vocabulary V. The out-of-vocabulary tokens ($token \notin V$) are replaced with the unknown token <unk>. In Section 2.6 we describe how we empirically derive the vocabulary V and in Section 2.3.4 we explain how the copy mechanism helps in overcoming the unknown tokens problem.

Figure 2 shows the output of this process. The original class is presented in Listing 1 and Listing 2 displays the buggy class after Buggy Context Abstraction. Listing 3 illustrates the class when tokens

that are out of vocabulary are replaced with the unknown token <unk>. Our sequence-to-sequence network receives Listing 2 as input.

2.3 SEQUENCE-TO-SEQUENCE NETWORK

In this phase we train SEQUENCER to learn how to generate a fix for a given bug. Specifically, we train a Sequence-to-Sequence Network with Encoder-Decoder model (with attention and copy mechanism) to translate the *abstract buggy context* of a bug to the corresponding target fixed line f_l. To train such network we rely on a large dataset of bug fixes mined from different sources, explained in Section 2.3.1 and Section 2.3.2. The bug fixes are divided into training and testing data, which are used to train and evaluate the Sequence-to-Sequence Network described in Section 2.3.4.

2.3.1 DATASETS

SEQUENCER is trained based on past modifications made to source code, *i.e.,* it is trained on past commits. In our experiments, we combine two sources of past commits, the CodRep dataset [6] and the Bugs2Fix dataset [33], into what appears to be the largest dataset of one-line bug fixes published to date. Both datasets 1) consider Java code and 2) have been built based on the history of open-source projects.

The CodRep dataset exclusively focuses on one-line source code replacements (aka one-line patches). The Bugs2Fix dataset contains diffs mined for bug-fixing commits (based on heuristics to only consider bug-fixing commits).

2.3.2 DATA PREPARATION

Since CodRep and Bugs2Fix datasets are in different formats, we first unify these two datasets as follows. First, we only keep diffs from Bugs2Fix which are fixes with a single-line replacement. Further, we filter out certain diffs if the changes are outside of a method.

Then, we divided the dataset into training and testing data. CodRep is originally split into 5 parts, numbered from 1 to 5, with each part containing commits from different groups of projects. Our training data consists of CodRep datasets 1,2,3 & 5 and the Bugs2Fix dataset. Our testing data is CodRep dataset 4 (or CodRep4 for short), which we use for testing because it contains independent projects from the training data and we want to validate our results over unforeseen projects.

Furthermore, in order to remove overfitting bias, we make sure to remove any sample from the training dataset that is also in the testing dataset. During the model setup, we use a random subset of 95% of the training data for model training and 5% as our validation dataset. The whole process is summarized in the appendix in Figure 12.

2.3.3 DESCRIPTIVE STATISTICS OF THE DATASETS

In total, we have 35,578 samples in our training set and 4,711 sample in our testing set.

Input Size Figure 4 shows the size distribution of the *abstract buggy context* in number of tokens before truncation is done. The CodRep training data has a median token length of 372; the Bugs2Fix dataset has a median length of 340 tokens; and the testing dataset has a median length of 411. These variations are a result of using different Java projects in the datasets, but we observe that the distribution of lengths is similar.

Prediction Size The lines from the *abstract buggy class* samples in our dataset had a median length of 6. 99% of the lines were 30 tokens or fewer, which fits well typical output sizes used for natural language processing (see Figure 13 in the appendix). To sum up, the order of magnitude of the sequence-to-sequence prediction goes from 350 tokens (input sequence) to 6 tokens (output sequence) in average.

Vocabulary Size In our training data, the full vocabulary size is 567,304 different tokens. Figure 3 shows the distribution of the number of occurrences for the whole vocabulary. It is a typical power-law like distribution with a long tail. We limit our training vocabulary to the 1,000 most common tokens.

Figure 3: Overview of vocabulary: token count occurrences follow a Zipf's law distribution.

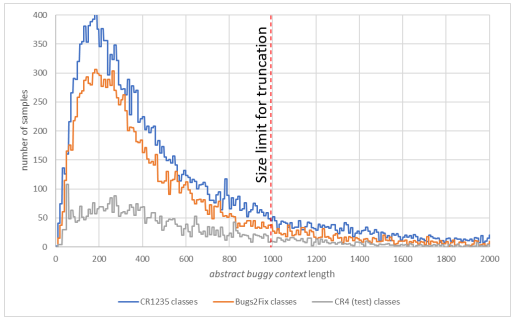


Figure 4: Only 14% of samples exceed the 1K token length limit and require truncation.

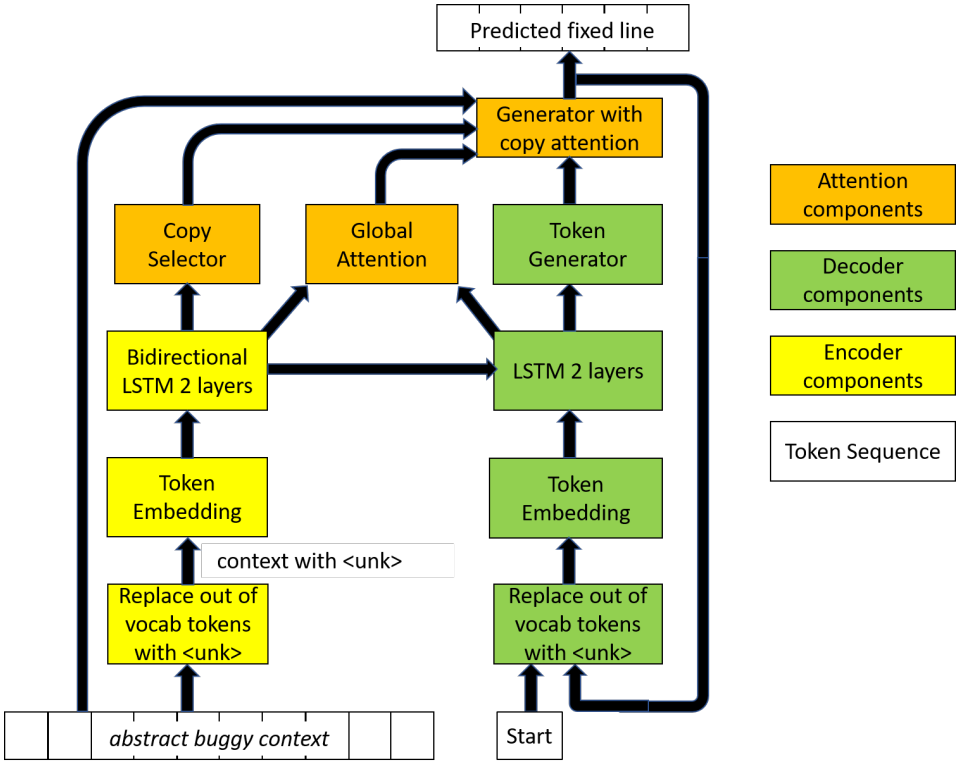


Figure 5: The sequence-to-sequence model used in SEQUENCER.

2.3.4 MODEL

Figure 5 shows our model for sequence-to-sequence learning to create Java source code patches. The basis of our model is a recurrent neural network similar to a natural language processing architecture [31]. During training, the *abstract buggy context* is provided into the encoder. Then, the decoder produces the prediction and back propagation is used to train the parameters in the network with stochastic gradient decent [16].

Encoder The encoder is a recurrent neural network using LSTM gates to process the input [13]. It is a bidirectional encoder which allows the encoding for a token to incorporate information from other tokens both before and after it in the input data [27]. The encoder converts the source sequence $X = [x_1, ..., x_n]$ into a sequence of encoder hidden states h_i using a learnable recurrence function g_e. After reading the last token, the last hidden state, h_n^e is used as the context vector c for use in

initializing the decoder [8]:

$$h_i^e = g_e(x_i, h_{i-1}^e); \tag{1}$$

Decoder The decoder is also a recurrent neural network using LSTM gates. When initialized by the encoder, it begins production of the patch candidate by receiving the special *start* token as input y_0. For each previous output token y_{j-1}, the decoder updates its hidden state h_j^d using the learnable recurrence function g_d [8]:

$$h_j^d = g_d(y_{j-1}, h_{j-1}^d, c) \tag{2}$$

The decoder states h_j^d are used in for token generation by the attention and copy mechanisms in Equation 4 and Equation 5. The model stops updating decoder hidden states and generating new tokens when the last token generated by the model is a special end-of-sequence token.

Attention In addition, we use an attention mechanism that provides a way to create a more specific context vector c_j for each output token y_j from the decoder using a linear combination of the hidden encoder states h_i^e [3]:

$$c_j = \sum_{i=1}^{n} \alpha_i^j h_i^e \tag{3}$$

Where α_i^j represents learnable attention weights. This context vector c_j is used by a learnable function g_a to allow each output token y_j to pay "*attention*" to different encoder hidden states when predicting a token from the vocabulary V:

$$P_V(y_j \mid y_{j-1}, y_{j-2}, ..., y_0, c_j) = g_a(h_j^d, y_{j-1}, c_j) \tag{4}$$

Copy The copy mechanism further contributes to Equation 4 to produce a token candidate. This component calculates p_{gen}, the probability that the decoder generates a token from its initial vocabulary. And $1 - p_{gen}$ is the probability to copy a token from input tokens depending on the attention vector α^j in Equation 3 [28]:

$$p_{gen} = g_c(h_j^d, y_{t-1}, c_j) \tag{5}$$

$$P(y_j) = p_{gen} P_V(y_j) + (1 - p_{gen}) \sum_{i:x_i = y_j} a_i^j \tag{6}$$

g_c in Equation 5 is learnable function. Using Equation 6, the output token y_j for the current decoder state is selected from the set of all tokens that are either: 1. tokens in the training vocabulary (including the <unk> token) or 2. tokens in the *abstract buggy context*.

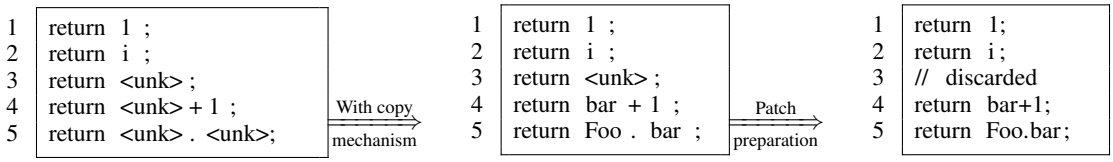
2.4 Patch Inference

Once the sequence-to-sequence network is trained, it can be used to generate patches for projects outside of training dataset. During patch inference, we still generate *abstract buggy context* for the bug, as described in Section 2.2. But we will use beam search to generate multiple likely patches for the same buggy line, as done in related work [33, 1]. Beam search works by only keeping the n best token predictions at each decoder state, and only these will be expanded at the next decoder state. n is often called beam width or beam size, and beam search with infinite n corresponds of doing a complete breath-first-search. In Listing 5 we have an example of predictions with beam size 5 for the bug presented in Listing 2. The output from this step will be processed by the patch preparation step.

2.5 Patch preparation

The raw output from the sequence-to-sequence network cannot be used as a patch directly: First, the predictions might still contain <unk> tokens, not handled by the copy mechanism; Second, the predictions contain a space between every token, which is not well-formed source code in many cases. (For example, a space is not allowed between the dot separator, ".", and a method call, but a space is required between a type and the corresponding identifier name.)

Consequently, we have a final patch preparation step as follows. We discard all line predictions that contains <unk> and we reformulate the remaining predictions into well-formed source code by



Listing 4: Without copy mechanism

Listing 5: With copy mechanism

Listing 6: After patch preparation

Figure 6: Illustration of the patch preparation step and the copy mechanism

removing or adding the required spaces. An example is shown between Listing 5 and Listing 6, whitespaces are adjusted and the third prediction from Listing 5 is removed since it contains <unk> token.

The remaining candidate fixed lines, $cand_i = \{pre_i^1, pre_i^2, ..\}$, will replace the buggy line b_i^l in buggy system b^s and generate candidate patches $\{patch_i^1, patch_i^2, ...\}$, which should be verified with any patch validation technique, such as test suite validation. When the test suite is weak to specify the bug, we can have different patches $\{patch_i^1, patch_j^1, ...\}$ for different bug locations $\{l_i, l_j, ...\}$ that passed the test suite. Then, the correctness can be verified by, for example, manual inspection.

2.6 Implementation Details & Parameter Settings

Library. We have implemented our Encoder-Decoder model using OpenNMT-py [17], built in the Python programming language and the PyTorch neural network platform [22].

Vocabulary. In this paper, we consider a vocabulary of the 1,000 most common tokens. To the best of our knowledge, this is one of the largest vocabularies considered for machine learning for patch generation: for comparison, DeepFix [9] has a vocabulary size of 129 words, and Tufano *et al.* [33] considered a vocabulary size of 430 words.

Limit for truncation. We truncate if the *abstract buggy context* is longer than 1,000 tokens. It is motivated by Figure 4, where we can see the most of *abstract buggy context* are less than 1,000 tokens long. SequenceR truncates by removing statements, class definitions, and method definitions until *abstract buggy context* is 1,000 tokens or less, but keeping the buggy line within the truncated buggy class.

Network parameters. We explored a variety of settings and network topologies for SequenceR. Most major design decisions are verified with ablation experiments that change a single variable at a time as detailed further in Section 4. We train our model with a batch size of 32 for 10,000 iterations. To prevent overfitting, we use a dropout of 0.3. In relation to the components shown in Figure 5, below are the primary matrix sizes associated with each component:

- Token embedding: 1,004x256 (1,000 tokens + 4 special tokens)
- Encoder bidirectional LSTM: 256x256x4x2x2
- Decoder LSTM: 512x256x4x2 + 256x256x4x2
- Token generator: 256x1004
- Bridge between encoder and decoder: 256x256x2
- Global Attention: 256x256 + 512x256
- Copy selector: 256x1

We use a beam size of 50 during inference, which is the default value used in the literature [33, 1] and which proves to be good empirically.

Usage After SequenceR is trained, we can use it to predict fixes to a bug. SequenceR takes as input the buggy file and a line number indicating where the bug is. The output is a list of patches in the diff format, so that the user can run their own patch validation step, which could either be test validation or manual inspection.

The source code of SEQUENCER is available at `https://github.com/kth/SequenceR`, together with the best model we have identified and the synthesized patches.

3 EVALUATION

In this section, we describe our evaluation of SEQUENCER.

3.1 RESEARCH QUESTIONS

The two first research questions focus on machine learning:

- RQ1: To what extent can the fixed line be perfectly predicted?
- RQ2: To what extent does the copy mechanism overcome the unlimited vocabulary of source code?

The last two research questions look at the system from a domain-specific perspective: we assess the performance of SEQUENCER from the viewpoint of program repair research.

- RQ3: How effective is SEQUENCER's sequence-to-sequence learning in fixing bugs in the well-established Defects4J benchmark?
- RQ4: What repair operators are captured with sequence-to-sequence learning?

3.2 EXPERIMENTAL METHODOLOGY

3.2.1 METHODOLOGY FOR RQ1

We train SEQUENCER with the parameter settings described in Section 2.6. The training and validation accuracy and perplexity will be plotted. Perplexity (ppl) is a measurement of how well a model predicts a sample and is defined as:

$$ppl(X, Y) = exp(\frac{-\sum_{i=1}^{|Y|} \log P(y_i \mid y_{i-1}, \ldots, y_1, X)}{\mid Y \mid})$$

where X is the source sequence, Y is the true target sequence and y_i is the *i*-th target token [17]. Luong *et al.* found a strong correlation between low perplexity and high translation quality [18].

The resulting model is tested on our testing dataset, CodRep4 (see Section 2.3.2). Next, in order to compare SEQUENCER against the state-of-the-art approach by Tufano *et al.* [33], we created CodRep4Medium. It is a subset of CodRep4 containing 1,116 samples where the buggy method length is limited to 100 tokens.

3.2.2 METHODOLOGY FOR RQ2

To evaluate the effectiveness of the copy mechanism (described in Section 2.3.4), we consider all samples from CodRep4. For each successfully predicted line, we categorize tokens in that line based on whether the token is in the vocabulary or not. And at the same time, for tokens that are out-of-vocabulary but are copied from the input sequence, we try to find the original location of the copied token.

3.2.3 METHODOLOGY FOR RQ3

We evaluate SEQUENCER on Defects4J [15], which is a collection of reproducible Java bugs. Most recent works in program repair research on Java use Defects4J as an evaluation benchmark [19, 39, 38, 35, 14].

Since the scope of our paper is on one-line patches, we first focus on Defects4J bugs that have been fixed by developers by replacing one single line (there are 75 such bugs). In order to study the effectiveness of sequence-to-sequence itself, we isolate the fault localization step as follows: the input to SEQUENCER is the actual buggy file and the buggy line number. SEQUENCER then

Approach	Prediction Accuracy	
	CodRep4Medium	CodRep4
simple seq2seq line2line, no copy	77/1116 (6.9%)	206/4711 (4.4%)
Tufano *et al.* [33]	157/1116 (14.1%)	N/A
SEQUENCER	344/1116 (30.8%)	950/4711 (20.2%)

Table 1: Comparison with state-of-the-art approach by Tufano *et al.*

produces a list of patches (recall that beam search produces several candidate patches). All patches are compiled and then executed against the test suite written by the developer.

Each candidate patch generated by SEQUENCER is then categorized as follows:

- **Compilable patch**: The patch can be compiled.
- **Plausible patch**: The patch is compilable and passes the test suite. It is a plausible candidate patch, yet still possibly incorrect because of the overfitting problem [30].
- **Correct patch**: The patch passes the test suite, and is semantically equivalent to the human patch. We hand-check for semantic equivalence for this evaluation.

As per the definitions, there is a strict inclusion structure in those categories: correct patches are necessarily plausible and compilable, plausible patches are necessarily compilable.

3.2.4 METHODOLOGY FOR RQ4

For RQ4, we aim at having a qualitative understanding of the cases for which our sequence-to-sequence repair approach works. For this, we use a mixed method combining grounded theory and targeted analysis. For the grounded theory, we have been regularly sampling successful cases, *i.e.,* cases in our testing dataset CodRep4 for which SEQUENCER was able to predict the fixed line, for each case, the authors reached a consensus to know whether 1) the case is interesting from a programming perspective, and 2) the case highlights a phenomenon that has already been covered in a previously found case. For the targeted analysis, we specifically searched for 3 kinds of results: cases where the copy mechanism was used and cases where a specific programming construct was involved (method call, field reference and string literals).

3.3 EXPERIMENTAL RESULTS

3.3.1 ANSWER TO RQ1: PERFECT PREDICTIONS

We trained our model on a GPU (Nvidia k80) for 1.2 hours. For a typical training run on our golden model, Figure 7 shows the training and validation accuracy per token generated (the accuracy for the entire patch would be lower) and Figure 8 shows the perplexity (ppl) per token generated over the training and validation datasets. In this particular run, the best results for both the perplexity and accuracy on the validation dataset occur at 10,500 iterations. We chose 10,000 iterations as the standard training time for our model.

CodRep4 On the 4,711 prediction tasks of our best model, SEQUENCER is able to generate the perfect fix in 950 cases (from Table 1). In all those cases, the predicted line that replaces the buggy line is exactly the line fix implemented by the developer. The copy mechanism is used in a number of cases, this will be further discussed in subsubsection 3.3.2.

Comparison to state-of-the-art To the best of our knowledge, the state-of-the-art approach is from Tufano *et al.* [33]. However, this approach is limited to fixes only inside small methods, consisting of less than 100 tokens. SEQUENCER does not make any assumption on the size of the buggy method. In order to compare against [33], we select those 1,116 tasks from CodRep4 where the buggy line resides in a method smaller than 100 tokens. Those 1,116 tasks are called the CodRep4Medium testing dataset.

Our testing accuracy for both CodRep4 and CodRep4Medium are shown in Table 1. From the table, we see that the accuracy of SEQUENCER is 344/1116 (30.8%) while [33] is 157/1116 (14.1%). This is a clear indicator that SEQUENCER outperforms the current state-of-the-art showing twice as many

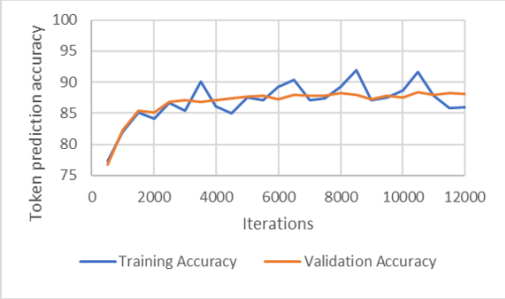


Figure 7: Training and validation accuracy

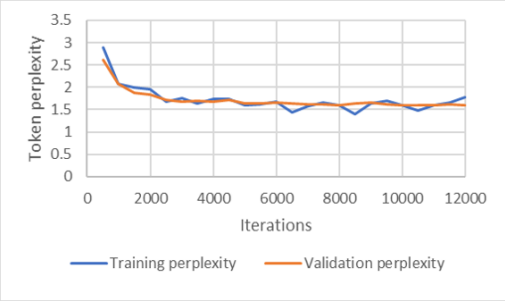


Figure 8: Training and validation perplexity

correct predictions. It shows that our construction of the *abstract buggy context*, together with the copy mechanism, leads to higher accuracy than only having the buggy method as context with a specific encoding for variables.

We now concentrate on the effectiveness of the approach depending on the buggy method length. Overall, we observe that SEQUENCER has a lower accuracy on longer methods (30.8% accuracy on CodRep4Medium, 20.2% accuracy on CodRep4). This phenomenon is explained by the fact that fixes in long methods are usually more complex and involve more context variables, identifiers and literals that are not easily captured by the learning system. This phenomenon has also been previously observed [33].

3.3.2 ANSWER TO RQ2: COPY MECHANISM

We now look at to what extent the copy mechanism is used. Figure 9 shows the origin of tokens in successfully predicted lines, per patch size. Let us consider the highest bar, corresponding to all successfully predicted lines consisting of 7 tokens. For those 7-token patches, the blue bar means that all tokens are taken from the vocabulary. The non-blue bars means that the copy mechanism has been used to predict the line fix. Overall, there is a minority of patches (216/950, 23%) for which all tokens come from the vocabulary. Also, the longest successful patch generated by SEQUENCER was 68 tokens long, but the longest successful patch without the copy mechanism was only 27 tokens long.

Figure 9 also lets us analyze the location origin of the copied token. The orange bars represent those patches for which copied tokens all come from the buggy line: this is the majority of cases (641/950, 68%). However, we also observe cases where some copied tokens have been taken from the buggy method (gray bars) and cases where the copied tokens has been taken from the buggy class, *i.e.,* taken from the class context as captured in our encoding.

As an example, Listing 7 replaces masterNode with nonMasterNode as the correct human-generated patch. SEQUENCER was able to generate this patch because even though the token 'nonMasterNode' is not seen in the training dataset, it can be copied from within the buggy method for use in patch generation. As this example is a 4 token long patch, it would contribute to the gray bar for patch length 4 in Figure 9.

```
while( nonMasterNode == null ) {
  nonMasterNode = randomFrom( internalCluster() .getNodeNames() );
  if ( nonMasterNode.equals( masterNode ) ) {
-       masterNode = null ;
+       nonMasterNode = null;
  }
}
```

Listing 7: Example of the copy mechanism creating a correct patch by incorporating a variable which is not in the vocabulary from the broader context around the buggy line.

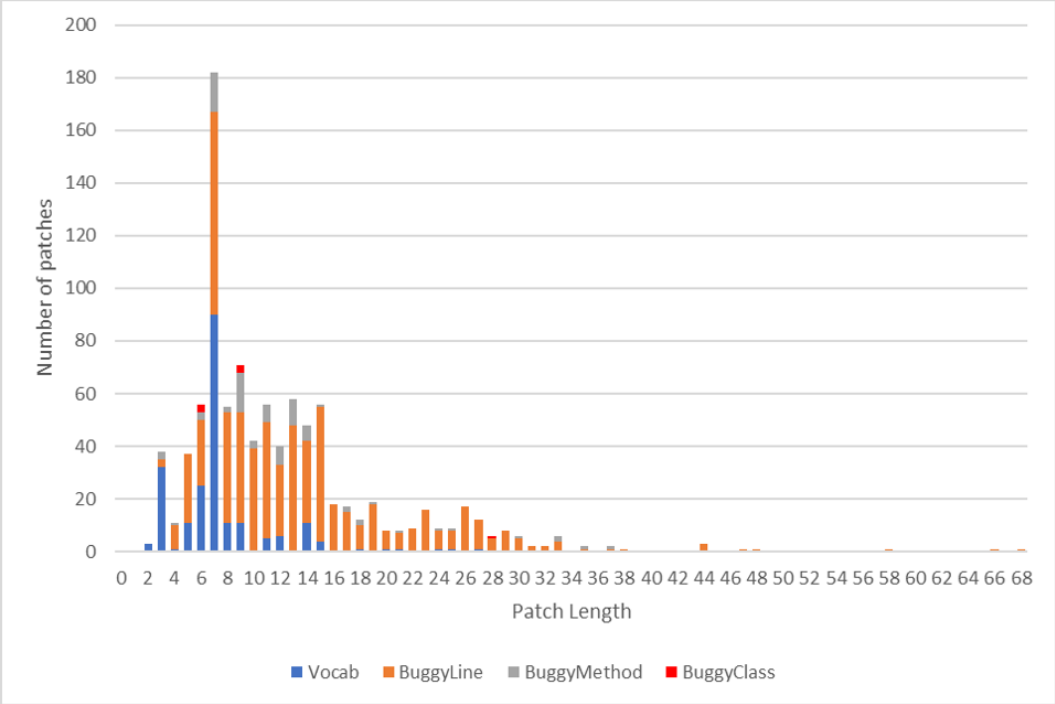


Figure 9: Histogram showing correctly generated patches: 1) that only use tokens in our 1,000 token vocabulary, 2) that need to copy tokens from the buggy line, 3) from the buggy method and 4) from the buggy class.

Overall, Figure 9 shows that the copy mechanism is extensively used (734/950, 77%) and that our class level abstraction enables us to predict difficult cases where only the buggy line or the buggy method would not have been enough.

In order to understand the benefits of context size with the copy mechanism, we measured the distance in tokens to reach a copied token used to generate a patch. In the 87 cases where a copied token was needed from the buggy method b_m, the median distance from the buggy line b_l to the nearest use of the copied token was 9 tokens, 90% of the 87 cases were within 49 tokens of b_l, and 100% were found within a 122 token distance. In the 7 cases when a copied token was needed from the buggy class b_c, the median distance to the copied token from b_m was 25 tokens, and 100% were found within a 241 token distance. In addition to ablation study results discussed is Section 4, the preceding data supports our decision to create the *abstract buggy class*. (Figure 14 and Figure 15 in the appendix show detailed results for these distance measures).

3.4 Answer to RQ3: Defects4J Evaluation

As explained in Section 3.2.3, we consider 75 Defects4J bugs that have been fixed with a one-line patch by human developers. In total SequenceR finds 2,321 patches for 58 of the 75 bugs. The main reason that we are unable to fix the remaining 17 bugs is due to fact that some bugs are not localized inside a method, which is a requirement for the fault localization step that SequenceR assumes as input. We have 2,321 patches instead of 2,900 (58x50) because some predictions are filtered by the patch preparation step (Section 2.5), *i.e.,* patches that contain the <unk> token. The statistics about all bugs can be found in Figure 10. Out of 75 bugs, SequenceR successfully generated at least one patch for 58 bugs, 53 bugs have at least one compilable patch, 19 bugs have at least one patch that passed all the tests (*i.e.,* are plausible) and 14 bugs are considered to be correctly fixed (semantically identical to the human-written patch). Of these 14 bugs, in 12 cases the plausible patch with the highest ranking in the beam search results was the semantically correct patch.

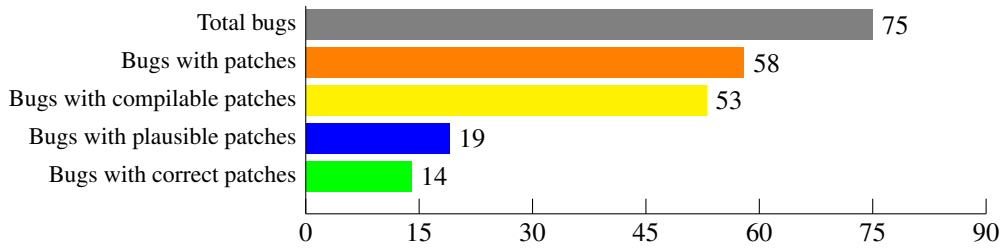


Figure 10: SEQUENCER results on the 75 one-line Defects4J bugs.

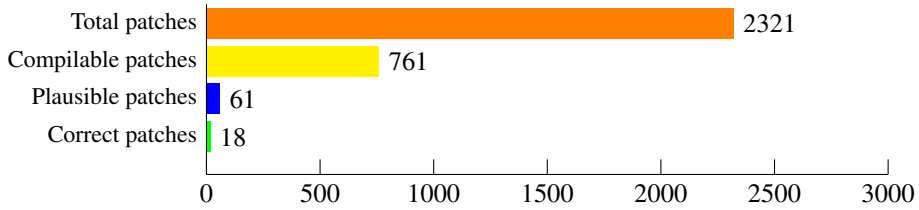


Figure 11: Stastistics on patches synthesized by SEQUENCER for the 75 one-line Defects4J bugs.

Figure 11 gives a different perspective on this data, focusing on patches (and not bugs). SEQUENCER is able to generate 761 compilable patches (33% of all patches). The 61 plausible patches are spread over 19 bugs, which means that there can be several plausible patches for the same bug, a phenomenon well-known in the program repair field [19]. One reason is that some Defects4J bugs have a weak test suite. To the best of our knowledge, we are the first to report the plausibility of patches generated by machine learning approaches. In the end, after validation and patch equivalence check, SEQUENCER is able to generate 18 patches that are semantically equivalent to be correct bug fix.

For SEQUENCER applied to Defects4J bugs, we observe that out of 61 plausible patches, 18 are correct, which is a ratio of 30%. A careful analysis of prior techniques (GenProg, RSRepair, and AE) shows that they have correct/plausible ratios of less than 12% [24]. Such a high ratio is evidence that SEQUENCER has learned to produce outputs which represent reasonable patch proposals, and patches which pass all tests are worth considering for approval by human developers.

Listing 8 shows the SEQUENCER patch for Math 75, which is semantically equivalent to the human patch. We observe that it contains some unnecessary parentheses, and the same behavior can be observed for several other patches found by SEQUENCER. However, in this case, the parentheses does not change the order of evaluation. Therefore the SEQUENCER patch for Math 75 is semantically equivalent to the human patch.

Interestingly, `getPct` is not part of the vocabulary, and it did not appear in the buggy method. The `getPct` method is defined in the same buggy class, as captured by our *abstract buggy context*. In Defects4J, the copy mechanism is also useful to capture the right tokens to add in the patch.

```
- return  getCumPct((Comparable<?>) v);
+ return  getPct (((Comparable<?>)(v)));
```

Listing 8: Found patch for Math 75

We now compare those results against the patches found by recent program repair tools that are publicly available. Elixir [25], CapGen [35] and SimFix [14] have reported 26, 21, 34 correctly repaired bugs for all Defects4J bugs, where the patch is identical to the human patch or claimed as correct. Of those correctly repaired bugs, 22, 19 and 17 respectively are for the 75 one-line bugs that we consider for SEQUENCER. We notice that the majority of claimed correct patches are for one-line bugs. We observe that SEQUENCER does not fix more one-line Defects4J bugs. However, our goal is different: while those approaches are driven with intelligent design and require substantial configuration and handcrafted rules, our goal with SEQUENCER is to be agnostic and to *not* design any repair operator upfront. To that extent, it is remarkable that such a generic approach is able to

learn bug-fixing patterns and synthesizes 18 patches that are semantically equivalent to the human repair, without any static or dynamic analysis.

3.5 Answer to RQ4: Qualitative Case Studies

We now present the diversity of repair operators that are captured by SequenceR. We also highlight again the effectiveness of the copy mechanism by using a **bold underlined** font for those tokens that were copied (*i.e.,* that are outside the vocabulary of the 1,000 most common tokens).

3.5.1 Case study: method call change

Our training and evaluation data consists of object-oriented Java software. We observe that SequenceR captures different kinds of operations related to method calls.

Call change Here a call to method **writeUTF** is replaced by a call to method **writeString**.

```
− out.writeUTF( failure );
+ out.writeString( failure );
```

Listing 9: Call change

Call deletion The buggy line chains two method calls; this successful prediction consists of deleting one of them.

```
− FieldMappers x = context.mapperService().smartNameFieldMappers( fieldName );
+ FieldMappers x = context.smartNameFieldMappers( fieldName );
```

Listing 10: Call deletion.

Argument addition In this patch, SequenceR adds an argument (which in Java, means calling another method).

```
− stage.getViewport().update( width, height );
+ stage.getViewport().update( width, height, true );
```

Listing 11: Argument addition

Target change

In this successful case, the patch also calls method **isTerminated** but on another target (**scheduledExecutorService** instead of **executorService**, which is copied from the input context).

```
− if( !( executorService.isTerminated() ) ){
+ if( !( scheduledExecutorService.isTerminated() ) ){
```

Listing 12: Target change

3.5.2 Case study: if-condition change

SequenceR can change if conditions, and in this particular case, removes two clauses from the boolean formula.

```
− if( ( ( t >= 0 ) && ( t <= 1 ) ) && ( intersection != null ) )
+ if( intersection != null )
```

Listing 13: if-condition change

3.5.3 Case study: Java keyword change

SequenceR is also able to generate patches involving the replacement of programming language keywords, indicating clues of syntax understanding.

```
- break ;
+ continue ;
```

Listing 14: Java keyword change

3.5.4 CASE STUDY: CHANGE FROM FIELD ACCESS TO METHOD CALL

A good practice of software engineering is to implement encapsulation by calling methods instead of directly accessing fields, this is handled by SEQUENCER as follows (`size` to `size()`)

```
- app.log( "PixmaPackerTest", ( "Number of textures: " + ( atlas.getTextures().size ) ) );
+ app.log( "PixmaPackerTest", ( "Number of textures: " + ( atlas.getTextures().size() ) ) );
```

Listing 15: change from field access to method call

3.5.5 CASE STUDY: OFF-BY-ONE REPAIR

Finally, SEQUENCER is also able to repair classical off-by-one errors.

```
- nextIndex = currentIndex;
+ nextIndex = ( currentIndex ) - 1;
```

Listing 16: off-by-one repair

Overall, SEQUENCER uses all three kinds of token operations: 1. Token deletion, *e.g.,* Listing 10; 2. Token addition, *e.g.,* Listing 11; 3. Token replacement, *e.g.,* Listing 9.

4 ABLATION STUDY

We perform an ablation study to understand the relative importance of each component of our approach. The process is as follows. First, we identify the golden model based on a greedy optimization in the parameter search space. This is the model that we described in section 3. Then we change one single parameter to a reasonably different value and report the performance on the same testing dataset. The ablation results demonstrate that parameter selections for the golden model produce the highest acceptance rates for the configurations we tested. Due to randomness in learning, for each parameter, we run each configuration 4 times and report the best run out of 4[1]. Due to computational constraints, we did not gather more runs for each configuration.

First, we consider the very coarse grain features. Table 2 shows the performance of four models, starting from a simplistic seq-to-seq model that only takes a single buggy line b_l as input when learning to produce the fixed line f_l. Then we show beam search, copy, and the use of the *abstract buggy context* improving the model performance. These results confirm our answer to RQ2 that the copy mechanism is essential to the performance of the system.

Second, Table3 shows the results of our 'Golden model' against the results of single specific, targeted changes made to the model. Ablation ID 1 shows that our 10K training limit is sufficient given our training data. ID 2 shows that a vocabulary smaller than 1K tokens performs worse - likely due to a loss of learned tokens that can be used even if an instance of the token is not in the *abstract buggy context*. ID 3 shows that a vocabulary larger than 1K tokens performs worse - perhaps due to the additional tokens having insufficient training examples for learning a proper embedding. ID 4 is about an original idea: in order to provide more opportunities to learn a quality embedding, we created unsupervised pretraining data for the encoder/decoder. Using this unsupervised data did not improve the model, it worsens it.

[1]For example, if model A results in a random score uniformly distributed between 0 and 100 when tested and model B results in a score between 0 and 200, a single test of each model is 25% likely to observe model A scoring better than model B. But if the best result of 4 tests is used, model A would score better than model B only 3% of the time.

Model description	CR4Full	ratio
50K vocab, no copy, beam size 1, no context	55	baseline
50K vocab, no copy, beam size 50, no context	206	3.7x
1K vocab, copy, beam size 50, no context	826	15.0x
Golden Model (includes *abstract buggy context*)	950/4711	17.3x

Table 2: Performance impact of the key features of beam size, copy, and context.

ID	Model description	CR4Full	change
0	Golden Model	950/4711	—
1	more training iterations (20K vs 10K)	901	-5%
2	smaller token vocabulary (700 vs 1000)	882	-7%
3	larger token vocabulary (1400 vs 1000)	905	-5%
4	with unsupervised pretraining	856	-10%
5	less training data (CodRep vs CR+Bugs2Fix)	810	-15%
6	no bridge layer from encoder to decoder	905	-5%
7	fewer LSTM layers on enc/dec (1 vs 2)	438	-54%
8	more LSTM layers on enc/dec (3 vs 2)	859	-10%
9	fewer LSTMs per layer (128 vs 256)	886	-7%
10	more LSTMs per layer (512 vs 256)	889	-6%
11	without context (only buggy line as input)	826	-13%
12	no truncation of *abstract buggy context*	crash	
13	truncate to larger context (4K vs 1K)	950	-0%
14	truncate to smaller context (500 vs 1K)	890	-6%
15	remove <START_BUG> and <END_BUG> tokens	356	-63%

Table 3: Results with selected configurations in the parameter neighborhood of the golden model.

ID 5 shows the value of combining the CodRep and Bugs2Fix data sets to improve the generalization of the model. ID 6 demonstrates the benefit of a using bridge between the encoder and decoder, perhaps allowing the encoder's hidden states to optimize more for the attention network.

IDs 7 through 10 demonstrate that our LSTM network is sized correctly; presumably a smaller network cannot generalize on the model data well enough whereas a larger network has too many degrees of freedom. Our speculation is that a 2 layer encoder/decoder network allows the layer connected directly to the token embedding to 'focus' the weight matrix on input syntax while the layer connected to the attention/copy mechanism 'focuses' on output generation. ID 11 shows the loss in accuracy when *abstract buggy context* is reduced to just the buggy line.

ID 12 shows that truncation is necessary otherwise an out-of-memory error crashes the system, due to too many time steps being stored in memory per token in the sequence. ID 13 shows that if we truncated to 4,000 tokens then the system passes, but the increased context size (4,000 vs the golden model 1,000) did not improve accuracy of the model. ID 14 shows that using a 500 token limit for *abstract buggy context* hurts accuracy presumably because there are less opportunities for token copy. We also speculate that a possible advantage of 1K truncation instead of 500 could be that 1K provides a type of unsupervised learning for the encoder hidden states, the global attention, and the copy mechanism.

ID 15 removes the <START_BUG> and <END_BUG> tokens from the *abstract buggy context* input. The target output is still the correct single-line patch. Without these labels, SEQUENCER must learn line break positions and learn a type of fault localization in order to create a valid patch. As expected, there is a significant accuracy loss, but the network was still able to create 356 correct patches.

5 RELATED WORK

The work presented here is on built on top of two big and active research fields: program repair and machine learning on code. We refer to recent surveys for getting a good overview on them: [21] for program repair and Allamanis *et al.*'s [2] for the latter. In the following, we focus on those works that are about learning and automatic repair.

sk_p is a program repair technique for syntactic and semantic errors in student programs submitted to MOOCs [23]. First, it uses the previous and next statement to predict the statement in the middle, *i.e.,* to replace the current statement. The probability of a patch is the product of the probabilities for all chosen statements. As we do, sk_p uses beam search to produce the top n predictions.

[4] also repairs MOOC student submissions in Python by combining learning and sketch-based synthesis. The approach by Wang *et al.* [34] considers MOOC but the technique itself is completely different: [34] does deep learning on program traces in order to predict the kind of bug affecting a student submission. The main differences between those works and ours are that 1) we consider a larger context (the buggy class) and 2) we consider real programs for training and testing that are bigger and more complex than student's submissions. Shin *et al.* [29] consider simple programs in the educational programming language Karel. As SEQUENCER, their system predicts to delete, insert or replace tokens. Henkel *et al.* [12] compute an embedding for symbolic traces and perform a pilot experiment for fixing error-handling code, which is very different from concrete bug fixing as we do here.

DeepFix is a program repair tool for fixing compiler errors in introductory programming courses [9]. The input is the whole program, (100 to 400 tokens long for their data), and the output is a single line fix. The vocabulary size is set to 129, which was enough to map every distinct token type to a unique word in the vocabulary. TRACER is another program repair tool for fixing compiler errors which outperforms DeepFix in terms of success rate [1]. Santos *et al.*'s [26] further refines the idea and evaluates it with an even larger dataset. The focus of those three works and ours is very different, they focus on compiler errors, we focus on logical bugs. For compiler errors, one does not need to consider the whole vocabulary, but only token types. On the contrary, we have to address this problem and we do so by using the copy mechanism.

DeepRepair [37] is an early attempt to integrate machine learning in a program repair loop. DeepRepair leverages learned code similarities, captured with recursive autoencoders [36], to select repair ingredients from code fragments that are similar to the buggy code. Our usage of learning is different, DeepRepair uses machine learning to select interesting code, SEQUENCER uses machine learning to generate the actual patch.

Tufano *et al.* investigated the feasibility of using neural machine translation for learning bug-fixing patches via [32, 33]. The authors first perform a source code abstraction process that relies on a combination of Lexer+Parser which replaces identifiers and literals in the code. The goal of this abstraction is it reduce the vocabulary while keeping the most frequent identifiers/literals. Then, they train a NMT model to translate an entire buggy method to the corresponding fixed method. SEQUENCER is different in the following ways. First, we consider the entire context of the buggy class, rather than only the buggy method in order for the model to use more tokens when predicting the fix, those with long-range dependencies. Second, our abstraction process uniquely utilize the copy mechanism (which they do not), which allows SEQUENCER to utilize a larger set of tokens when generating the fix. Beyond those two major qualitative differences, a quantitative one is that they only consider small methods, no longer than 100 tokens, while we have no such restriction.

Parallel work by Hata *et al.* [10] discusses a similar network architecture, also applied to one-line diffs. The major differences between [10] and our work are the following: First, they do project-specific training, which means that their approach is only evaluated on testing data coming from the same project. On the contrary, we do global training and we show that SEQUENCER captures repair operators applicable to any project. Our qualitative case studies are unique with that respect. Second, they only look at wellformedness of the output, while we also compile and execute the predicted patch. Our work is an end-to-end test-suite based repair approach. Third, their input is limited to the precise buggy code to replace, while SEQUENCER uses *abstract buggy context*, which allows for a broader set of tokens for the copy mechanism to select from.

6 CONCLUSION

In this paper, we have presented a novel approach to program repair, called SEQUENCER, based on sequence-to-sequence learning. Our approach uniquely combines an encoder/decoder architecture with the copy mechanism to overcome the problem of large vocabulary in source code. On a testing dataset of 4,711 tasks taken from projects which were not in the training set, SEQUENCER is able to successfully predict 950 changes. On Defects4J one-line bugs, SEQUENCER produces 61 plausible, test-suite adequate patches. To our knowledge, our paper is the first ever to show the effectiveness of the copy mechanism on source code, in order to alleviate the unlimited vocabulary problem.

This work opens promising research directions. First, we will explore to see if SEQUENCER also works for more complex mutli-line patches. Second, there is some preliminary work on tree-to-tree transformation learning [5], which conceptually is very appropriate for code viewed as parse trees. Such techniques may augment or supersede sequence-to-sequence approaches. Finally, the originality of our context abstraction is to capture class-level, long range dependencies: we will study whether such a network architecture is able to capture dependencies beyond that, at the package or application level.

REFERENCES

[1] Umair Z Ahmed et al. "Compilation error repair: for the student programs, from the student programs". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM. 2018, pp. 78–87.

[2] Miltiadis Allamanis et al. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 81.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[4] S. Bhatia, P. Kohli, and R. Singh. "Neuro-Symbolic Program Corrector for Introductory Programming Assignments". In: *2018 IEEE/ACM 40th International Conference on Software Engineering*. 2018, pp. 60–70.

[5] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. "Tree2Tree Neural Translation Model for Learning Source Code Changes". In: *arXiv* abs/1810.00314 (2018).

[6] Z. Chen and M. Monperrus. "The CodRep Machine Learning on Source Code Competition". In: *ArXiv e-prints* (July 2018). arXiv: `1807.03200 [cs.SE]`.

[7] KyungHyun Cho et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *CoRR* abs/1409.1259 (2014). arXiv: `1409.1259`.

[8] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[9] Rahul Gupta et al. "DeepFix: Fixing Common C Language Errors by Deep Learning." In: *AAAI*. 2017, pp. 1345–1351.

[10] Hideaki Hata, Emad Shihab, and Graham Neubig. "Learning to Generate Corrective Patches using Neural Machine Translation". In: *arXiv preprint 1812.07170* (2018).

[11] Vincent J Hellendoorn and Premkumar 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*. ACM. 2017, pp. 763–773.

[12] Jordan Henkel et al. "Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces". In: *Proceedings of ESEC/FSE*. 2018.

[13] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[14] Jiajun Jiang et al. "Shaping Program Repair Space with Existing Patches and Similar Code". In: (2018).

[15] René Just, Darioush Jalali, and Michael 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*. ACM. 2014, pp. 437–440.

[16] Jack Kiefer, Jacob Wolfowitz, et al. “Stochastic estimation of the maximum of a regression function”. In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[17] Guillaume Klein et al. “OpenNMT: Open-Source Toolkit for Neural Machine Translation”. In: *Proc. ACL*. 2017.

[18] Minh-Thang Luong et al. “Addressing the rare word problem in neural machine translation”. In: *arXiv preprint arXiv:1410.8206* (2014).

[19] Matias Martinez et al. “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset”. In: *Empirical Software Engineering* 22.4 (2017), pp. 1936–1964.

[20] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. “Angelix: Scalable multiline program patch synthesis via symbolic analysis”. In: *Proceedings of the 38th international conference on software engineering*. ACM. 2016, pp. 691–701.

[21] Martin Monperrus. “Automatic Software Repair: a Bibliography”. In: *ACM Computing Surveys* 51 (2017), pp. 1–24.

[22] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).

[23] Yewen Pu et al. “sk_p: a neural program corrector for MOOCs”. In: *CoRR* abs/1607.02902 (2016). arXiv: 1607.02902.

[24] Zichao Qi et al. “An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 24–36. ISBN: 978-1-4503-3620-8.

[25] Ripon K Saha et al. “Elixir: Effective object-oriented program repair”. In: *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE. 2017, pp. 648–659.

[26] Eddie Antonio Santos et al. “Syntax and sensibility: Using language models to detect and correct syntax errors”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 311–322.

[27] Mike Schuster and Kuldip K Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[28] Abigail See, Peter J. Liu, and Christopher D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks”. In: *CoRR* abs/1704.04368 (2017). arXiv: 1704.04368.

[29] Richard Shin, Illia Polosukhin, and Dawn Song. “Towards Specification-Directed Program Repair”. In: *ICLR Workshop*. 2018.

[30] Edward K Smith et al. “Is the cure worse than the disease? overfitting in automated program repair”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 532–543.

[31] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[32] Michele Tufano et al. “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 832–837.

[33] Michele Tufano et al. *An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation*. Tech. rep. arXiv:1812.08693, 2018.

[34] Ke Wang, Rishabh Singh, and Zhendong Su. “Dynamic Neural Program Embedding for Program Repair”. In: *arXiv preprint arXiv:1711.07163* (2017).

[35] Ming Wen et al. “Context-Aware Patch Generation for Better Automated Program Repair”. In: ICSE. 2018.

[36] Martin White et al. “Deep Learning Code Fragments for Code Clone Detection”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 87–98. ISBN: 978-1-4503-3845-5.

[37] Martin White et al. “Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities”. In: *Proceedings of SANER*. 2019.

[38] Qi Xin and Steven P Reiss. “Leveraging syntax-related code for automated program repair”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 660–670.

[39] Yingfei Xiong et al. “Precise condition synthesis for program repair”. In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, pp. 416–426.

A APPENDIX, FIGURES

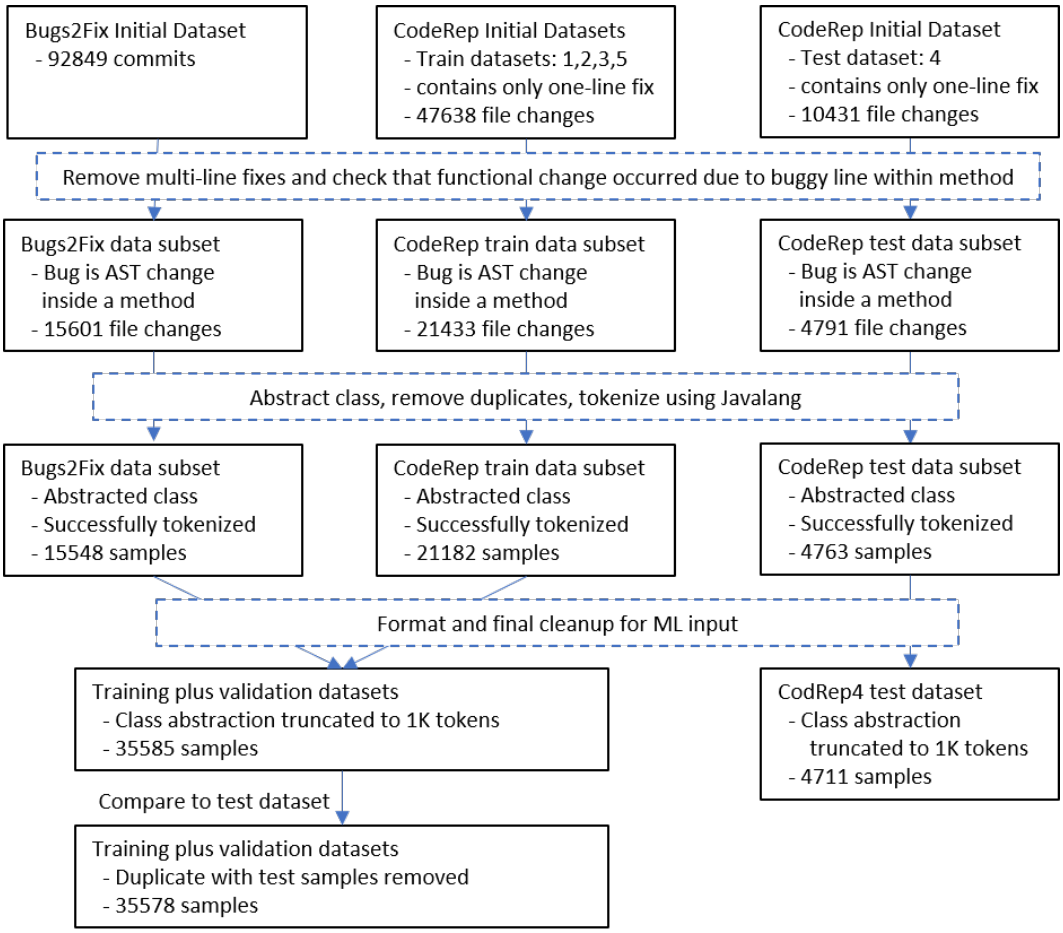


Figure 12: Overview of the construction of the training and testing datasets.

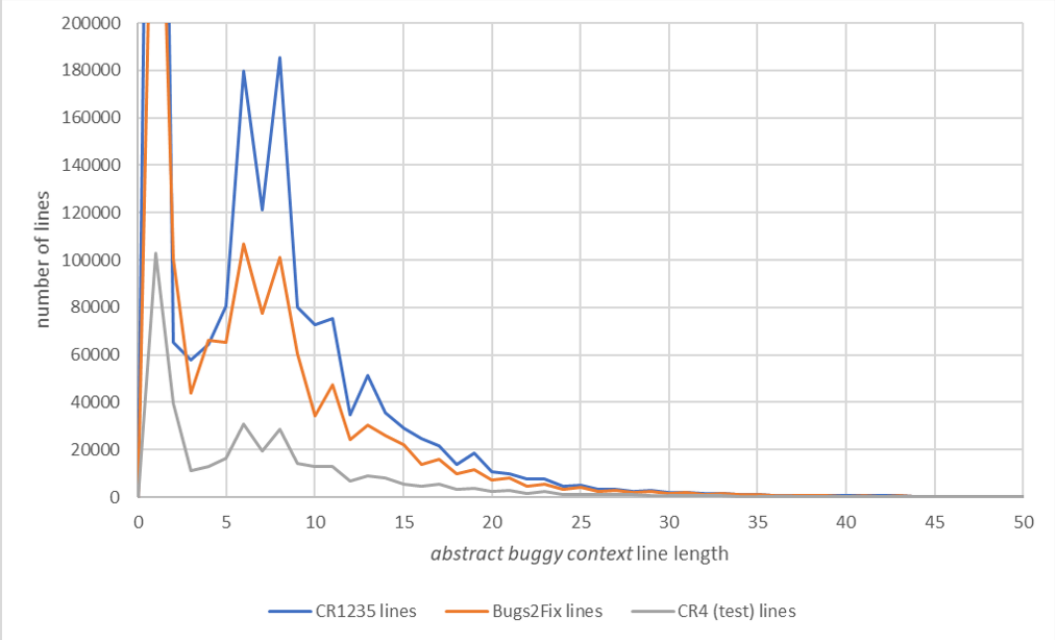


Figure 13: Histogram showing the number of lines in datasets at a given token count

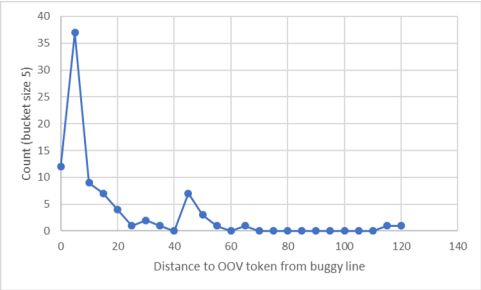


Figure 14: Histogram showing distance from buggy line when a token is copied from the buggy method for a patch.

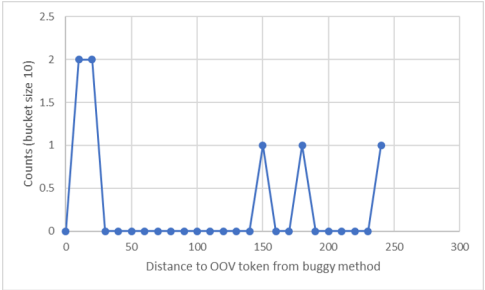


Figure 15: Histogram showing distance from buggy method when a token is copied from the buggy class for a patch.